

# IKI10400 • Struktur Data & Algoritma: Binary Search Tree

**Fakultas Ilmu Komputer • Universitas Indonesia**

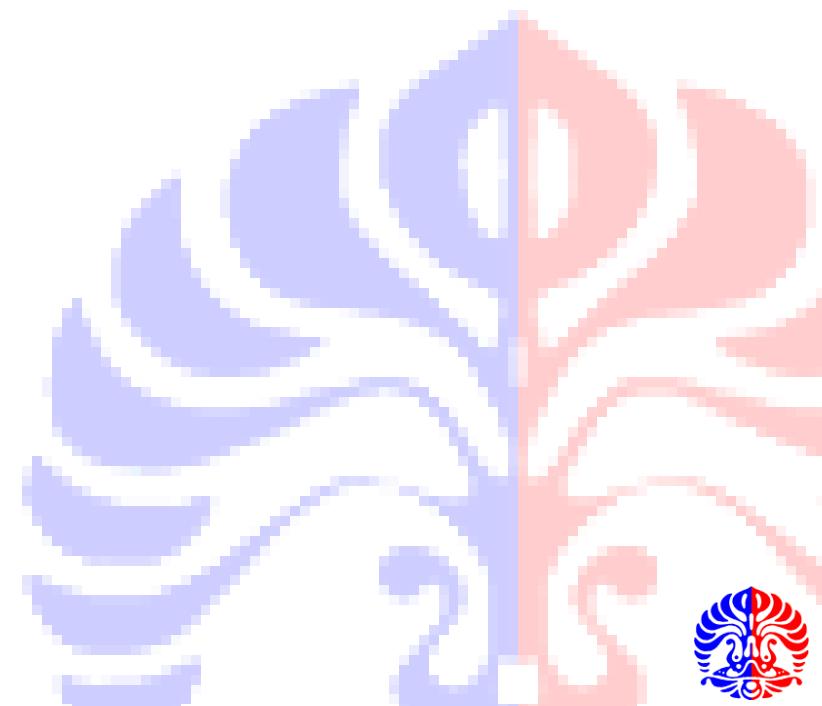
*Slide acknowledgments:*

Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung, Tisha Melia



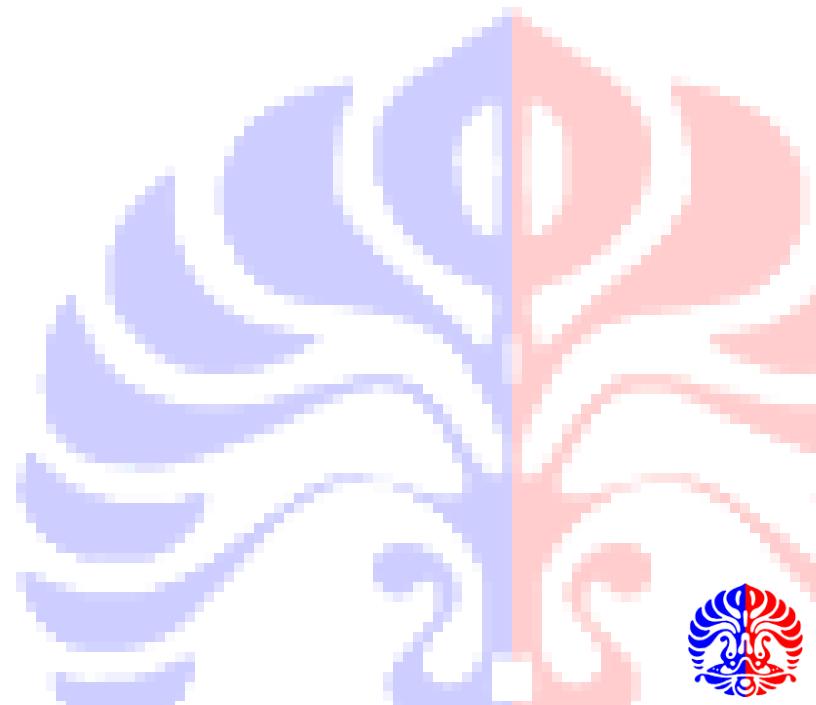
# Tujuan

- Memahami sifat dari Binary Search Tree (BST)
- Memahami operasi-operasi pada BST
- Memahami kelebihan dan kekurangan dari BST



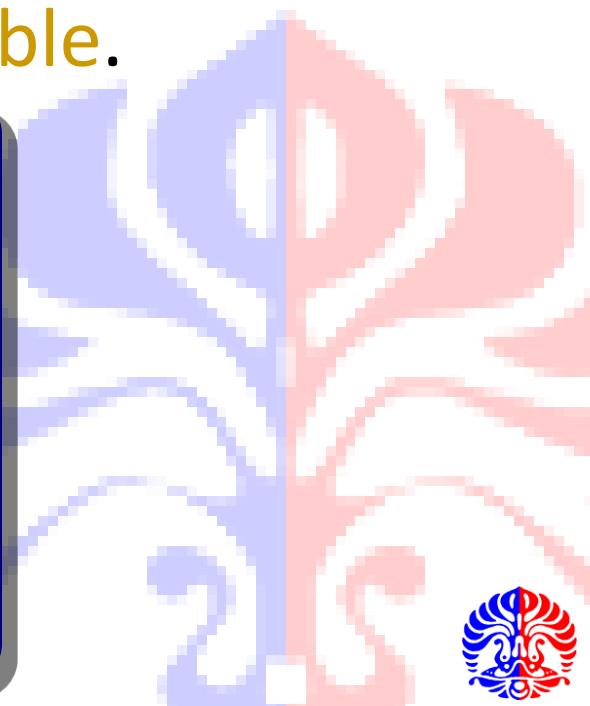
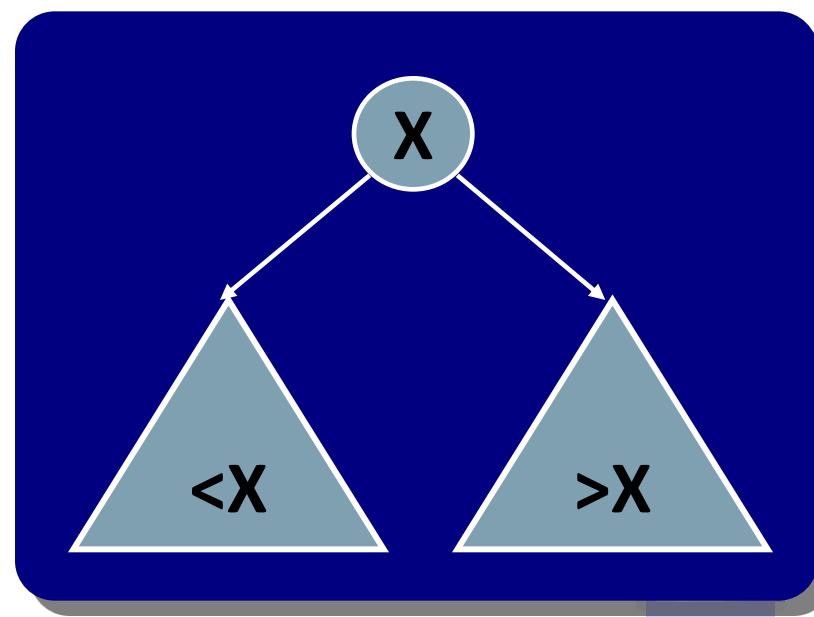
# Outline

- Properties of Binary Search Tree (BST)
- Operation
  - insert
  - find
  - remove

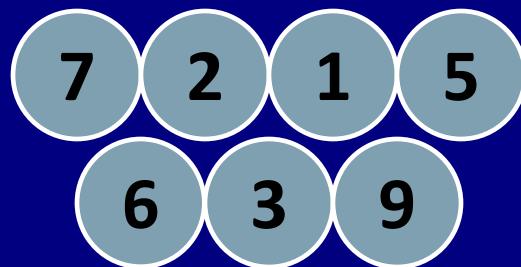


# Properties of Binary Search Tree

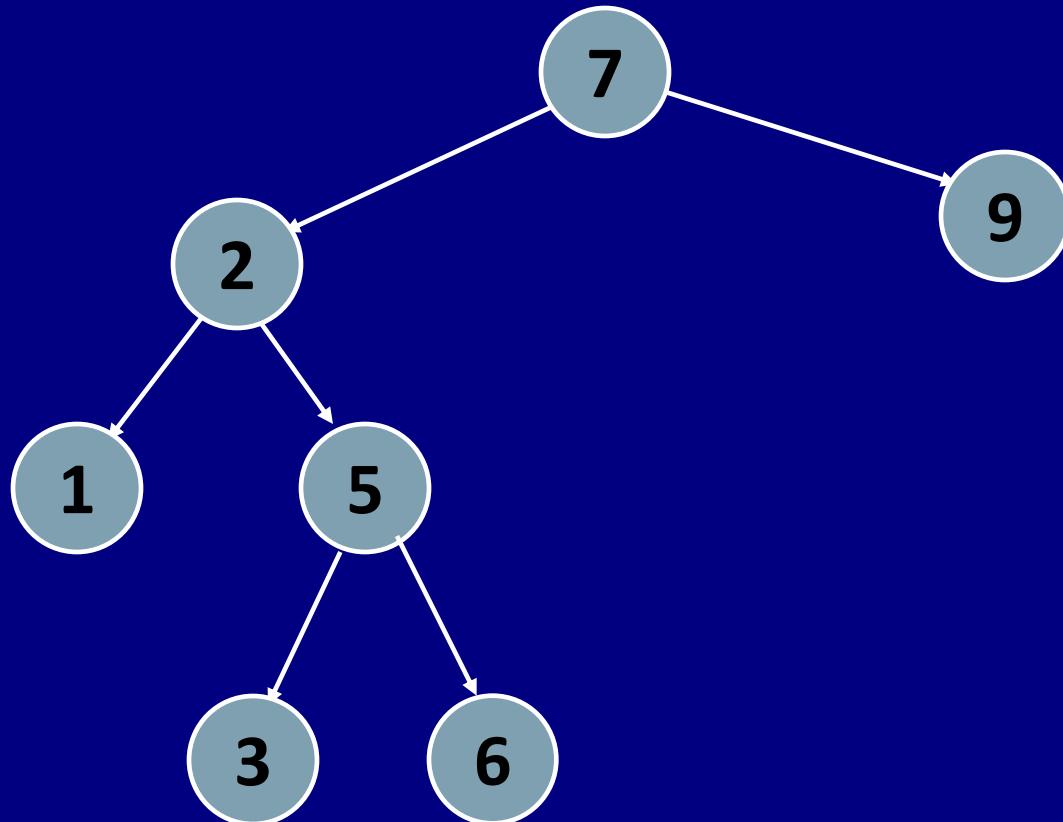
- Untuk setiap node **X** pada tree, nilai elemen pada subtree **sebelah kiri selalu lebih kecil dari elemen node X** dan nilai elemen pada **subtree sebelah kanan selalu lebih besar dari elemen node X**.
- Jadi object *elemen harus comparable*.



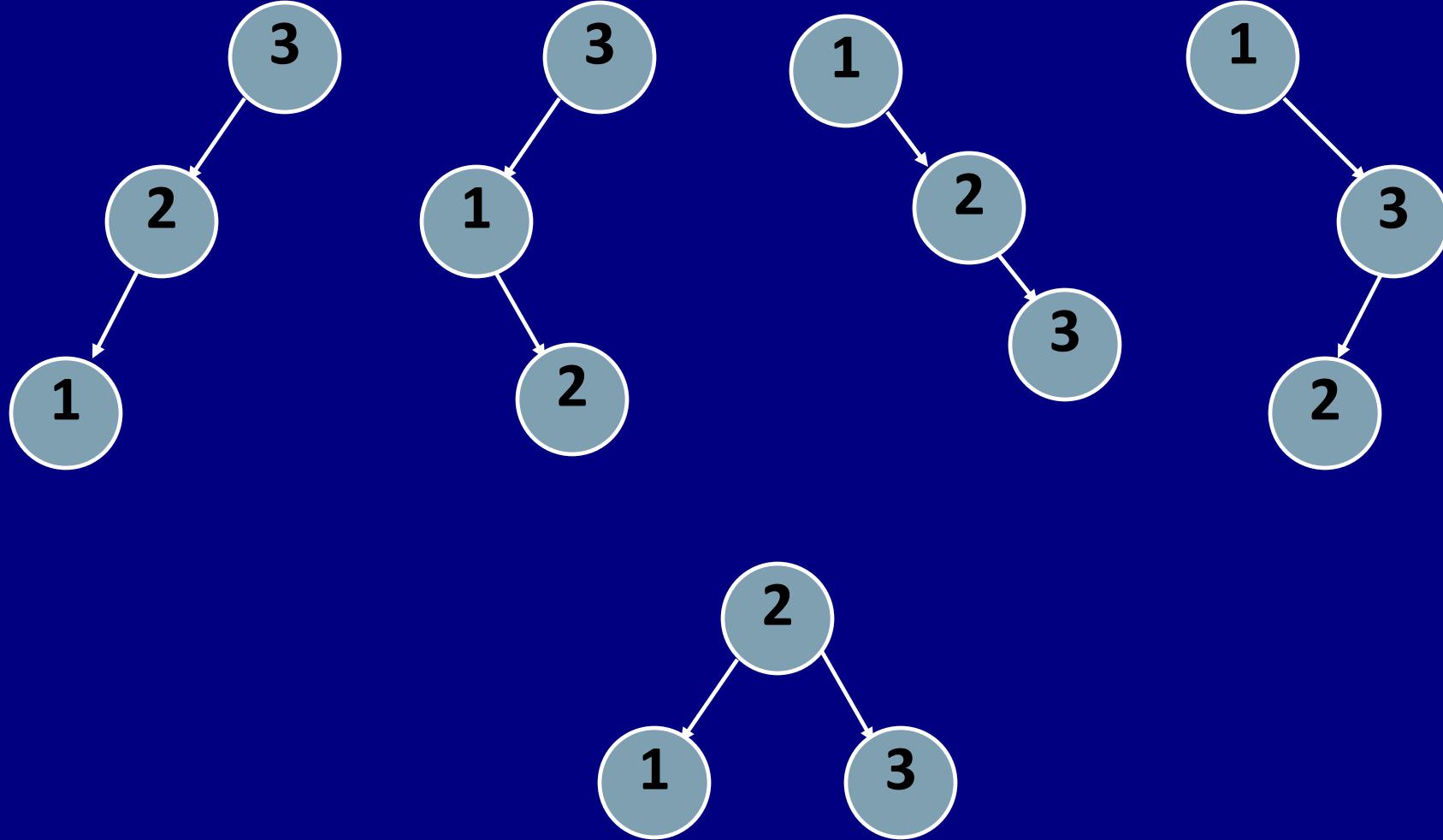
# Binary Search Tree



# Binary Search Tree

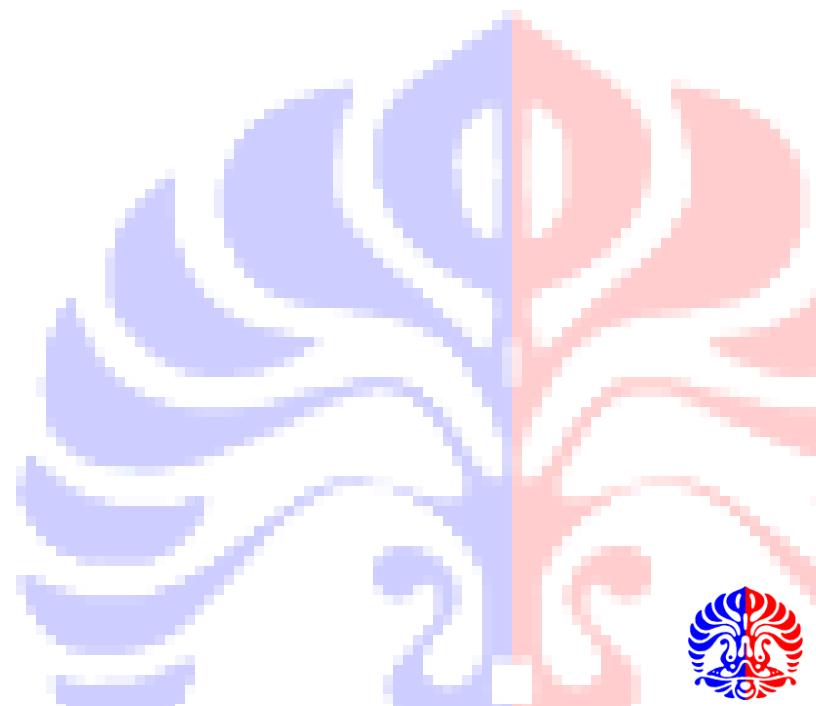


# Binary Search Tree



# Basic Operations

- insert
- findMin and findMax
- remove
- cetak terurut



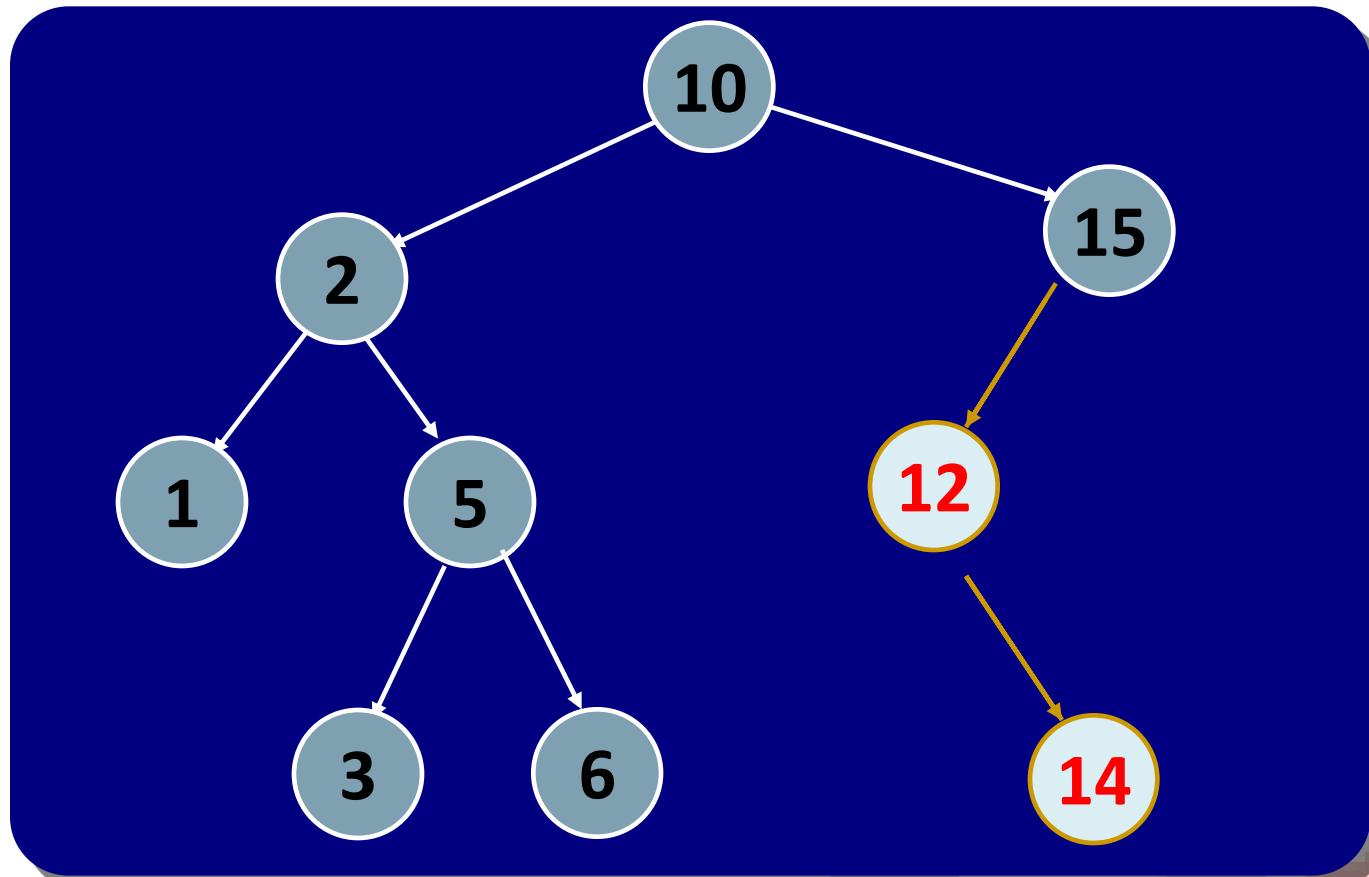
# Print InOrder

```
class BinaryNode {  
    void printInOrder( )  
    {  
        if( left != null )  
            left.printInOrder( );                                // Left  
        System.out.println( element );                         // Node  
        if( right != null )  
            right.printInOrder( );                             // Right  
    }  
}  
  
class BinaryTree {  
    public void printInOrder( )  
    {  
        if( root != null )  
            root.printInOrder( );  
    }  
}
```



# Insertion

- Penyisipan sebuah elemen baru dalam binary search tree, elemen tersebut pasti akan menjadi leaf



# Insertion: algorithm

- Menambah elemen X pada binary search tree:
  - mulai dari root.
  - Jika X lebih kecil dari root, maka X harus diletakkan pada sub-tree sebelah kiri.
  - jika X lebih besar dari root, then X harus diletakkan pada sub-tree sebelah kanan.
- Ingat bahwa: sebuah sub tree adalah juga sebuah tree. Maka, proses penambahan elemen pada sub tree adalah sama dengan penambahan pada seluruh tree. (melalui root tadi)
  - Apa hubungannya?
  - permasalahan ini cocok diselesaikan secara rekursif.



# Insertion

```
BinaryNode insert(int x, BinaryNode t)
{
    if (t == null) {
        t = new BinaryNode (x, null, null);
    } else if (x < t.element) {
        t.left = insert (x, t.left);
    } else if (x > t.element) {
        t.right = insert (x, t.right);
    } else {
        throw new DuplicateItem("exception");
    }
    return t;
}
```

# FindMin

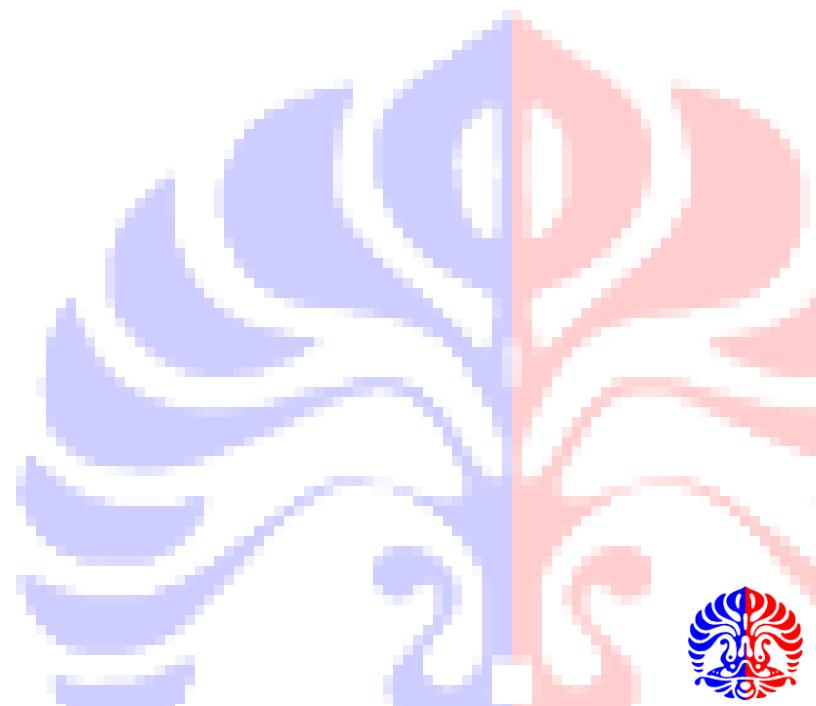
- Mencari node yang memiliki nilai terkecil.
- Algorithm:
  - ke kiri terus sampai buntu....:)
- Code:

```
BinaryNode findMin (BinaryNode t)
{
    if (t == null) throw exception;

    while (t.left != null) {
        t = t.left;
    }
    return t;
}
```

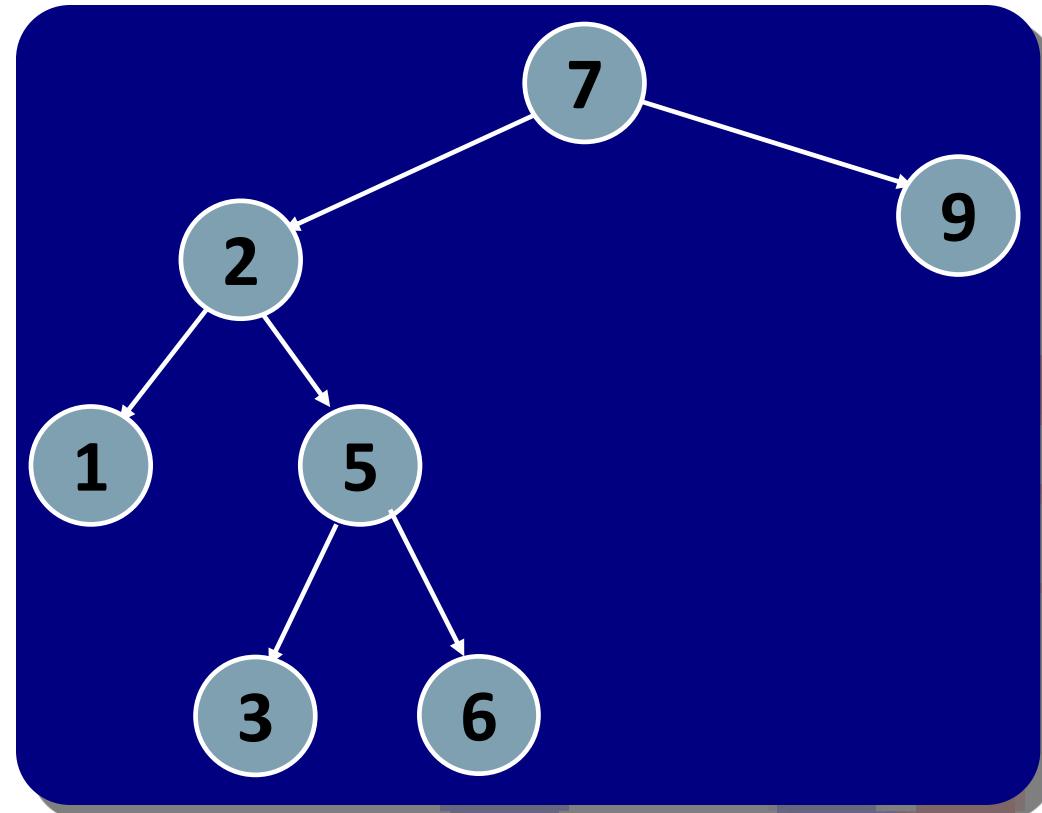
# FindMax

- Mencari node yang memiliki nilai terbesar
- Algorithm?
- Code?



# Find

- Diberikan sebuah nilai yang harus dicari dalam sebuah BST. Jika ada elemen tersebut, return node tersebut. Jika tidak ada, return null.
- Algorithm?
- Code?

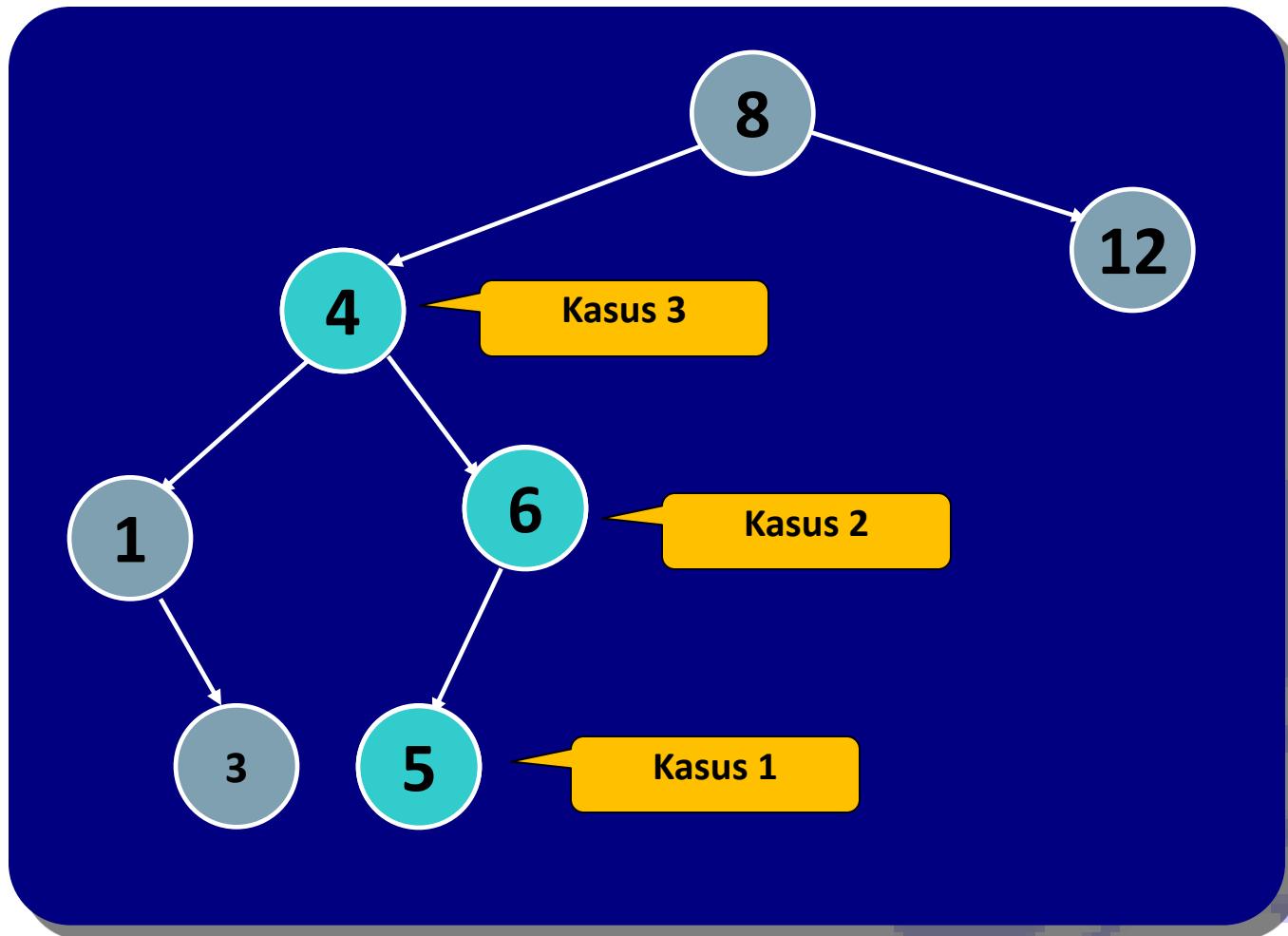


# Remove

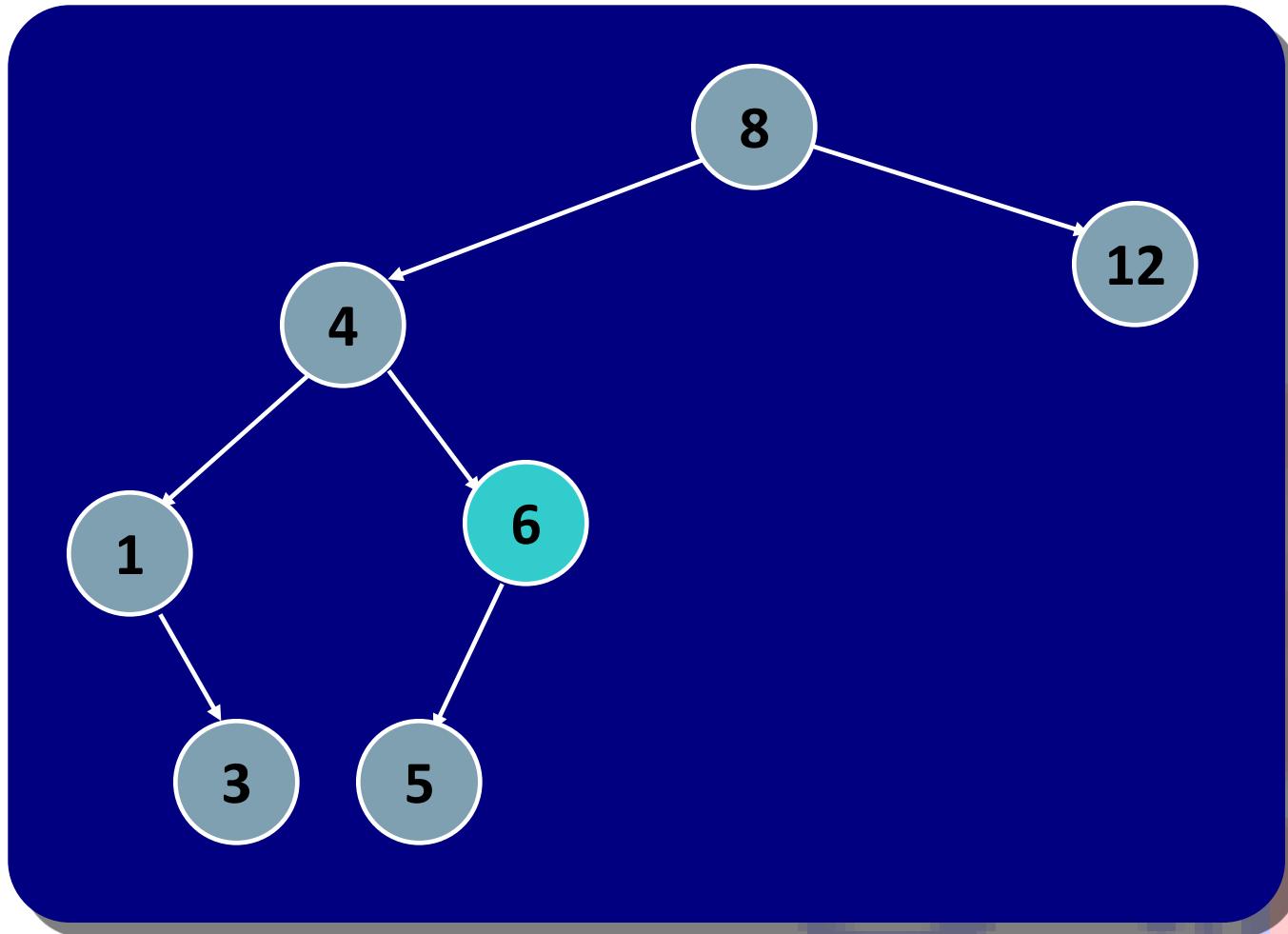
- **Kasus 1:** jika node adalah leaf (tidak punya anak), langsung saja dihapus.
- **Kasus 2:** jika node punya satu anak: node parent menjadikan anak dari node yang dihapus (cucu) sebagai anaknya. (mem-by-pass node yang dihapus).
- **Kasus 3:** jika node punya dua anak.....



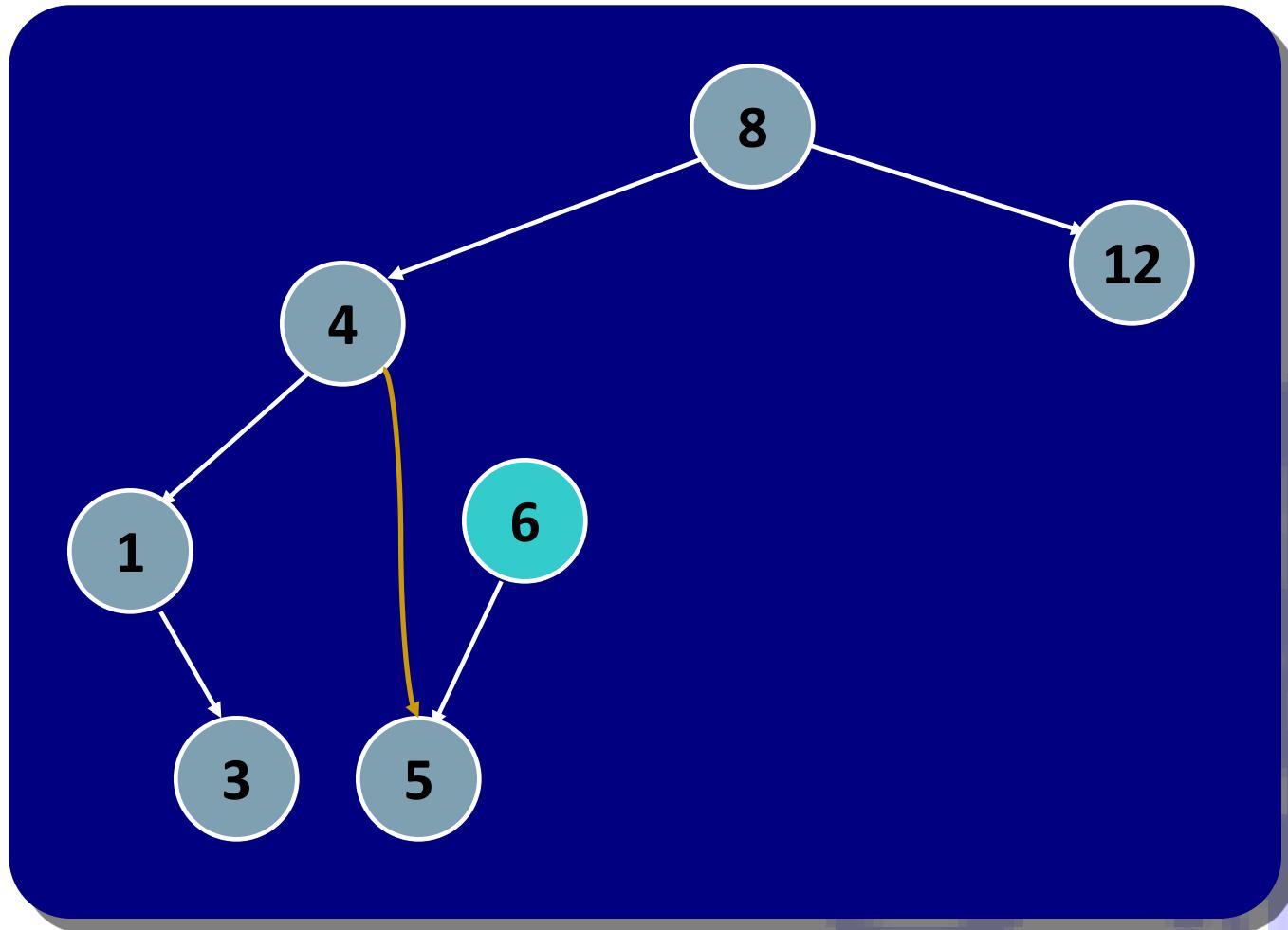
# Remove



# Removing 6



# After 6 removed



# Remove (lanj.)

## ■ Bagaimana bila node punya dua anak?

1. Hapus isi node (tanpa mendelete node)

2. Gantikan posisinya dengan:

- **Succesor Inorder** node terkecil dari sub tree kanan, dilanjutkan dengan melakukan removeMin di subtree kanan.

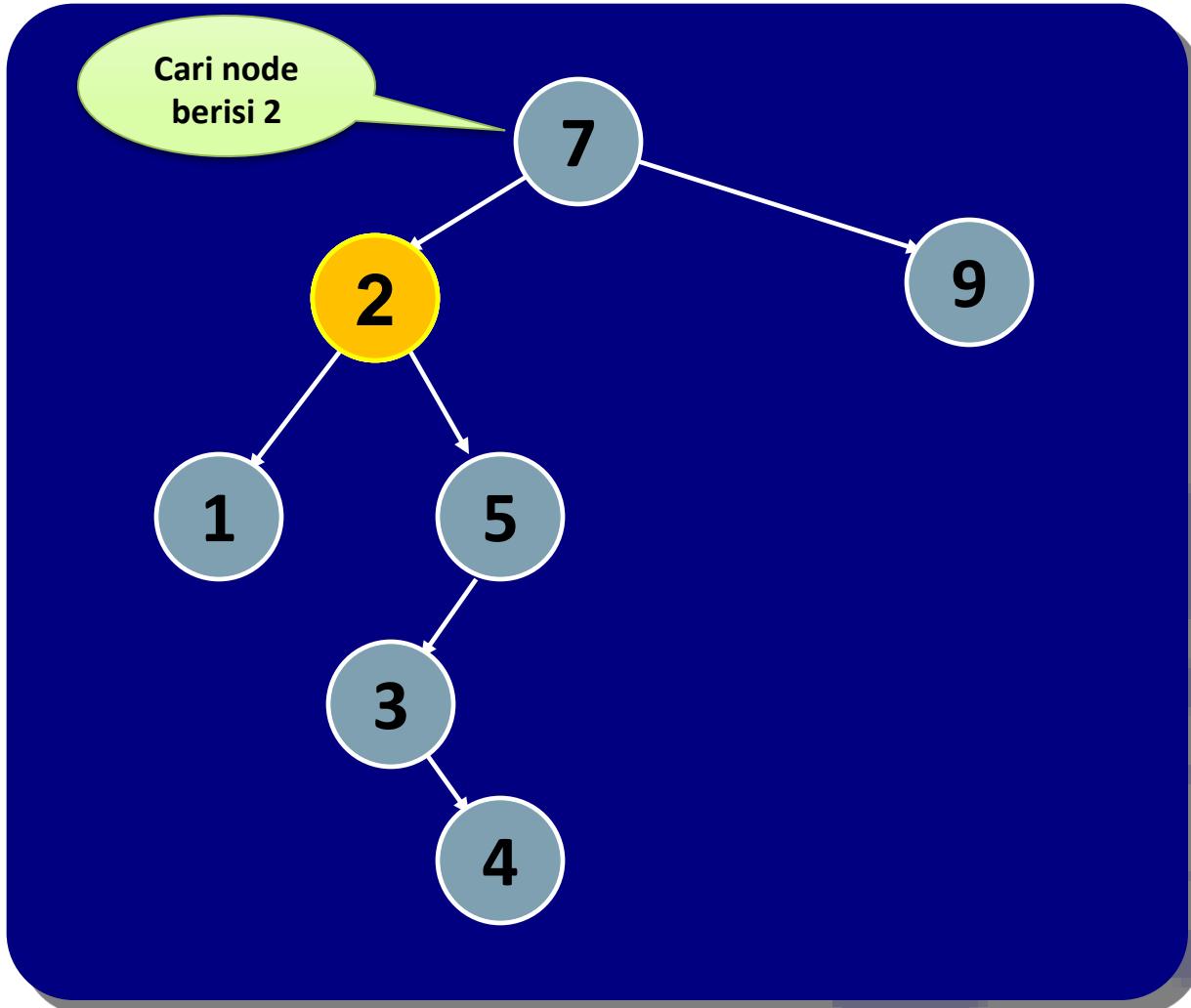
[Alternatif: dengan kaidah **Predecesor Inorder**,

2. Gantikan posisinya dengan:

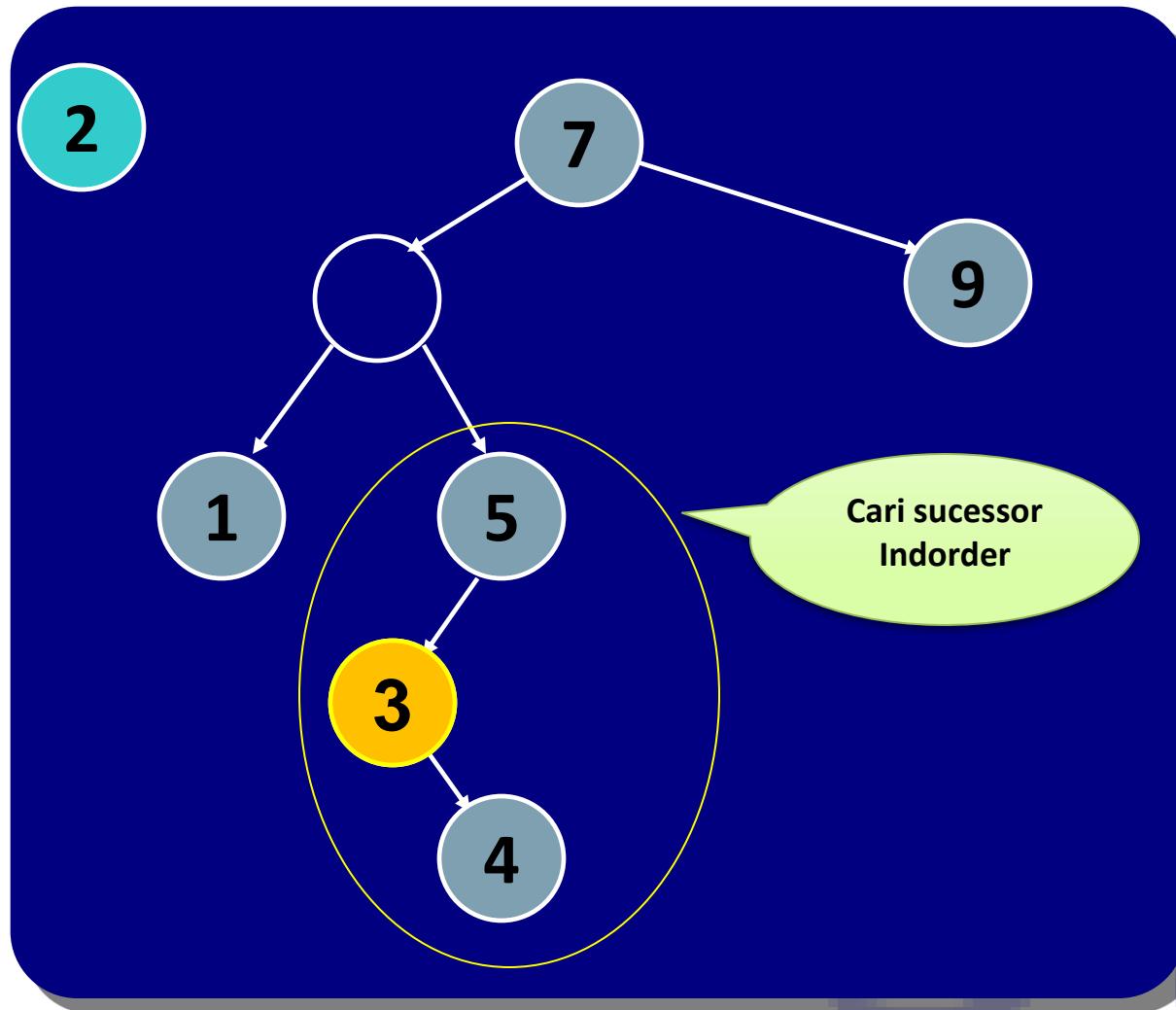
- **Predecesor Inorder**, node terbesar dari sub tree kiri, dilanjutkan dengan melakukan removeMax di subtree kiri.]



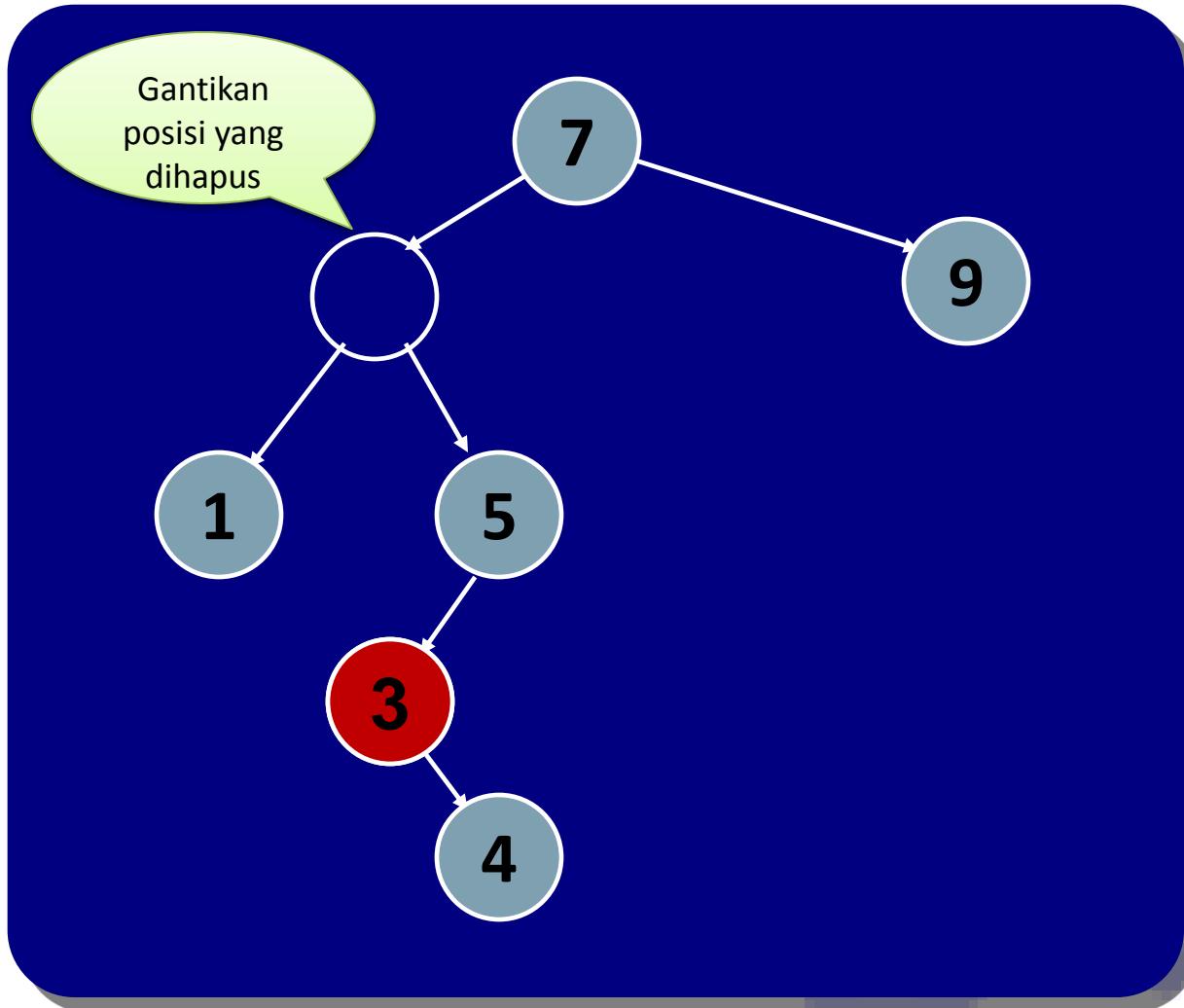
# Removing 2 (Successor Inorder)



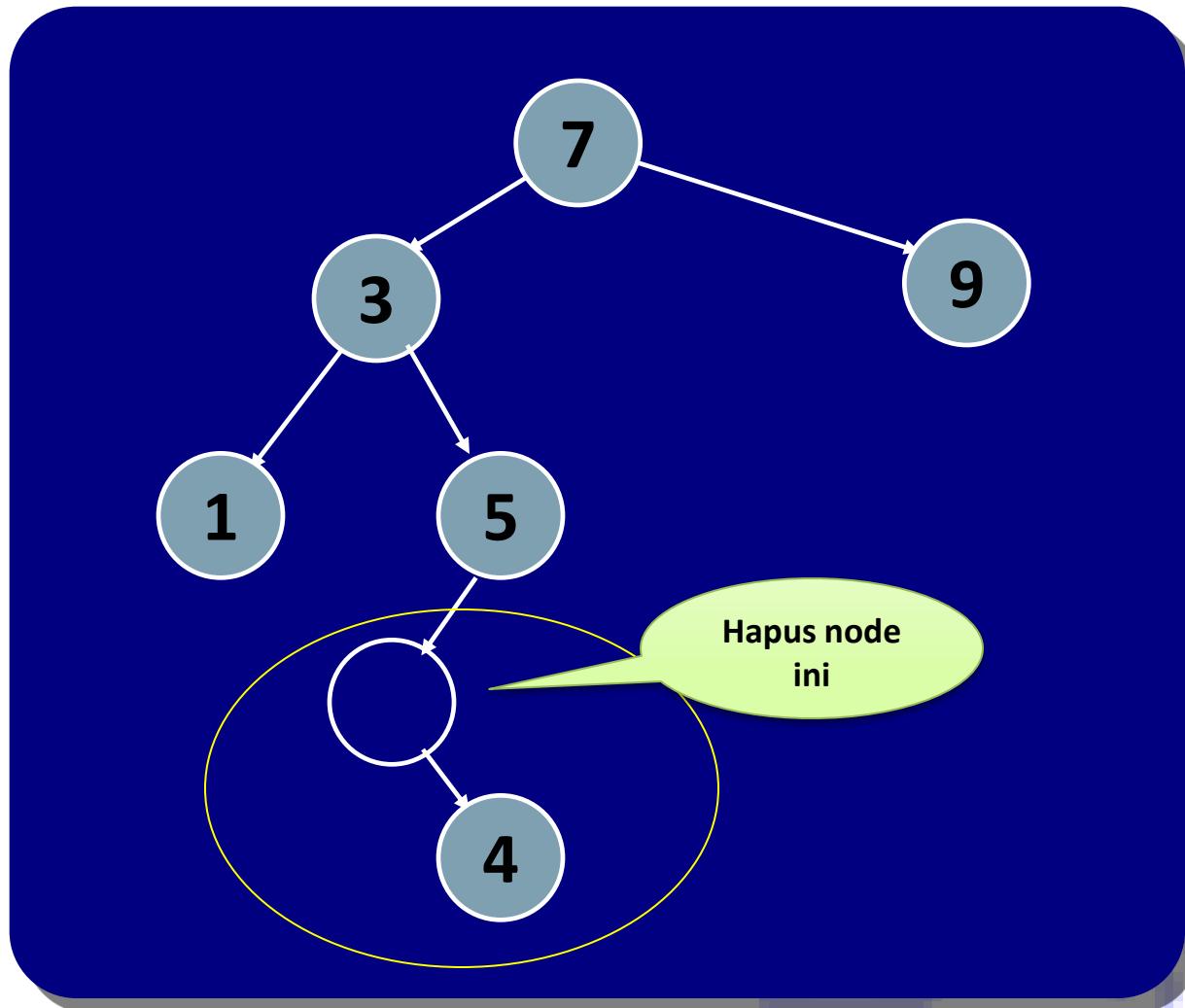
# Removing 2 (Successor Inorder)



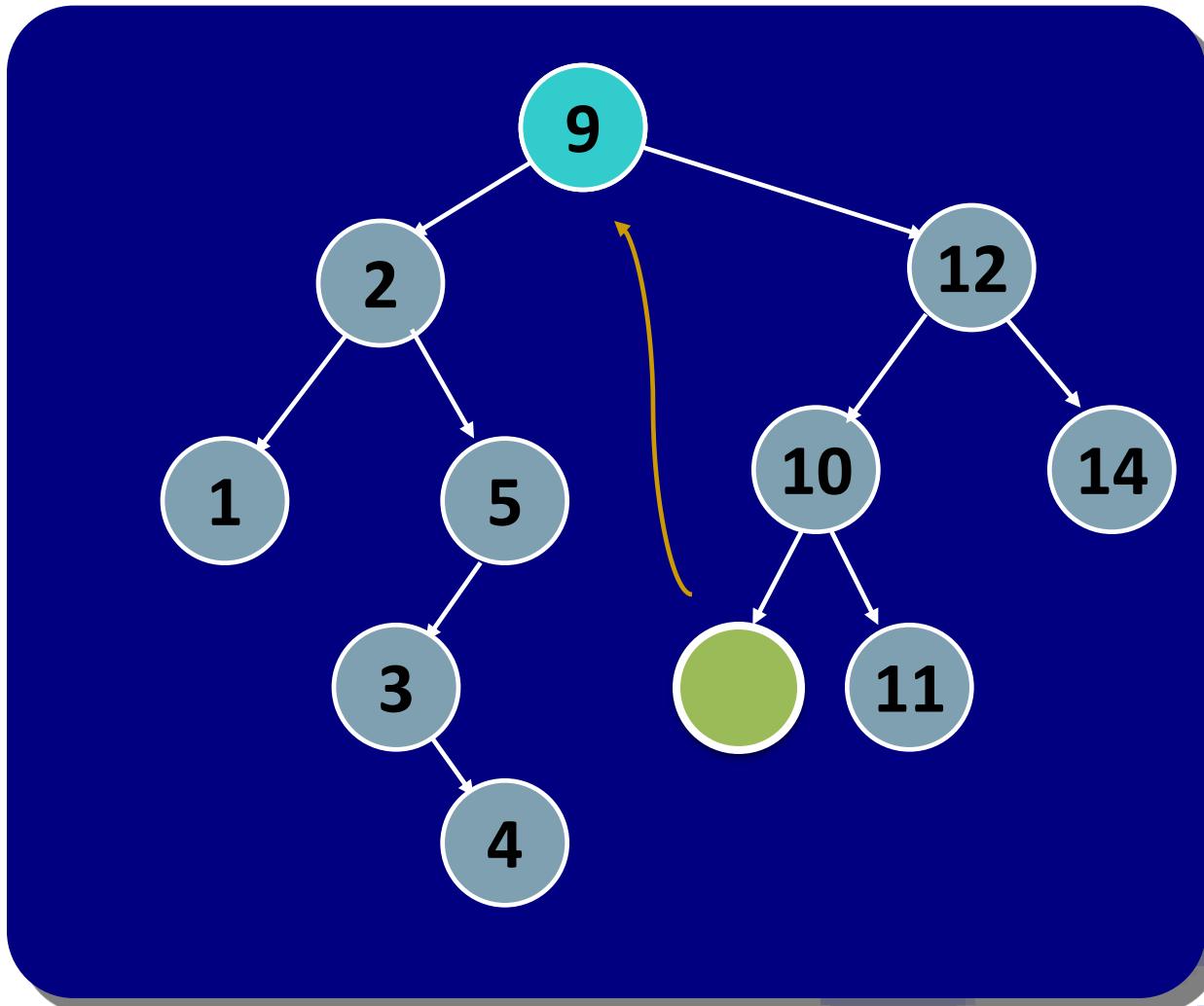
# Removing 2 (Successor Inorder)



# After 2 deleted



# Removing Root



# removeMin

```
BinaryNode removeMin(BinaryNode t)
{
    if (t == null) throw exception;

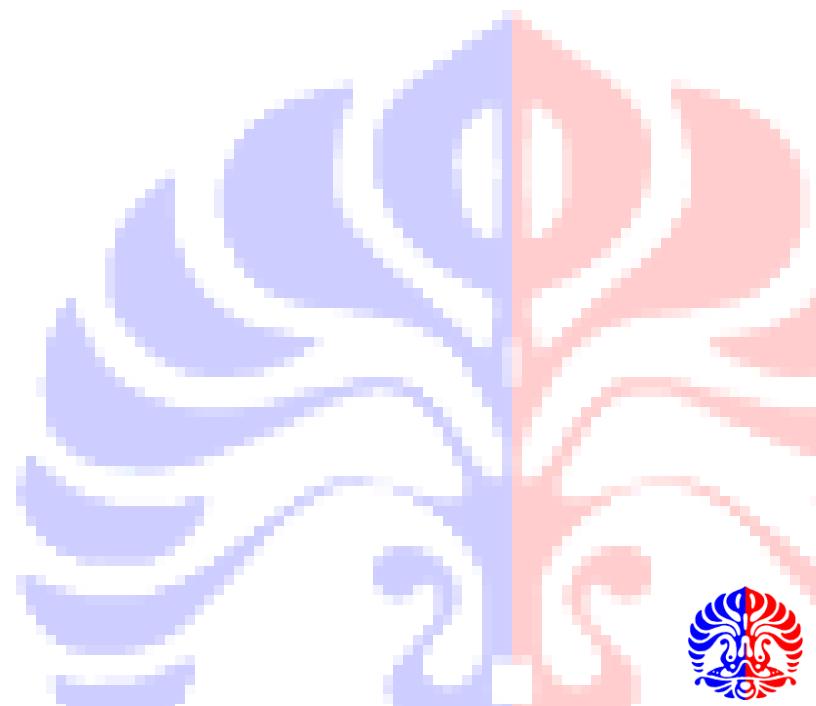
    if (t.left != null) {
        t.left = removeMin (t.left);
        return t;
    } else {
        return t.right;
    }
}
```

# Remove

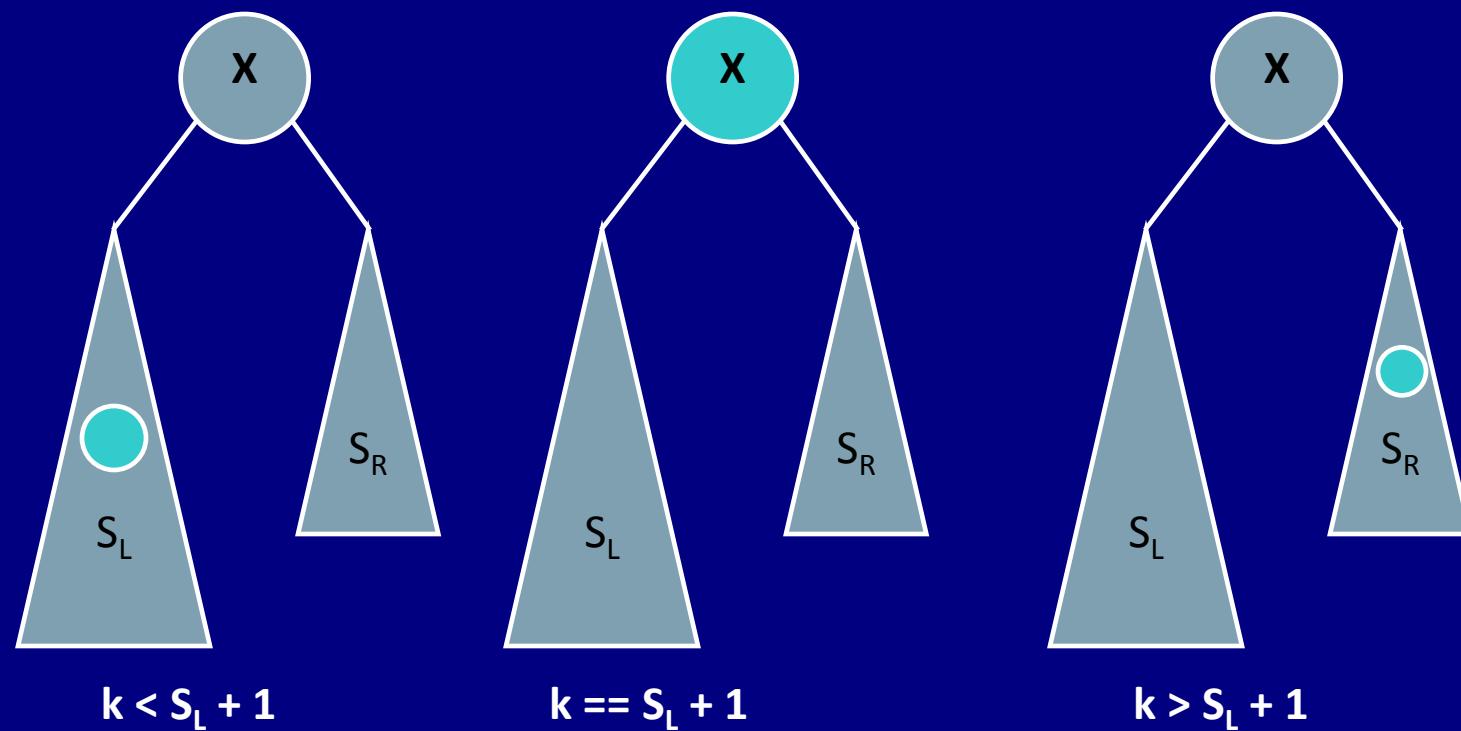
```
BinaryNode remove(int x, BinaryNode t) {  
    if (t == null) throw exception;  
    if (x < t.element) {  
        t.left = remove(x, t.left);  
    } else if (x > t.element) {  
        t.right = remove(x, t.right);  
    } else if (t.left != null && t.right != null) {  
        t.element = findMin(t.right).element;  
        t.right = removeMin(t.right);  
    } else {  
        t = (t.left != null) ? t.left : t.right;  
    }  
    return t;  
}
```

# removeMax

■ code?



# Find k-th element



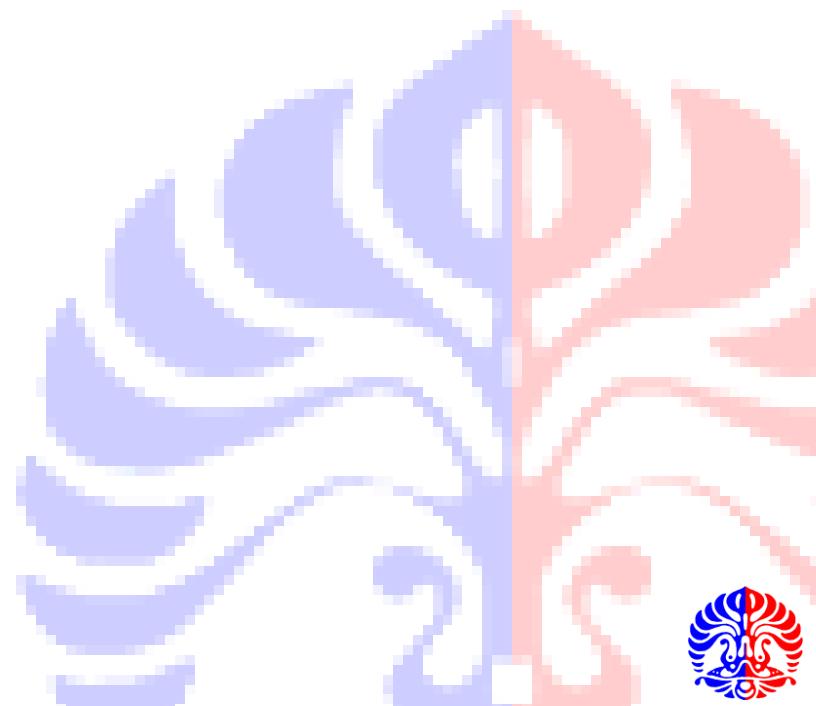
# Find k-th element

```
BinaryNode findKth(int k, BinaryNode t)
{
    if (t == null) throw exception;
    int leftSize = (t.left != null) ?
        t.left.size : 0;

    if (k <= leftSize ) {
        return findKth (k, t.left);
    } else if (k == leftSize + 1) {
        return t;
    } else {
        return findKth ( k - leftSize - 1, t.right);
    }
}
```

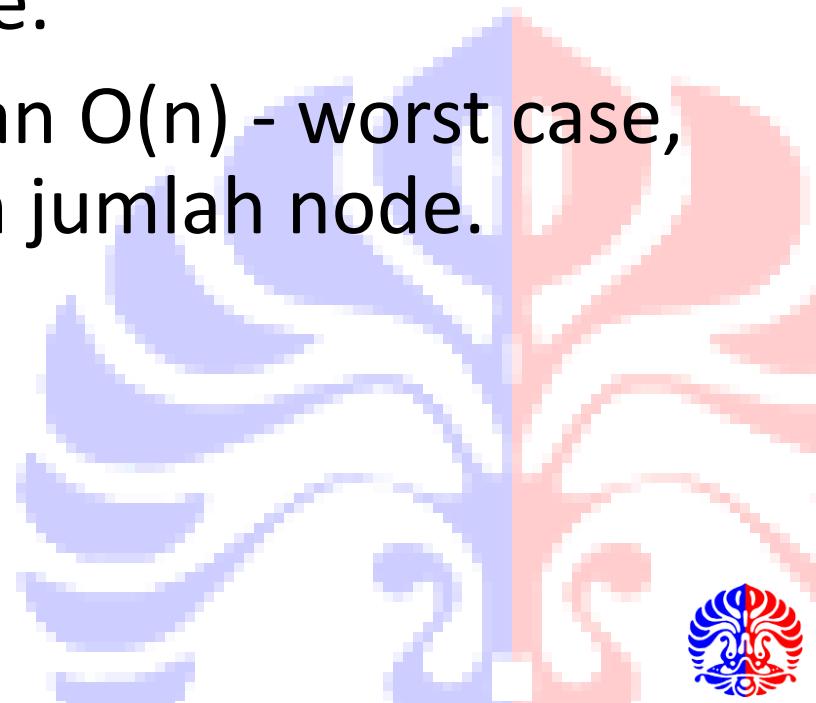
# Analysis

- Running time:
  - insert?
  - Find min?
  - remove?
  - Find?
- Worst case:  $O(n)$



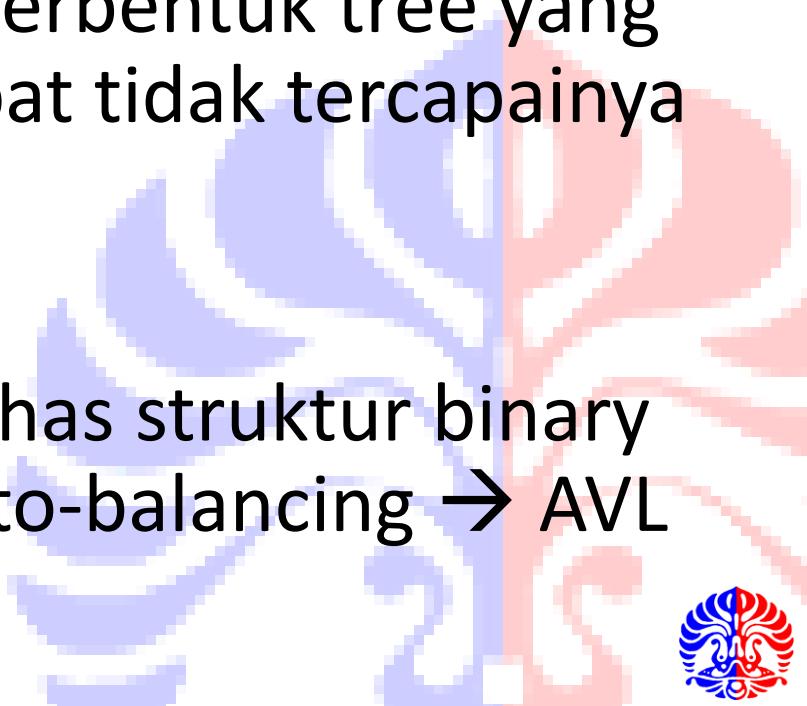
# Rangkuman

- Binary Search Tree menjamin urutan elemen pada tree.
- Tiap node harus comparable
- Semua operasi membutuhkan  $O(\log n)$  - average case, saat tree relatif balance.
- Semua operasi membutuhkan  $O(n)$  - worst case, tinggi dari tree sama dengan jumlah node.



# Selanjutnya:

- Sejauh ini struktur Binary Search terbentuk dengan asumsi data cukup acak sehingga seluruh bagian tree akan cukup terisi.
- Benarkah asumsi tersebut?
- Jika tidak benar, maka akan terbentuk tree yang “tidak balance” yang berakibat tidak tercapainya performance  $O(\log n)$
- Solusi?
- Dalam kuliah yang akan dibahas struktur binary tree dengan kemampuan auto-balancing → AVL tree



# IKI10400 • Struktur Data & Algoritma: *AVL Tree*

**Fakultas Ilmu Komputer • Universitas Indonesia**

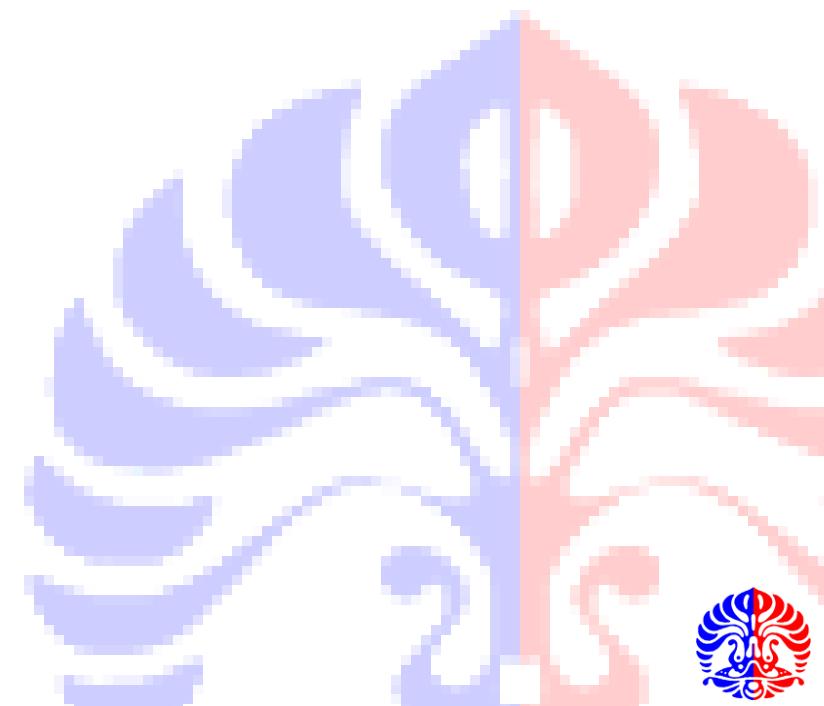
*Slide acknowledgments:*

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



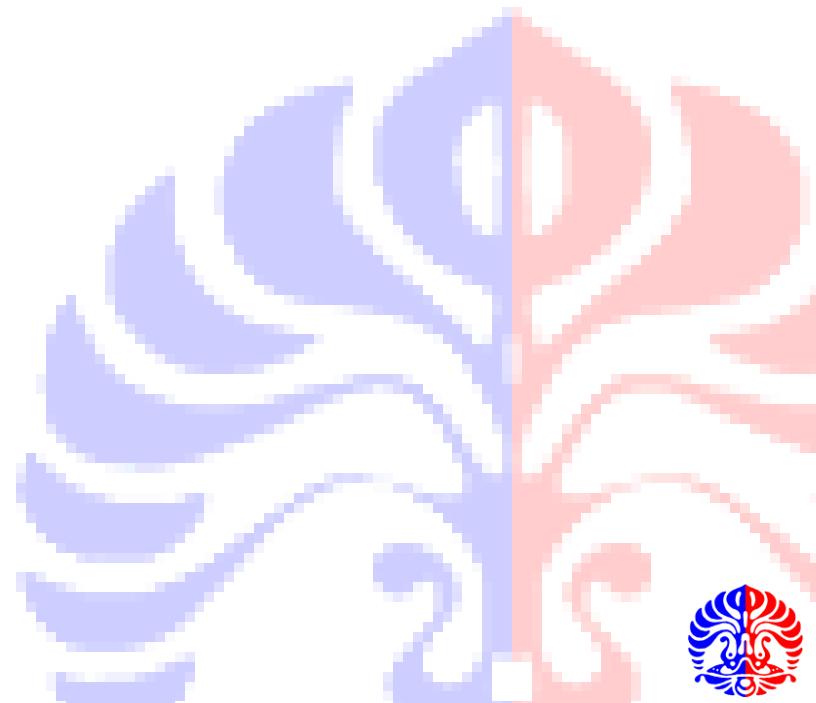
# Tujuan

- Memahami variant dari Binary Search Tree yang balanced



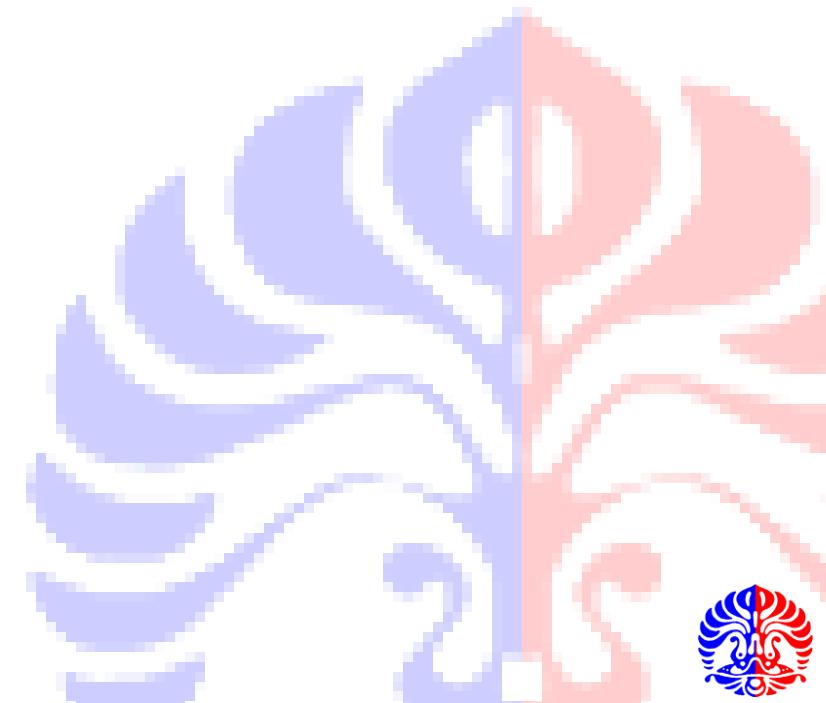
# Outline

- AVL Tree
  - Definition
  - Property
  - Operations



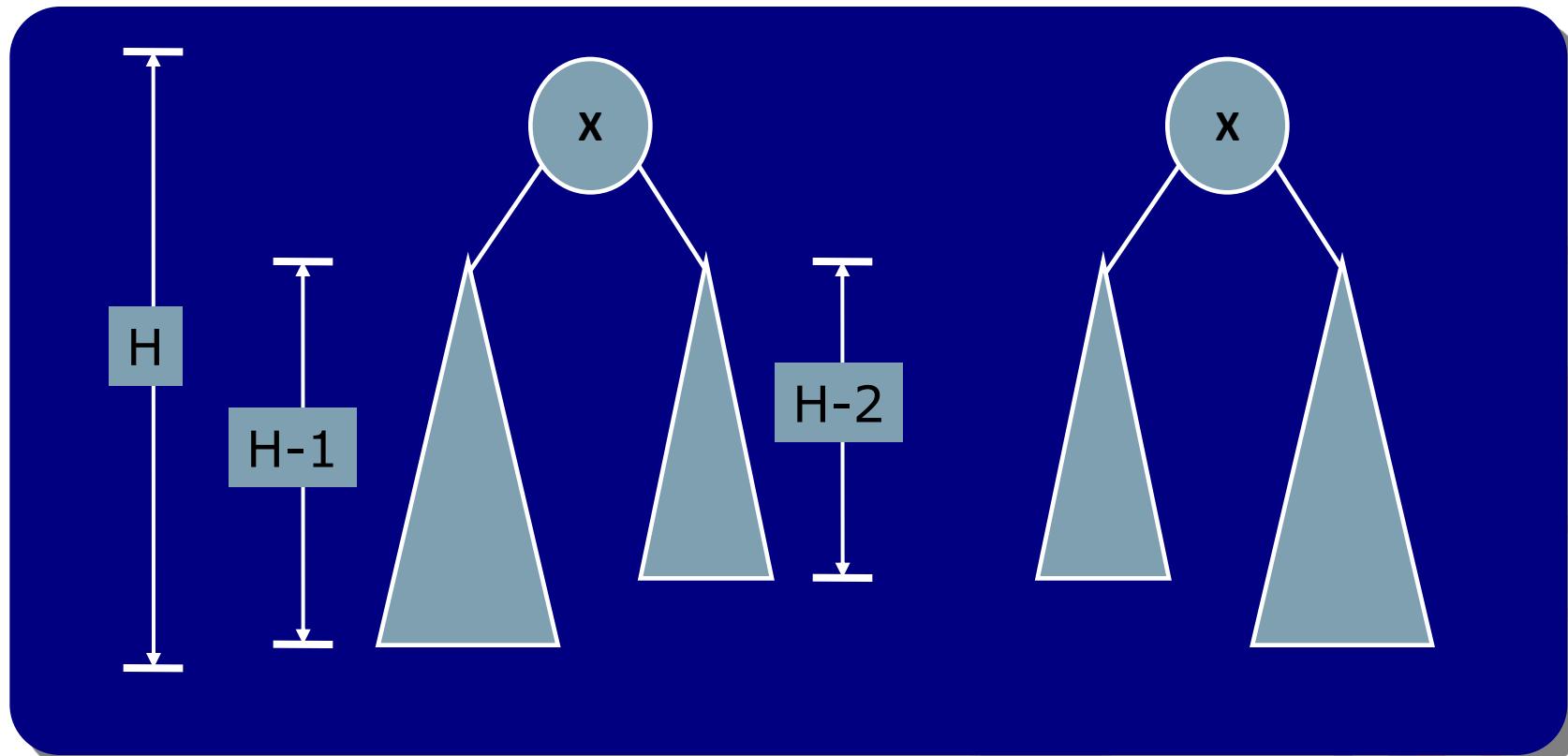
# Motivasi

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

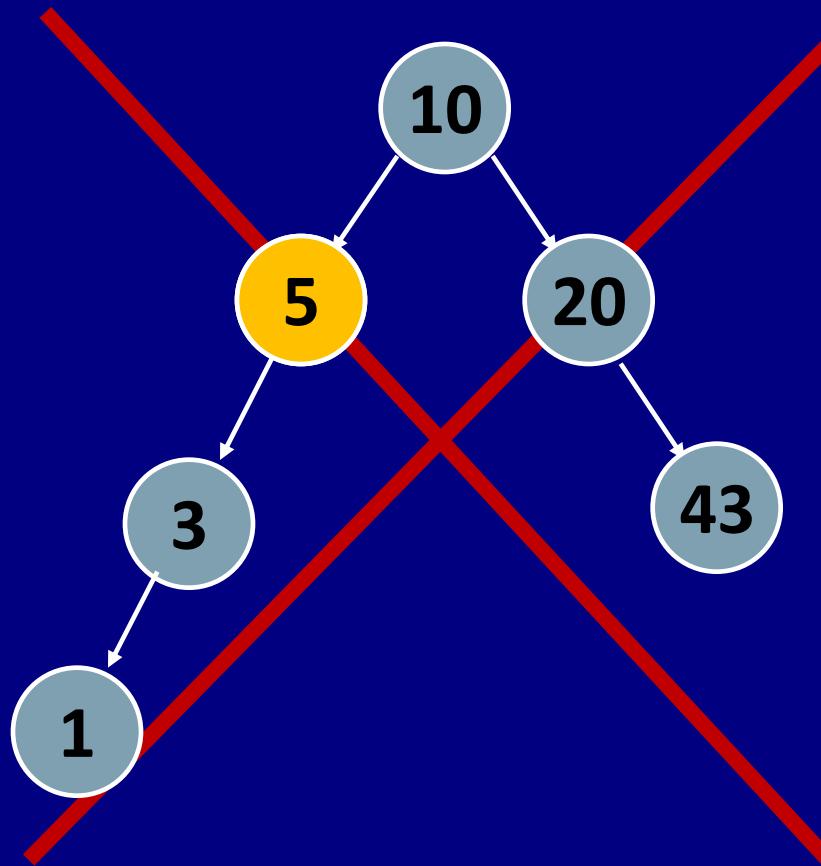
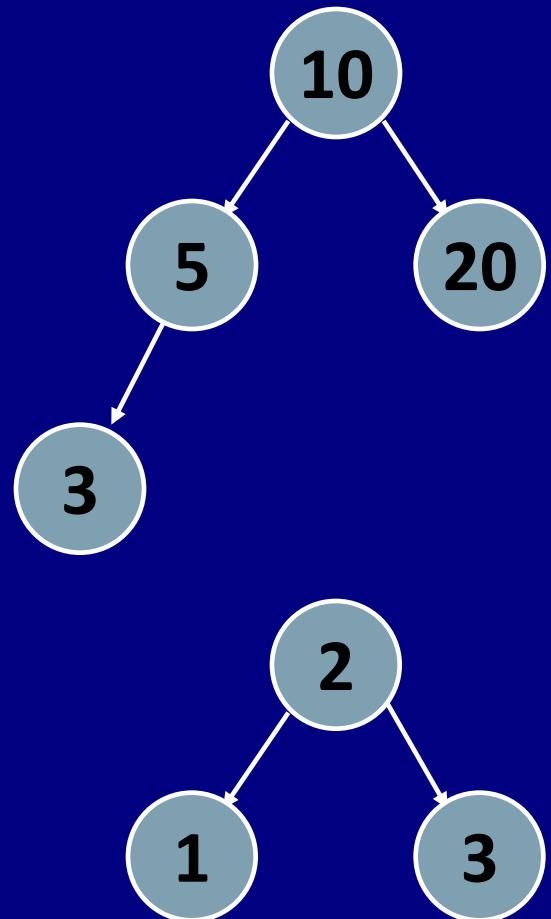


# AVL Trees

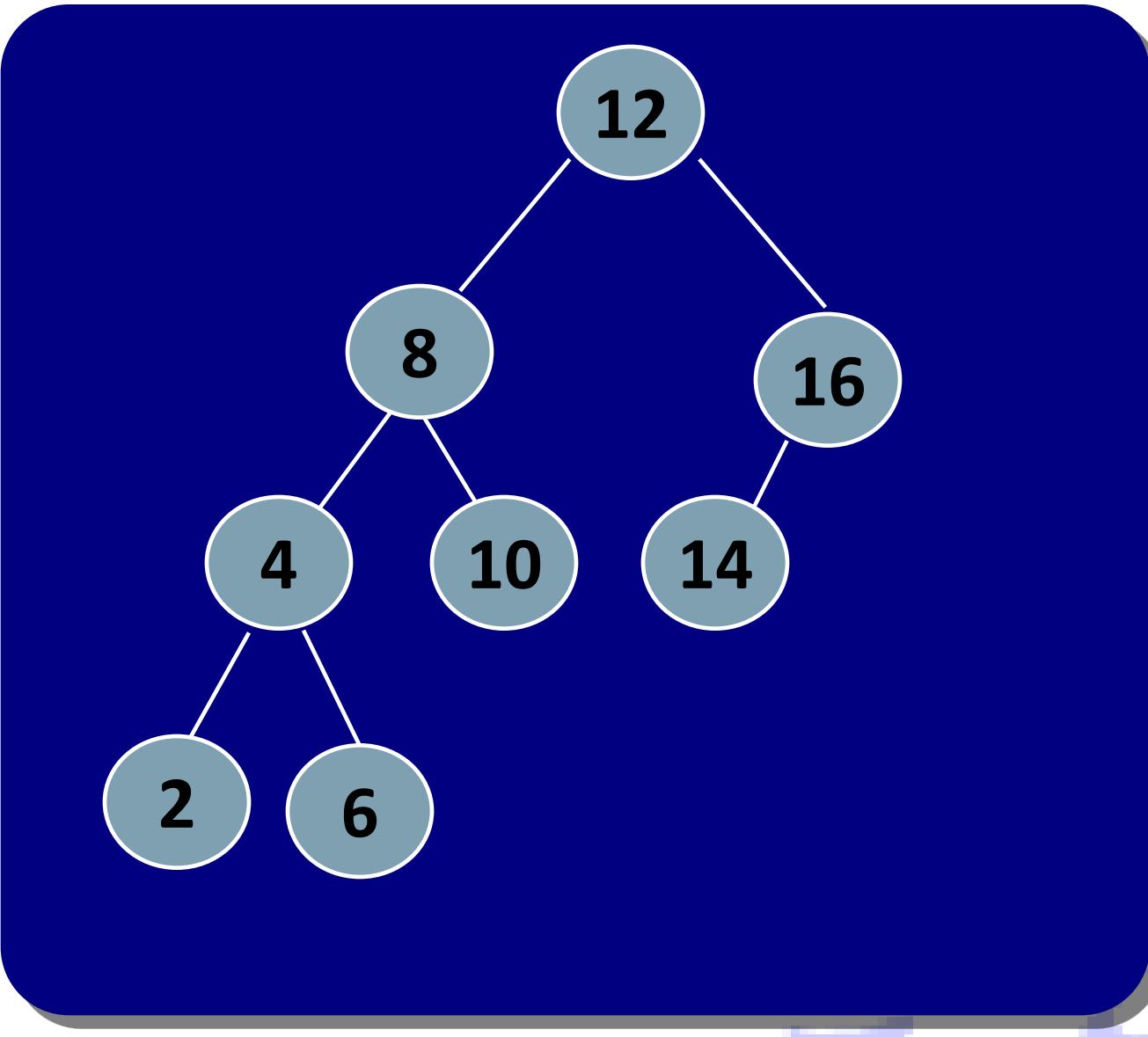
- Untuk setiap node dalam tree, ketinggian subtree di anak kiri dan subtree di anak kanan hanya berbeda maksimum 1.



# AVL Trees

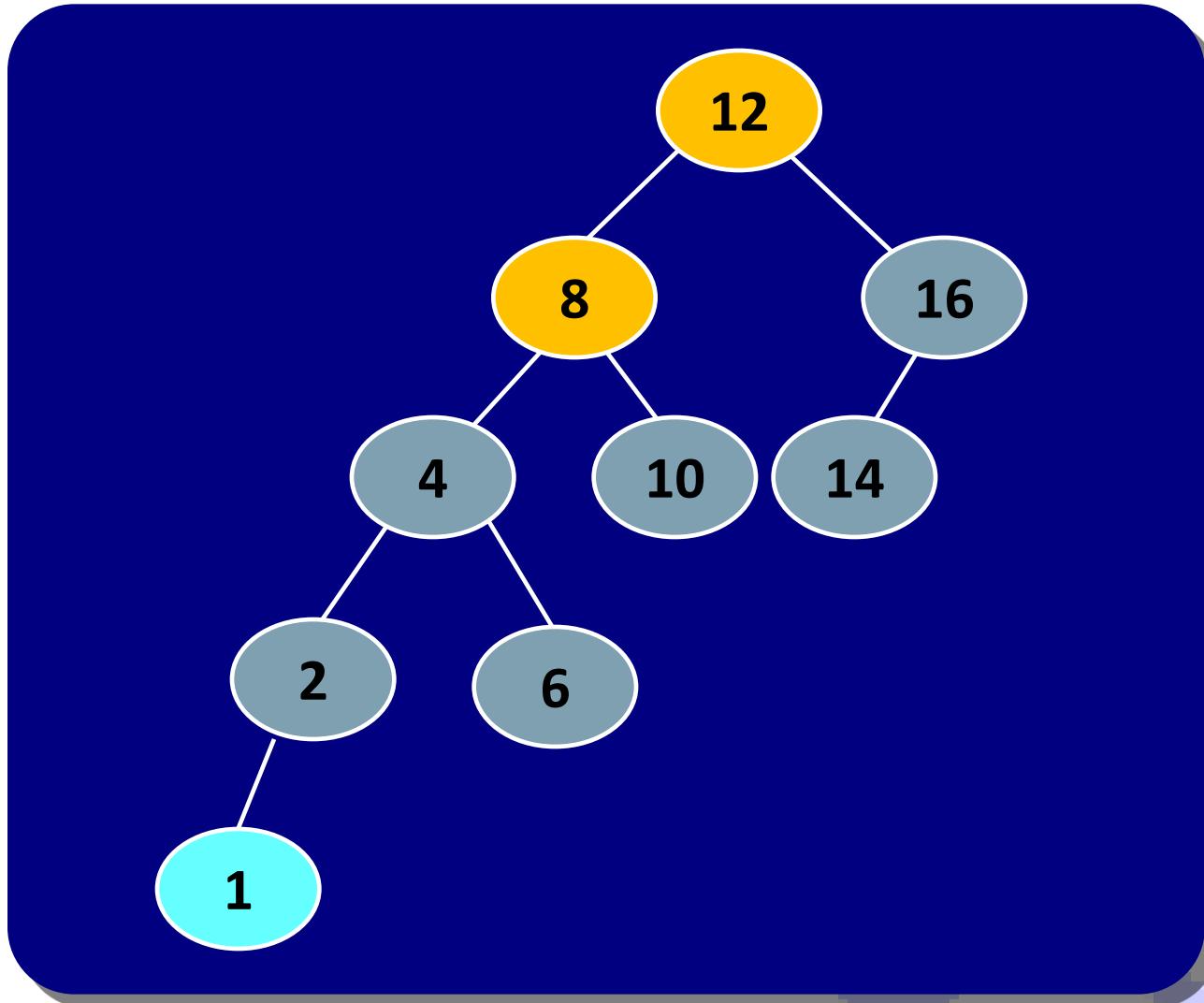


# AVL Trees



# Insertion pada AVL Tree

- Setelah *insert* 1



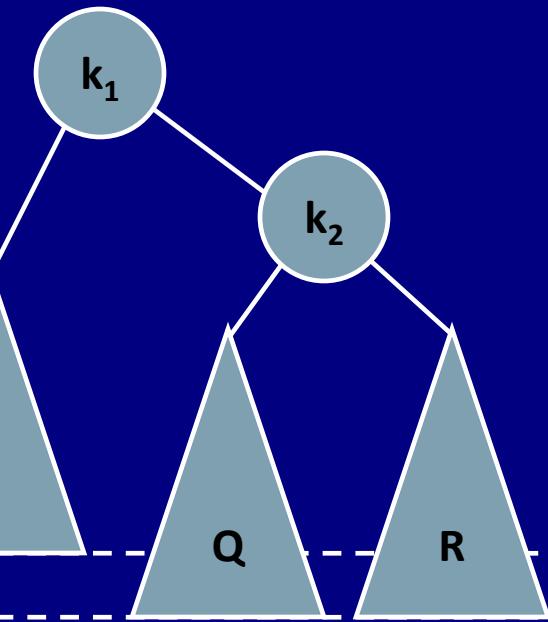
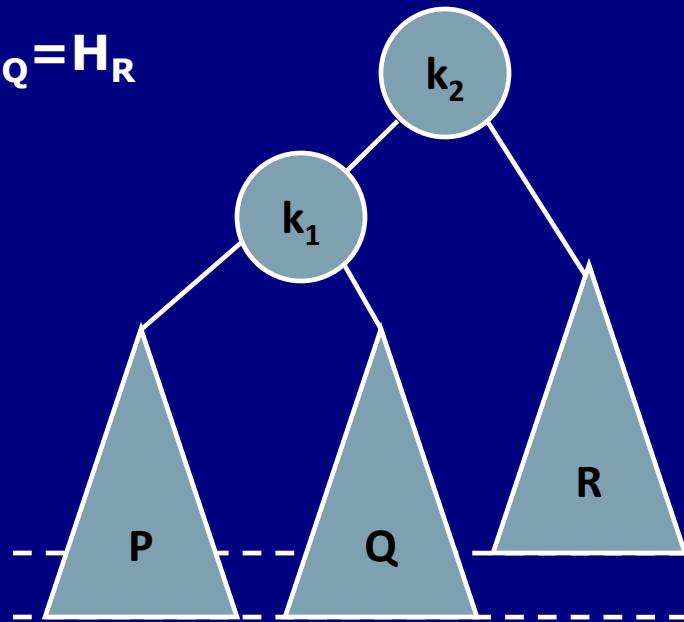
# Insertion pada AVL Tree

- 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*

$$H_P = H_Q = H_R$$



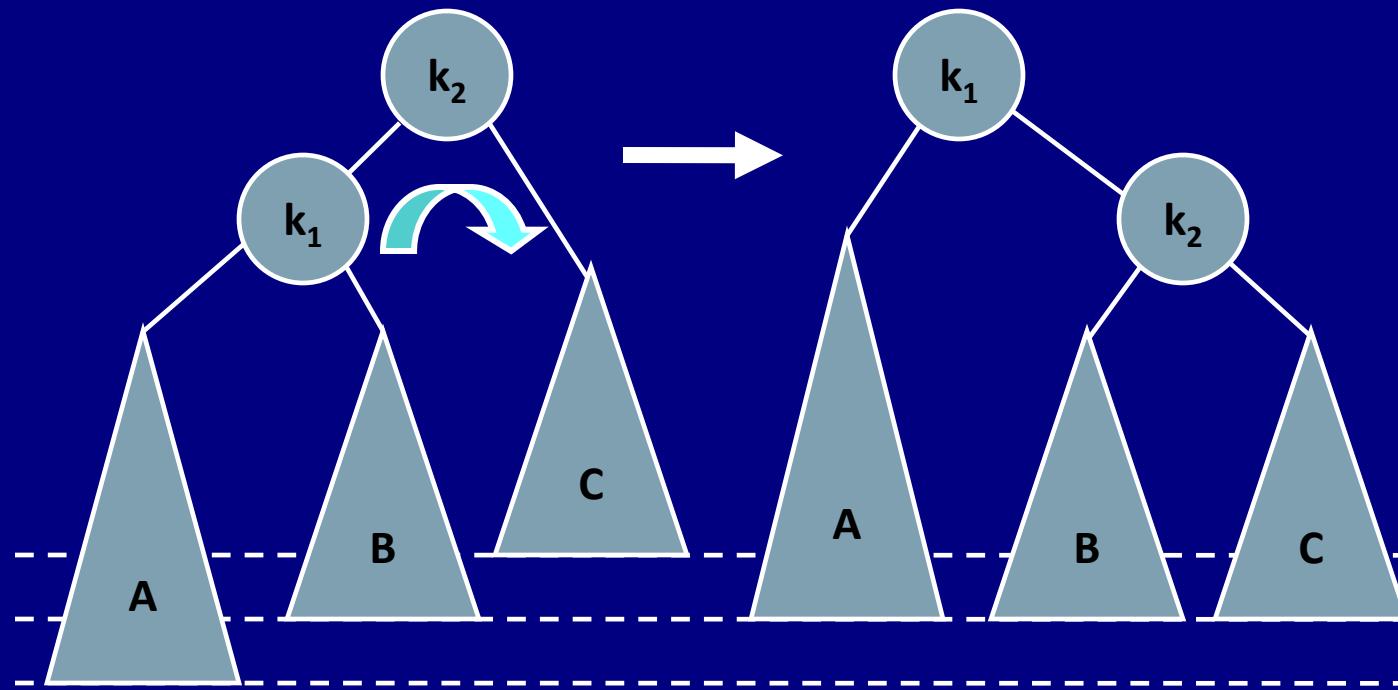
- Sebuah penambahan pada subtree:
  - P (outside) - case 1
  - Q (inside) - case 2

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



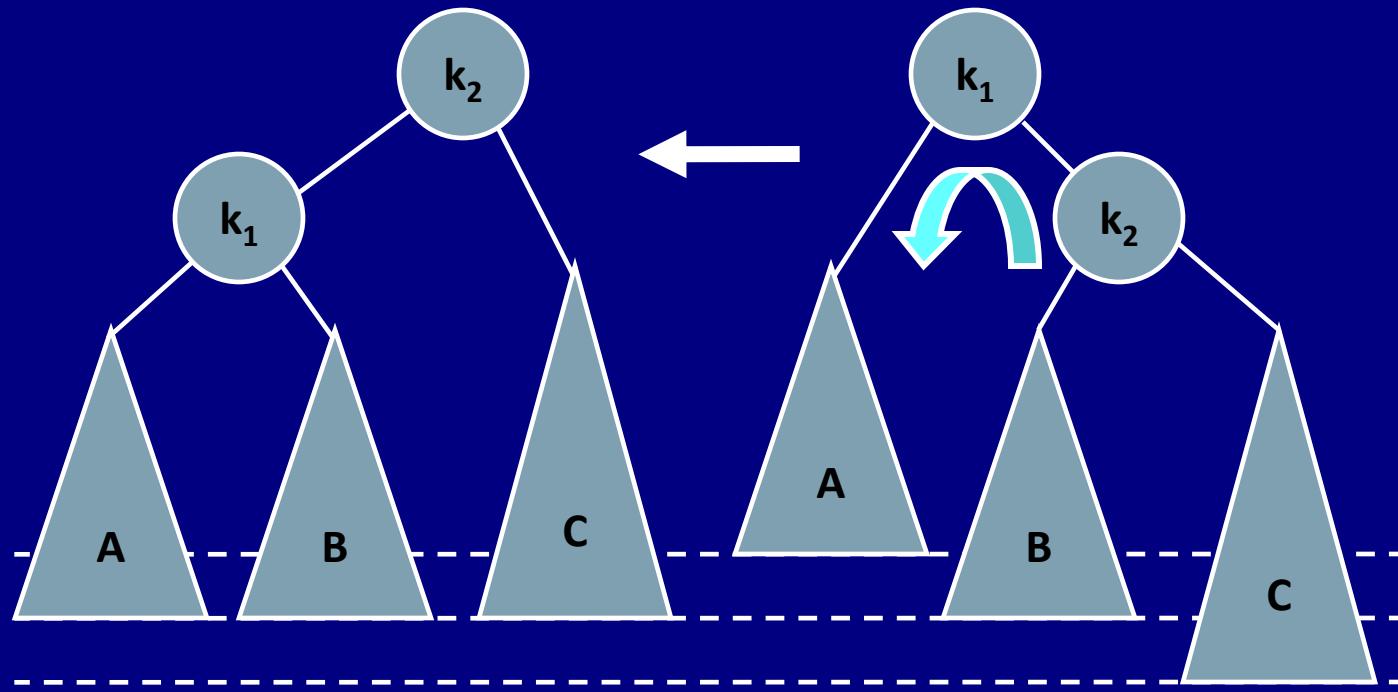
# Single Rotation (case 1)

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



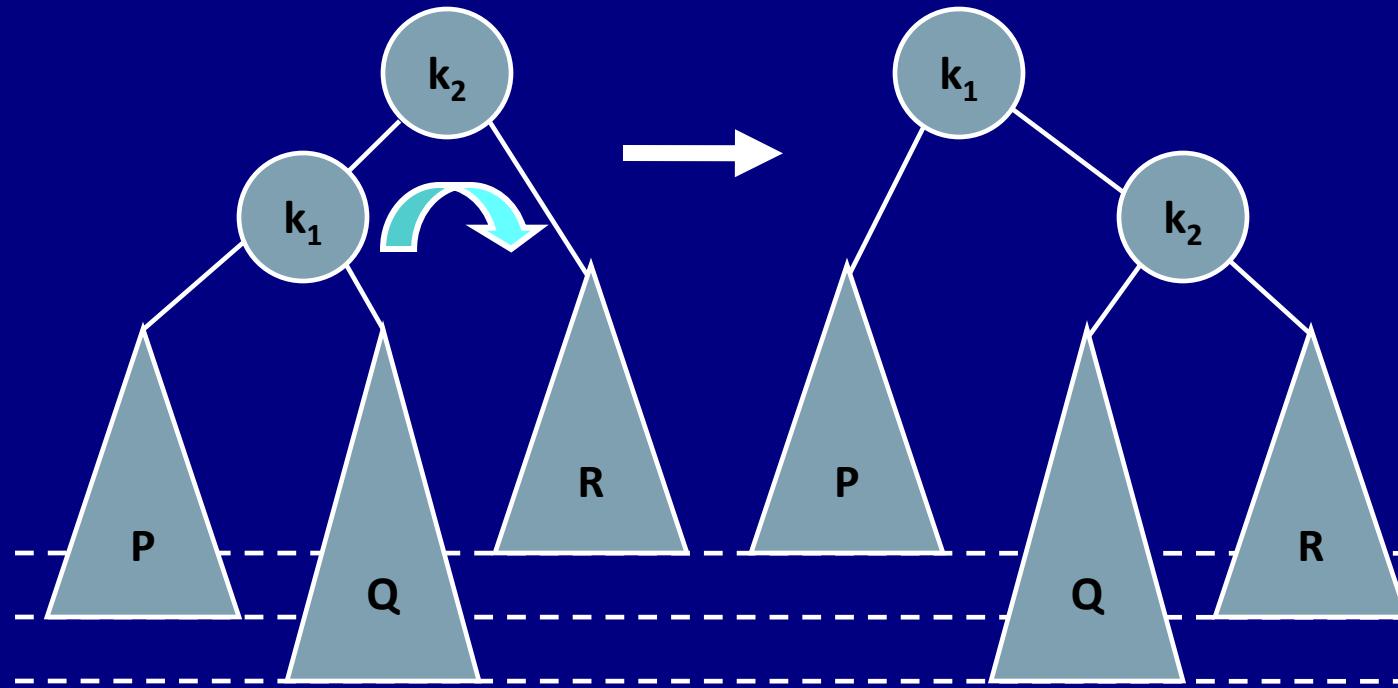
# Single Rotation (case 4)

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



# Keterbatasan Single Rotation

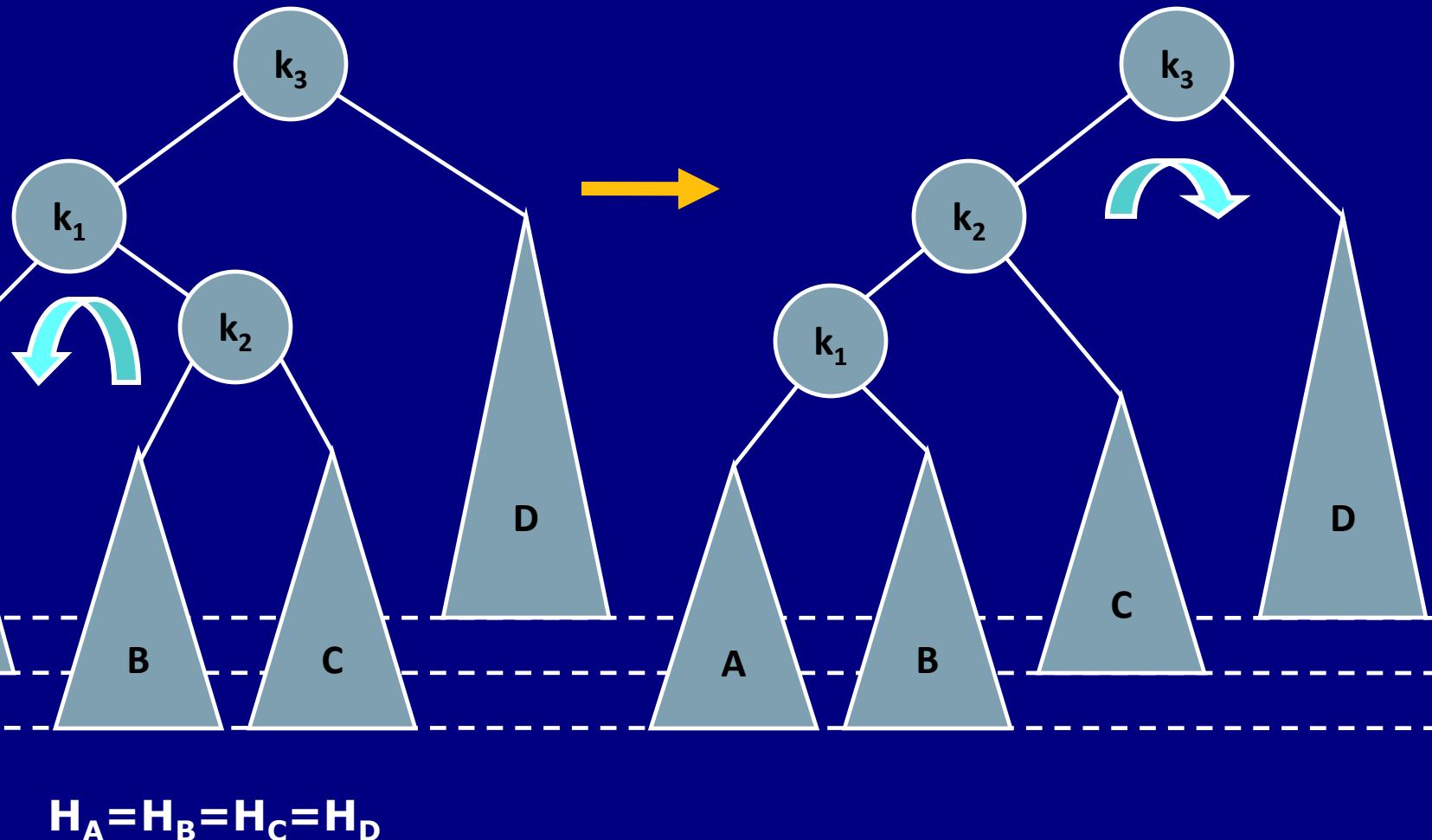
- Single rotation tidak bisa digunakan untuk kasus 2 dan 3 (*inside case*)



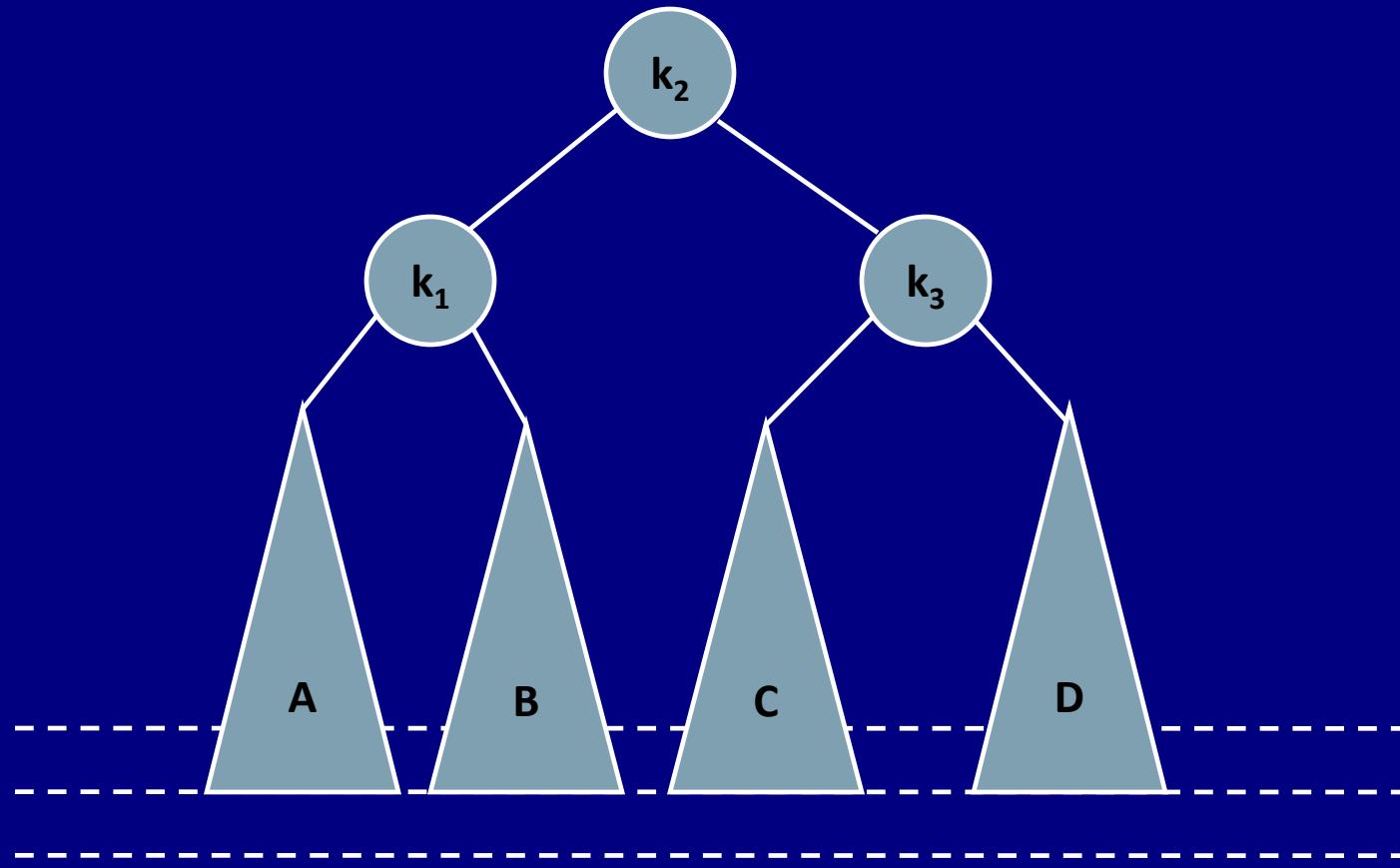
$$H_Q = H_P + 1$$

$$H_P = H_R$$

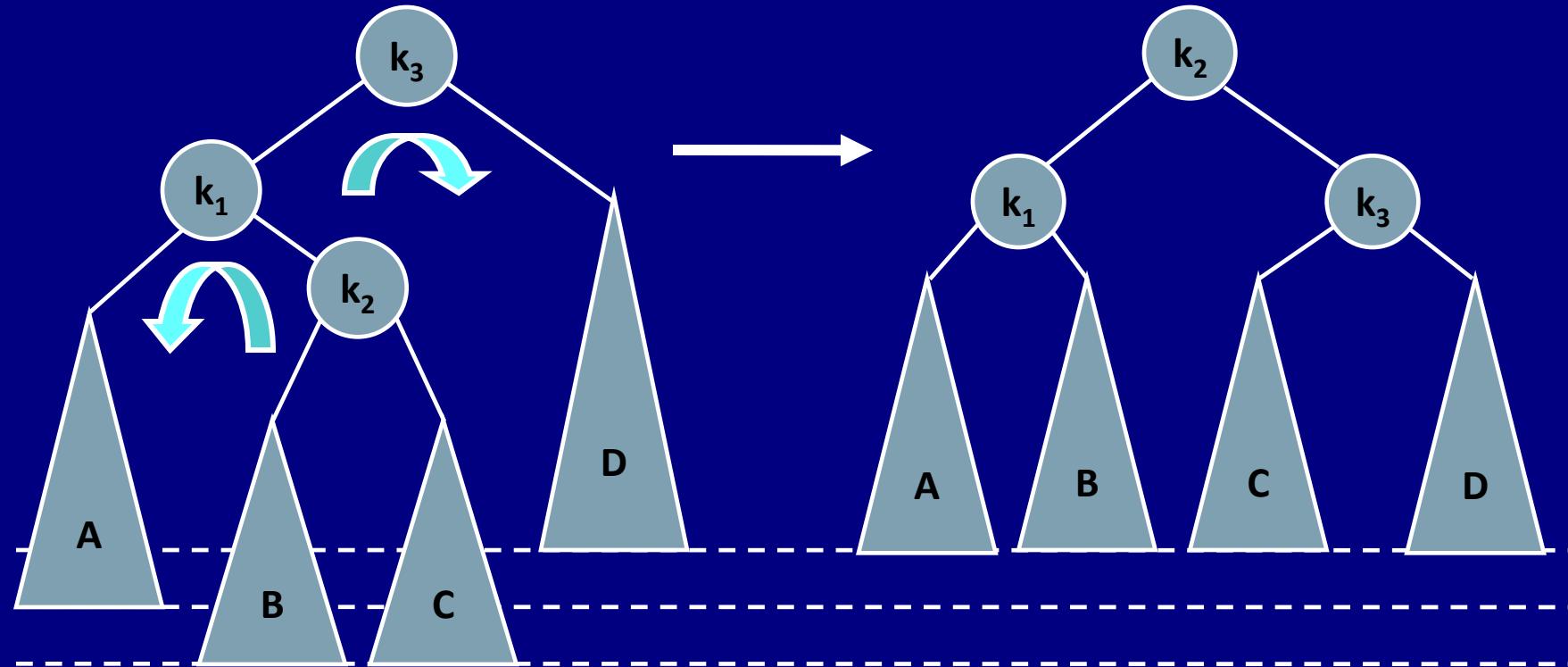
# Double Rotation: Langkah



# Double Rotation: Langkah

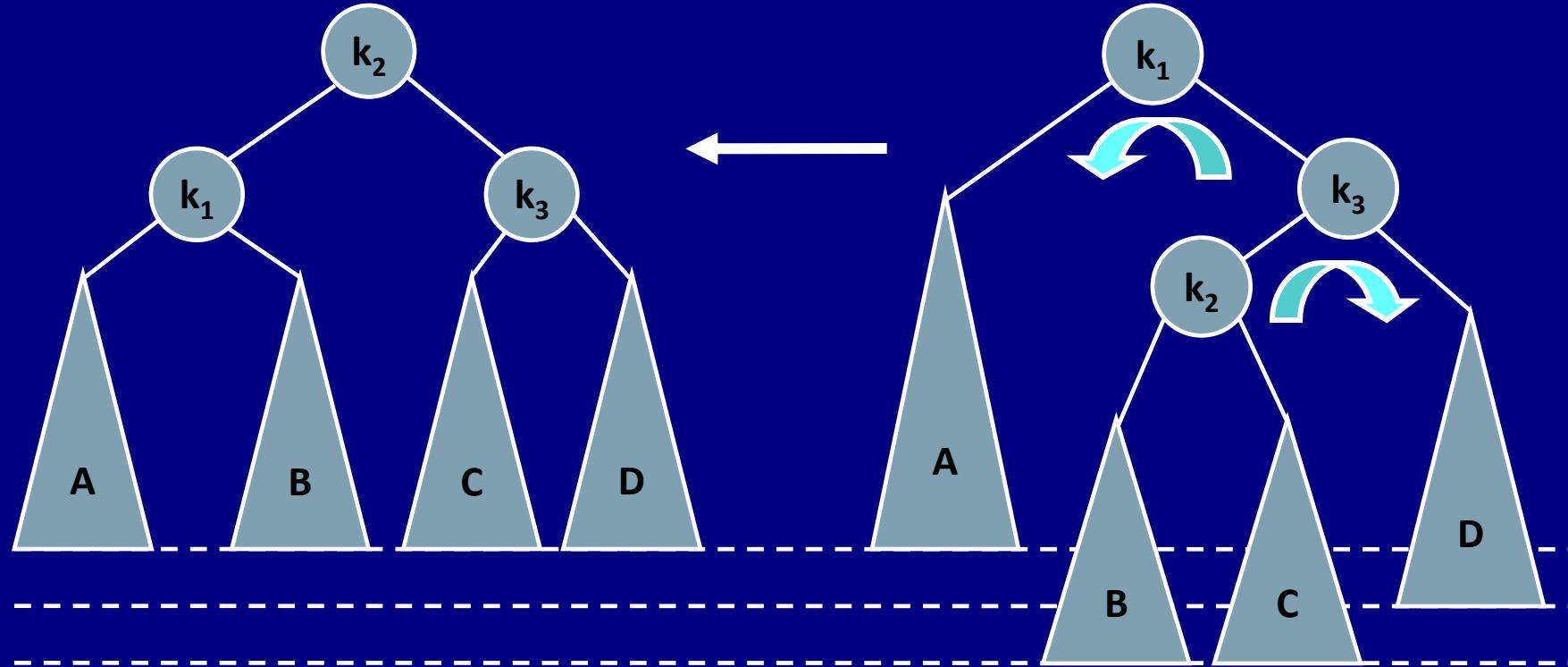


# Double Rotation



$$H_A = H_B = H_C = H_D$$

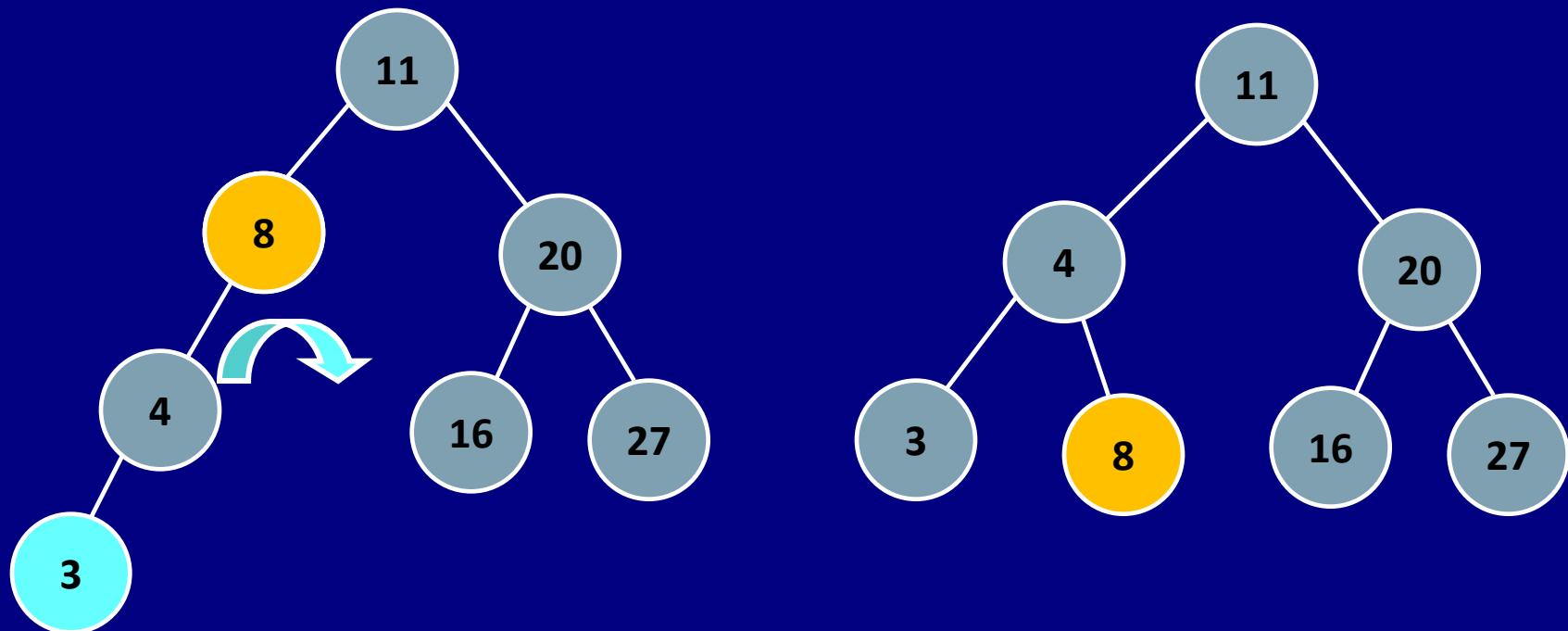
# Double Rotation



$$H_A = H_B = H_C = H_D$$

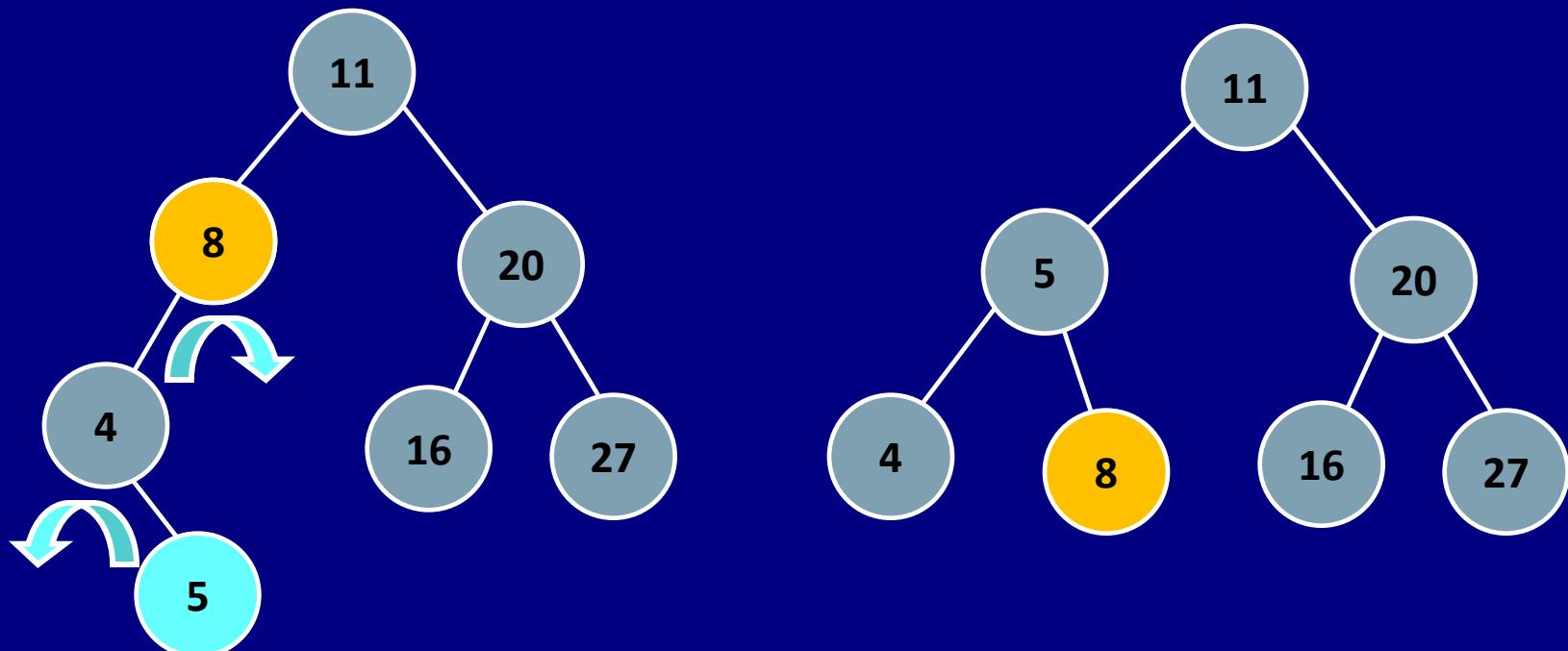
# Contoh

- penambahan 3 pada AVL tree



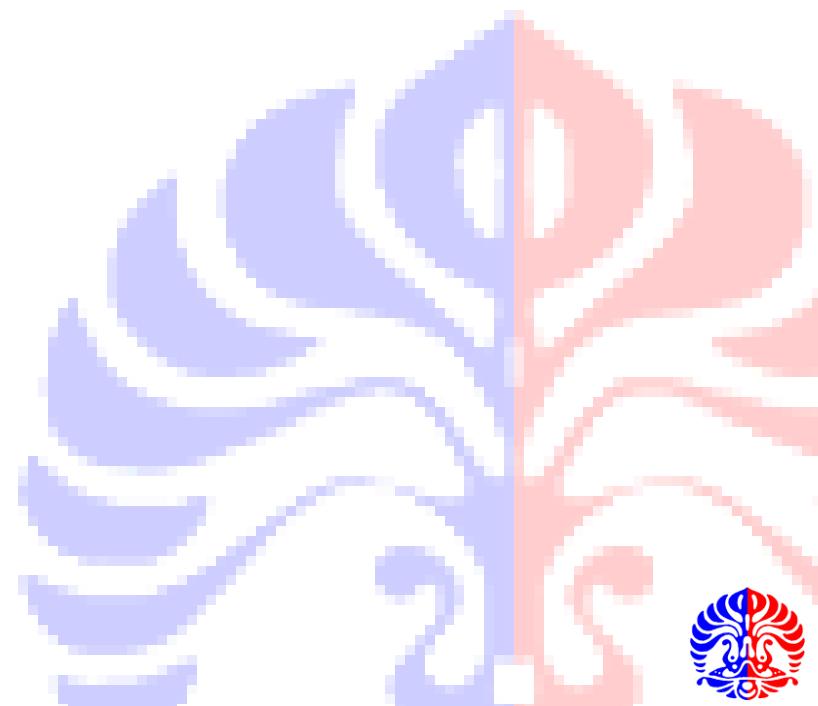
# Contoh

- penambahan 5 pada AVL tree



# AVL Trees: Latihan

- Coba simulasiakan 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

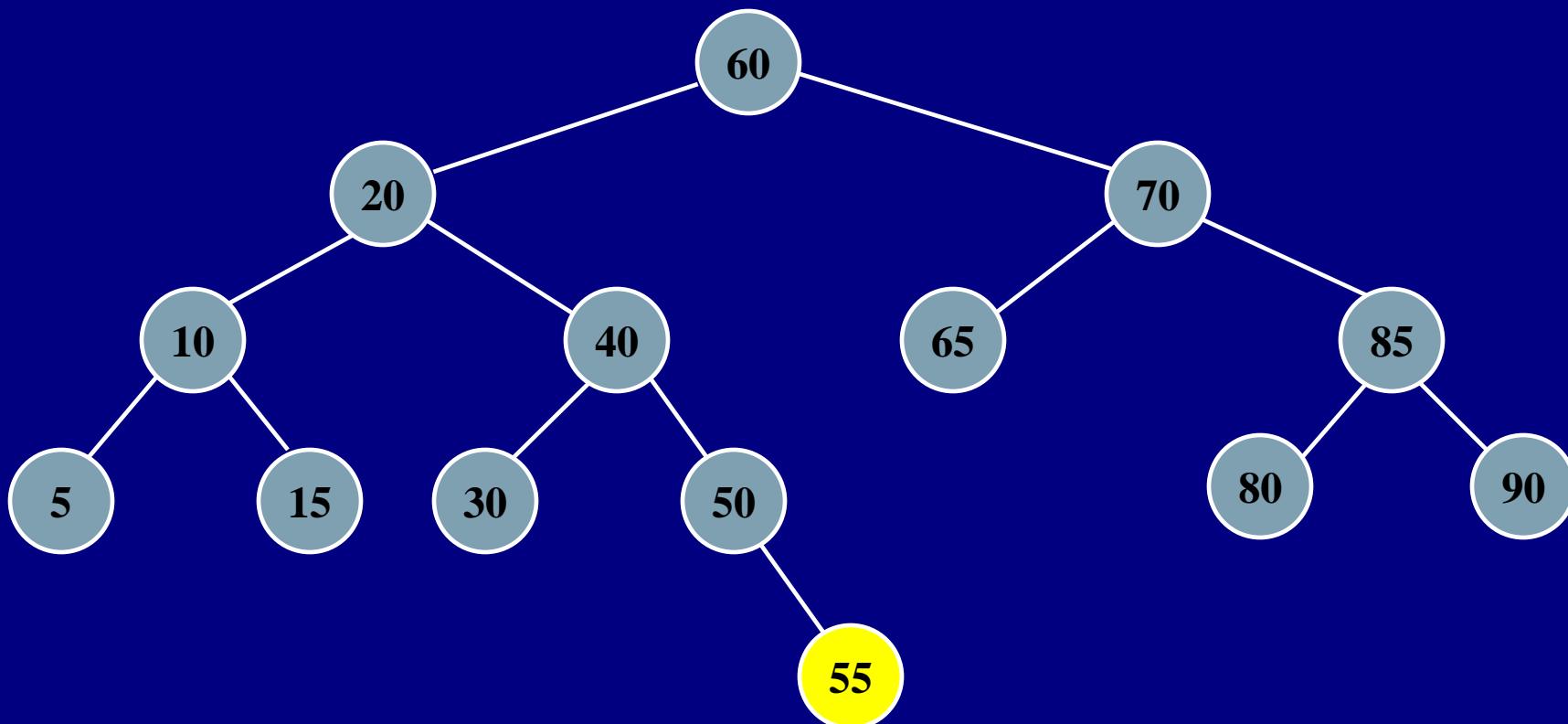
## ■ Deletion:

- 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

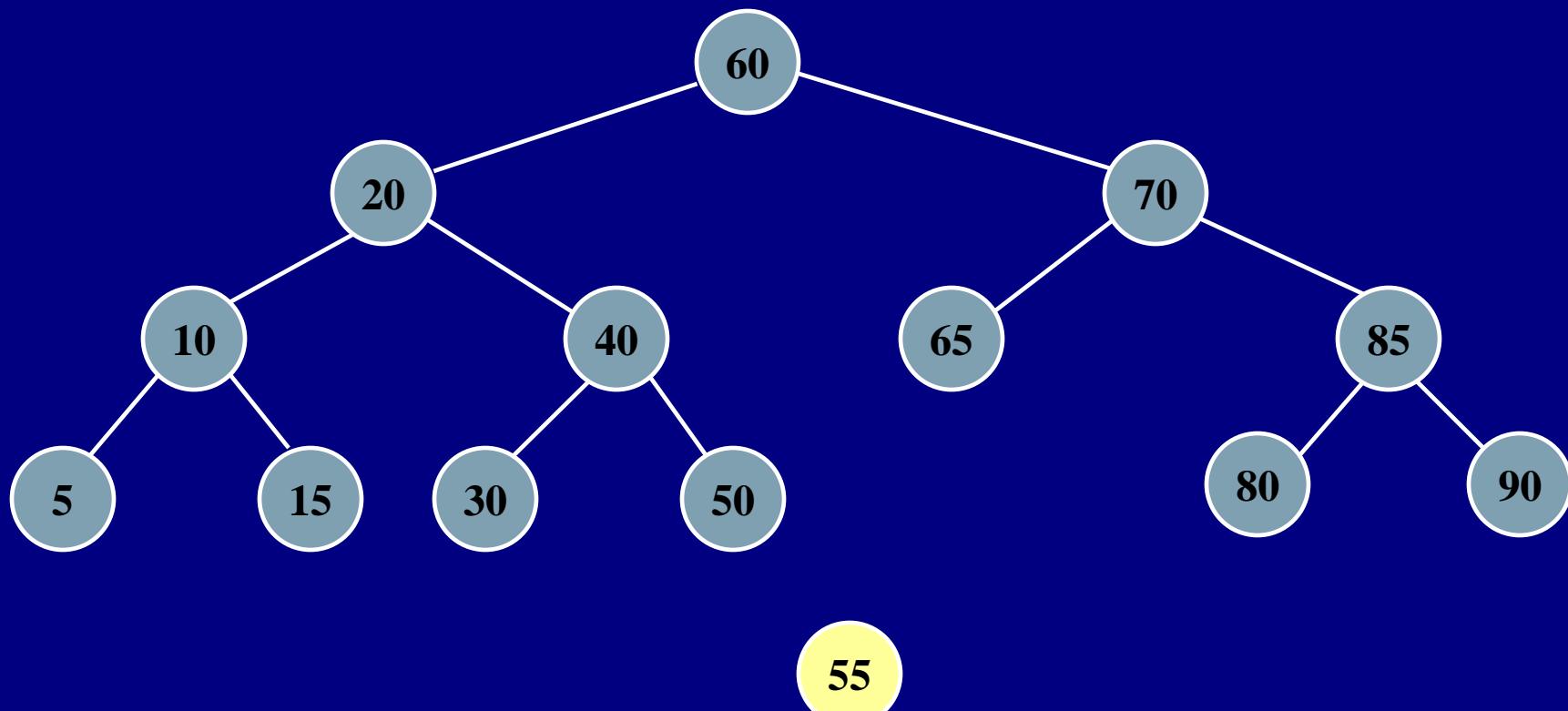
## ■ Rebalancing



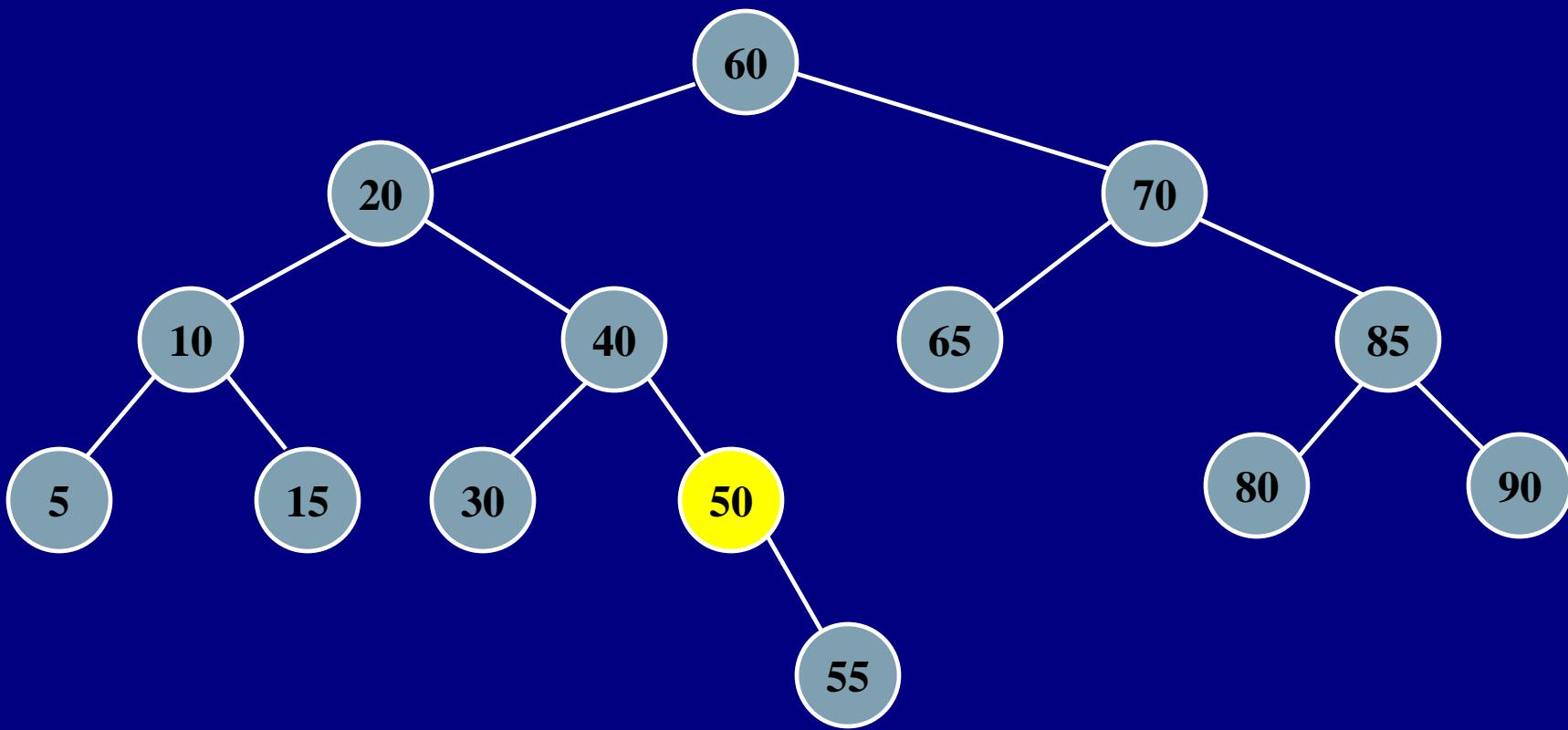
# Delete 55 (Kasus 1)



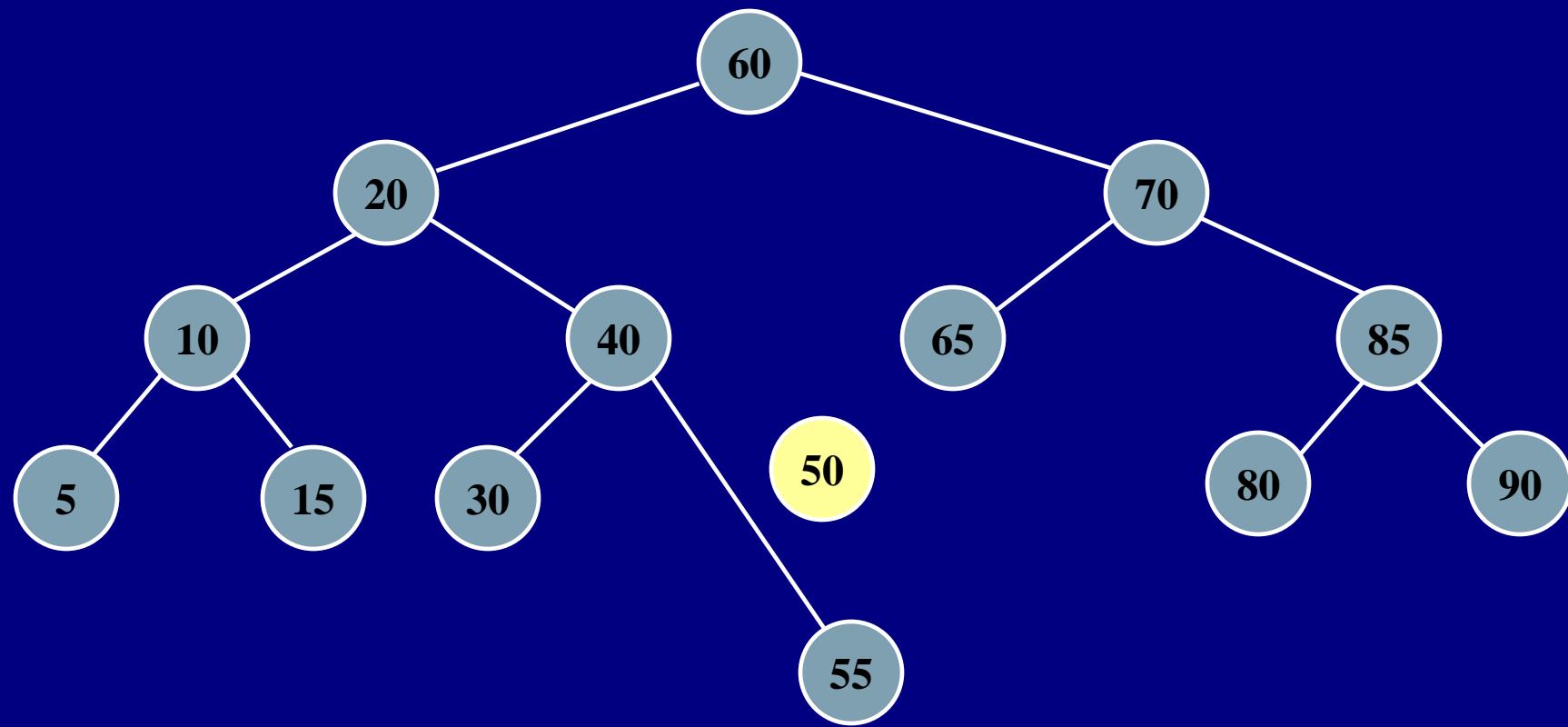
# Delete 55 (Kasus 1)



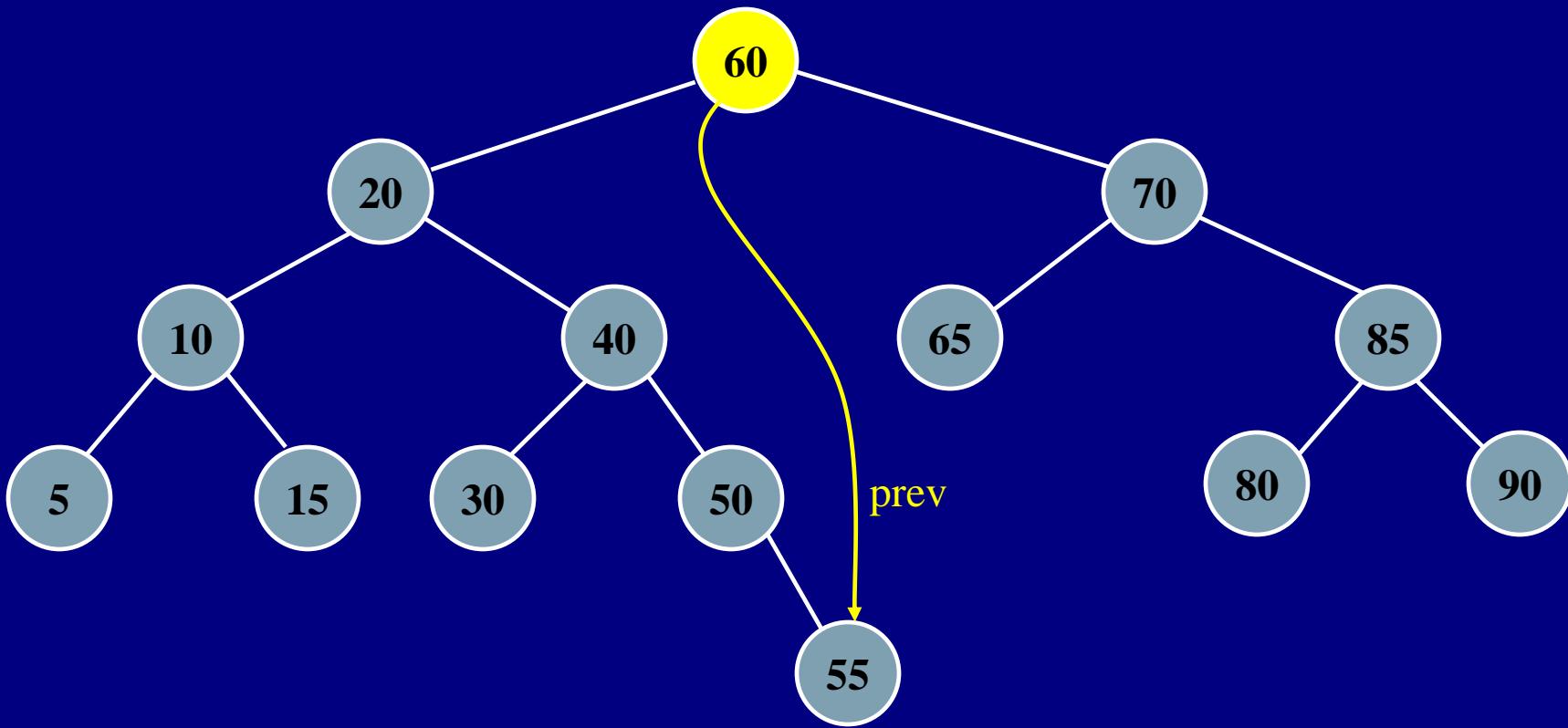
# Delete 50 (Kasus 2)



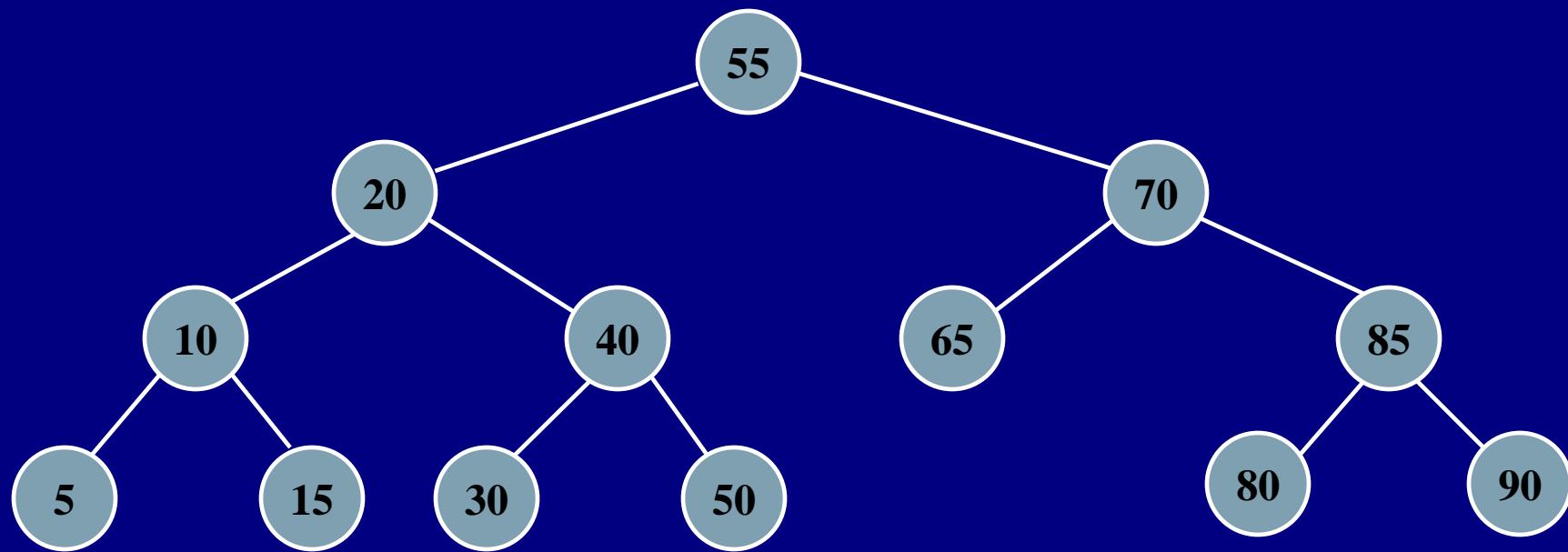
# Delete 50 (Kasus 2)



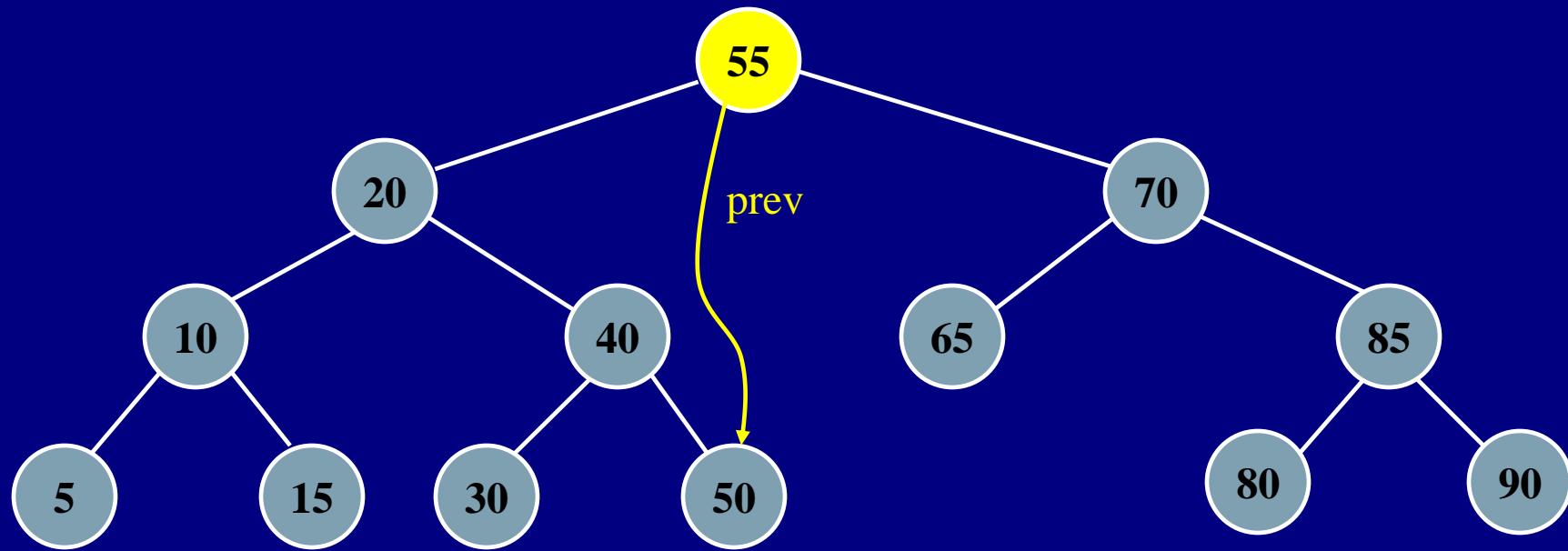
# Delete 60 (Kasus 3)



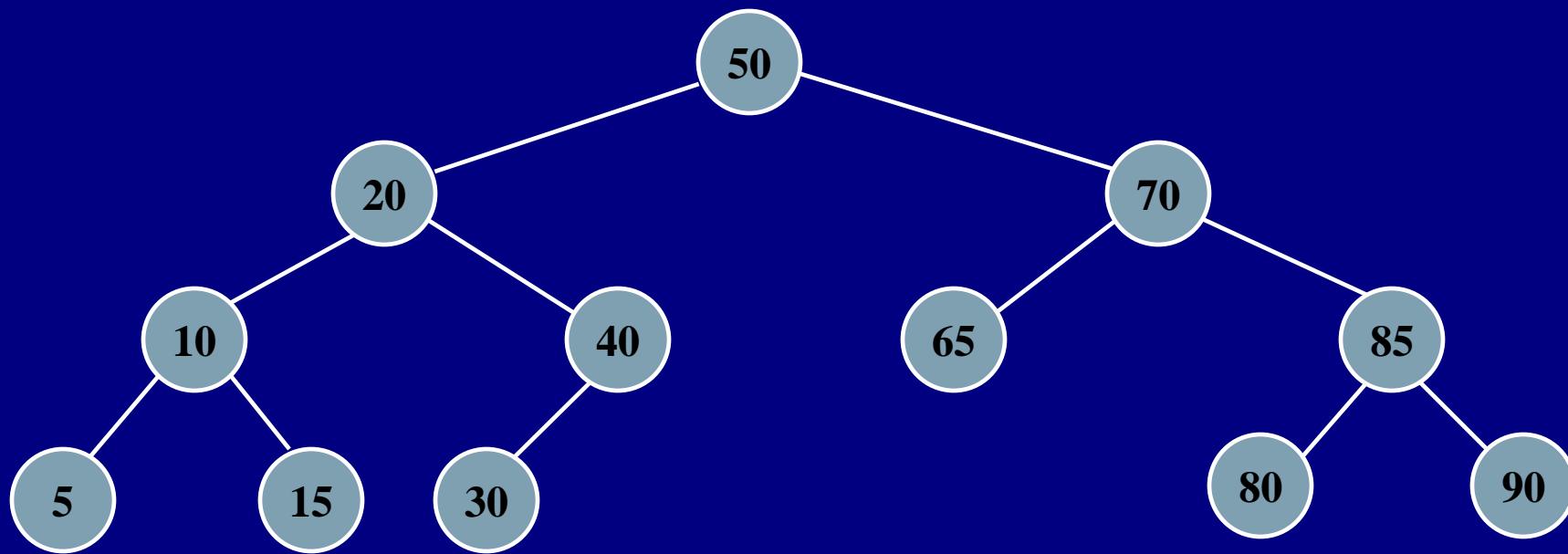
# Delete 60 (Kasus 3)



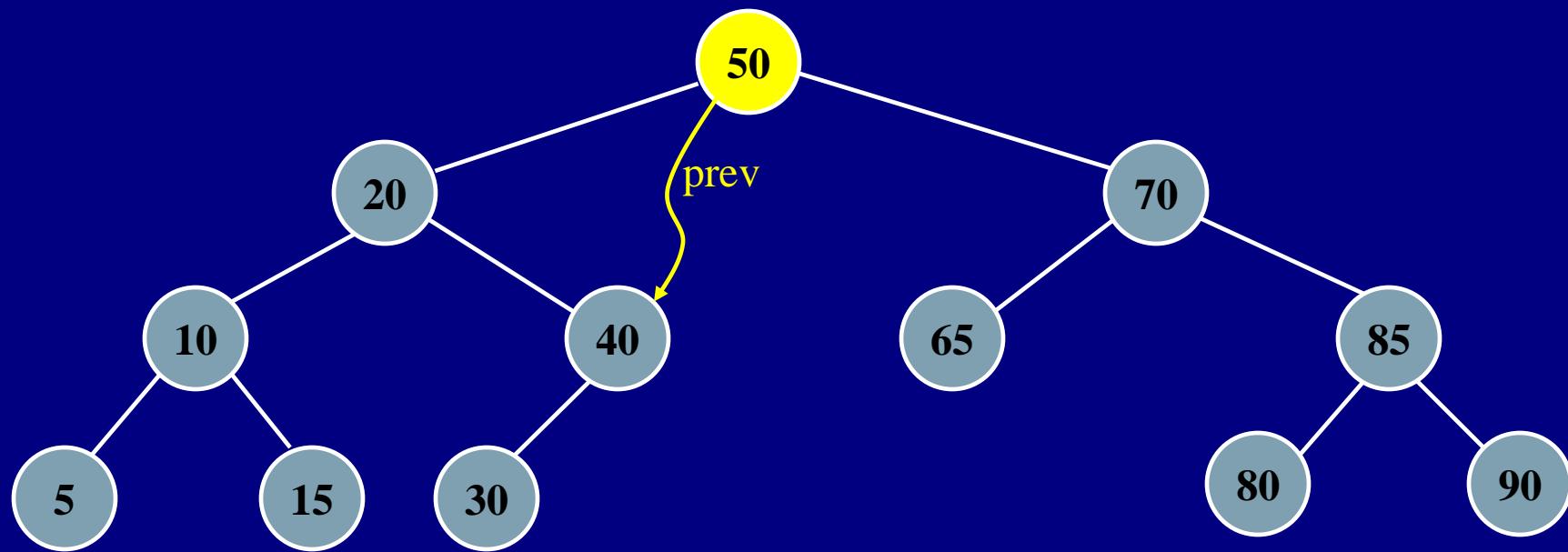
# Delete 55 (Kasus 3)



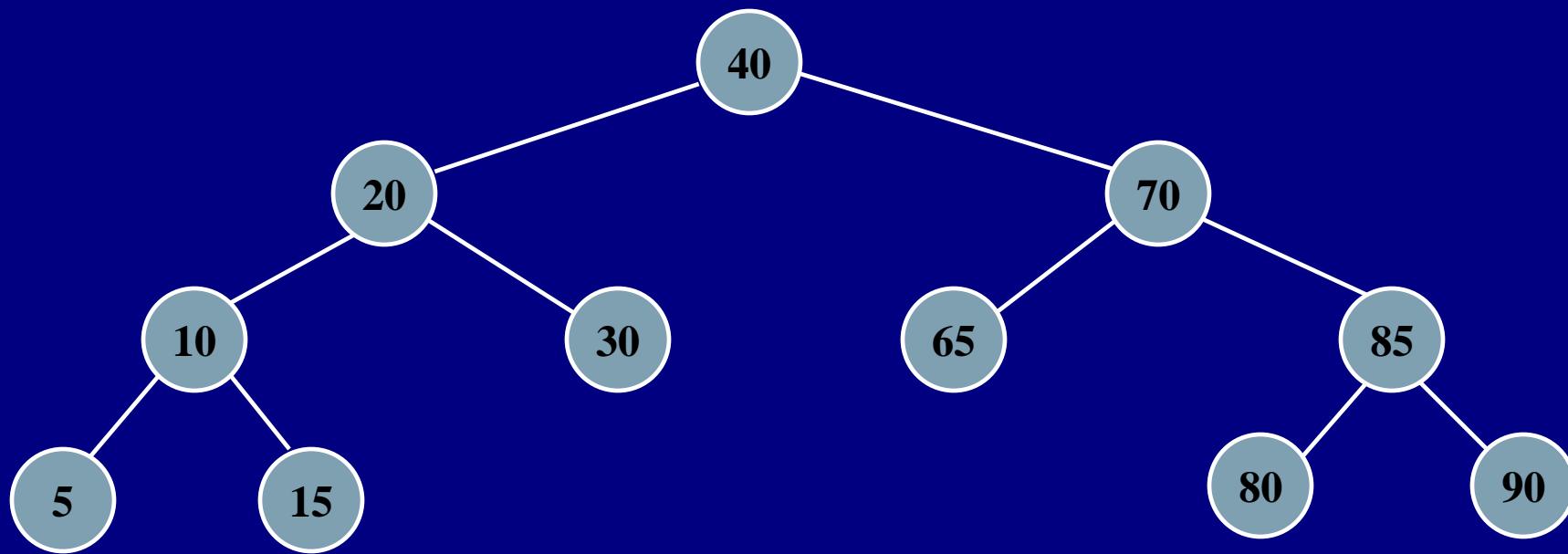
# Delete 55 (Kasus 3)



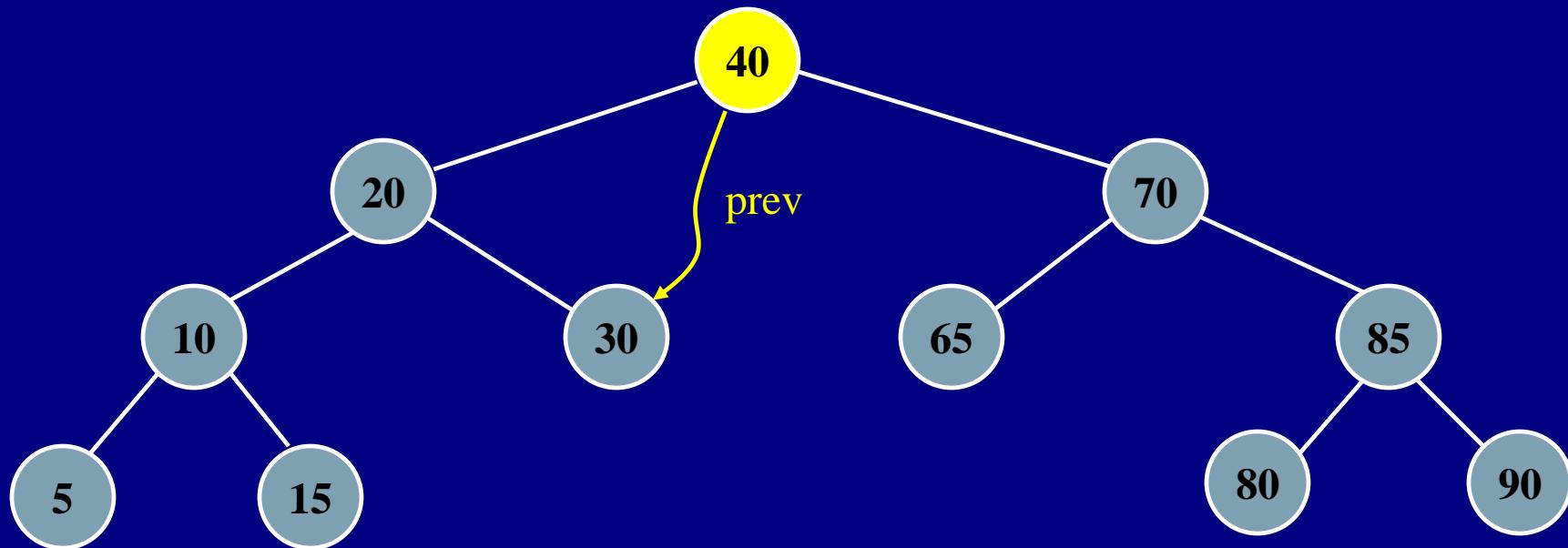
# Delete 50 (Kasus 3)



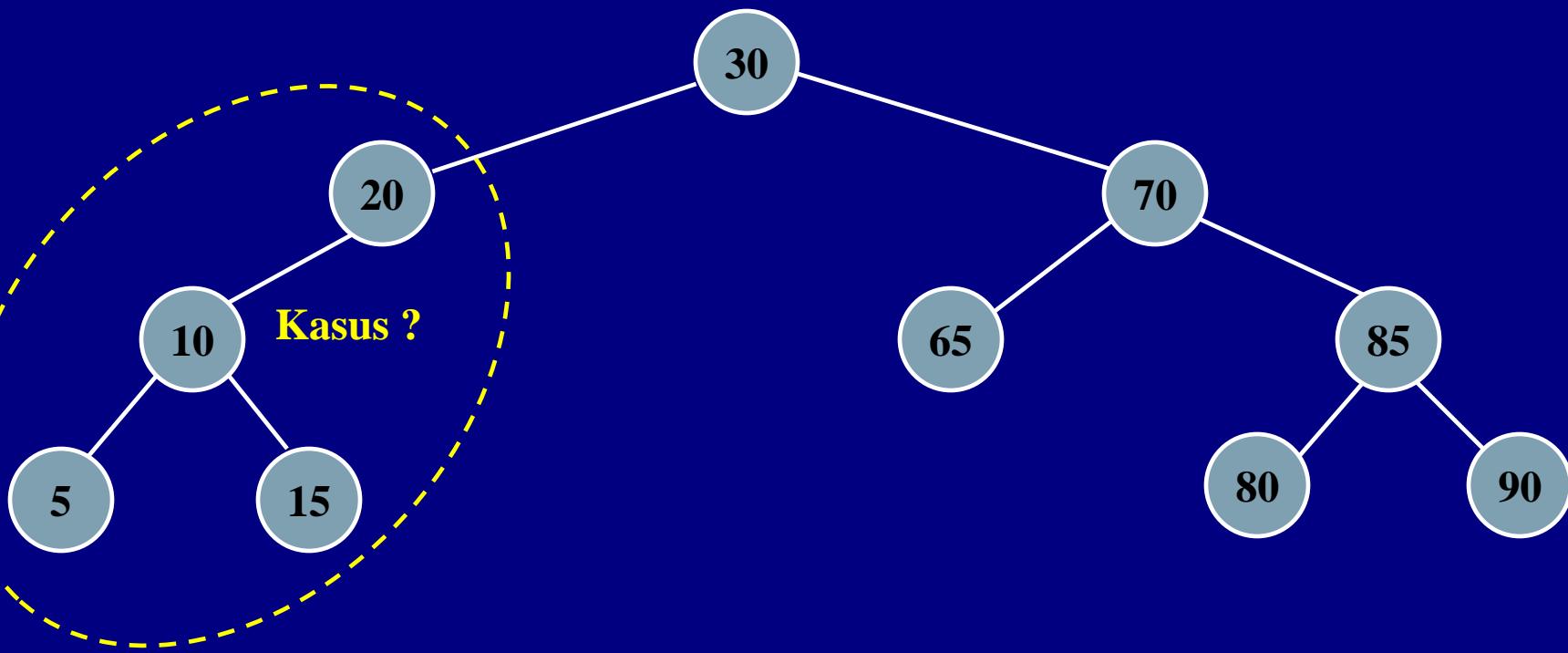
# Delete 50 (Kasus 3)



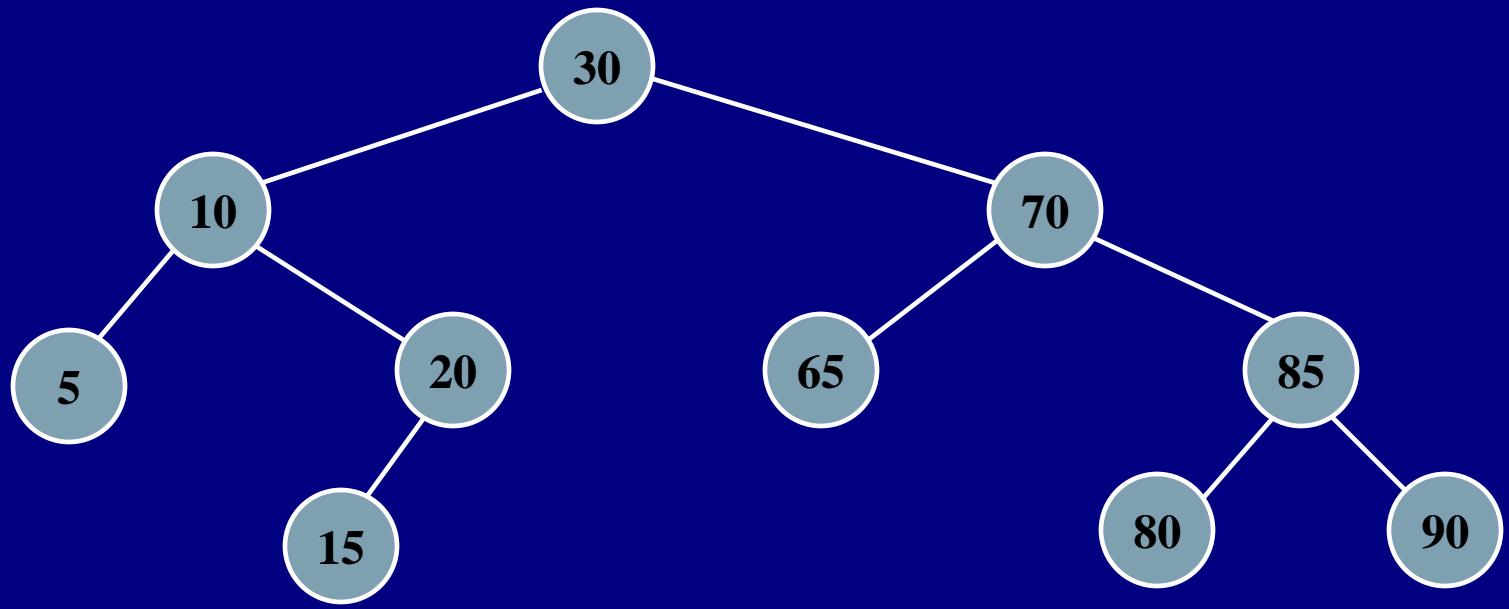
# Delete 40 (Kasus 3)



# Delete 40 : Rebalancing



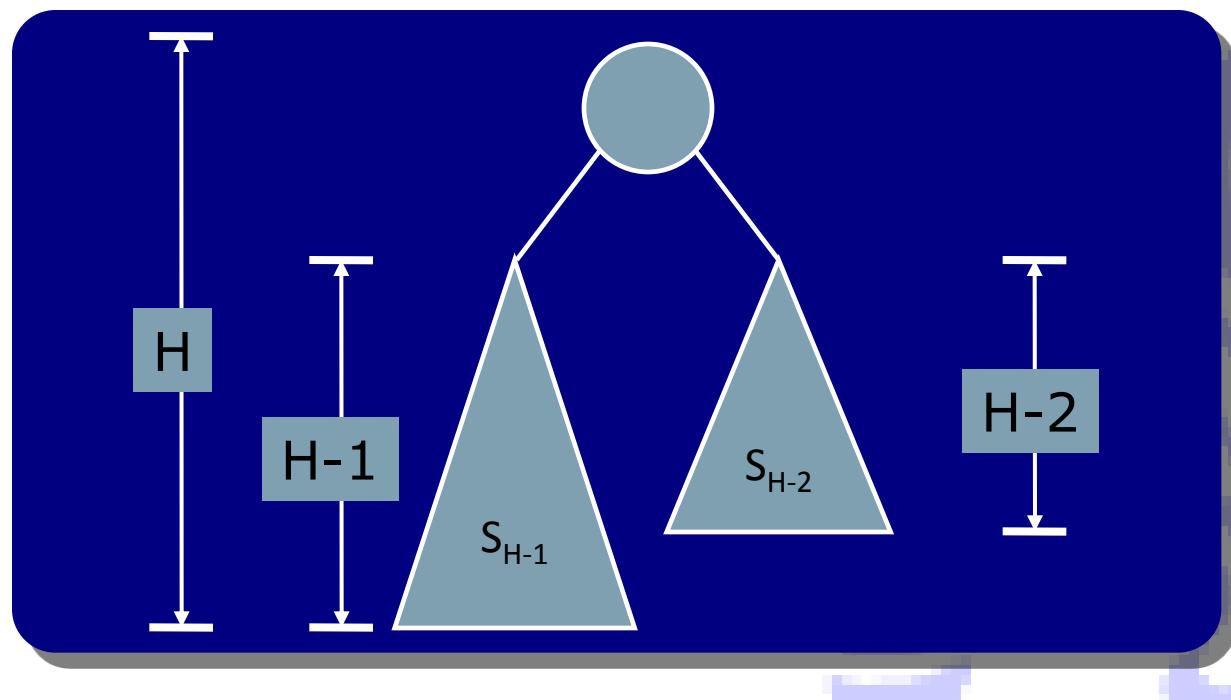
# Delete 40: setelah *rebalancing*



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  $H \rightarrow$  min memiliki  $F_{H+3}-1$  nodes, dimana  $F_i$  elemen ke- $i$  dari deret bilangan fibonacci:

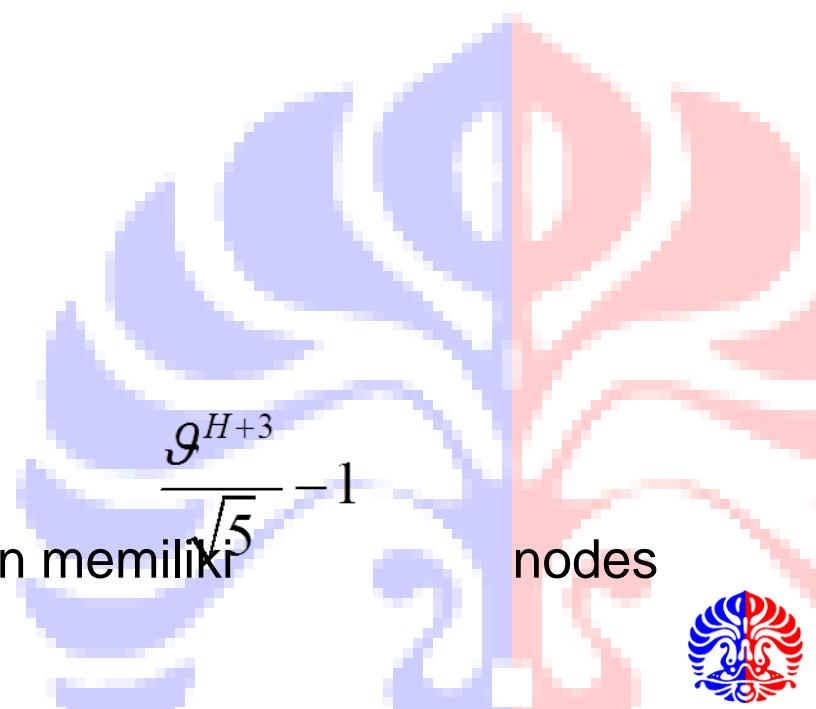
1,1,2,3,5,8,13,21,...

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

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

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

- Sebuah AVL Tree dengan tinggi  $H \rightarrow$  min memiliki



# Tinggi AVL Tree

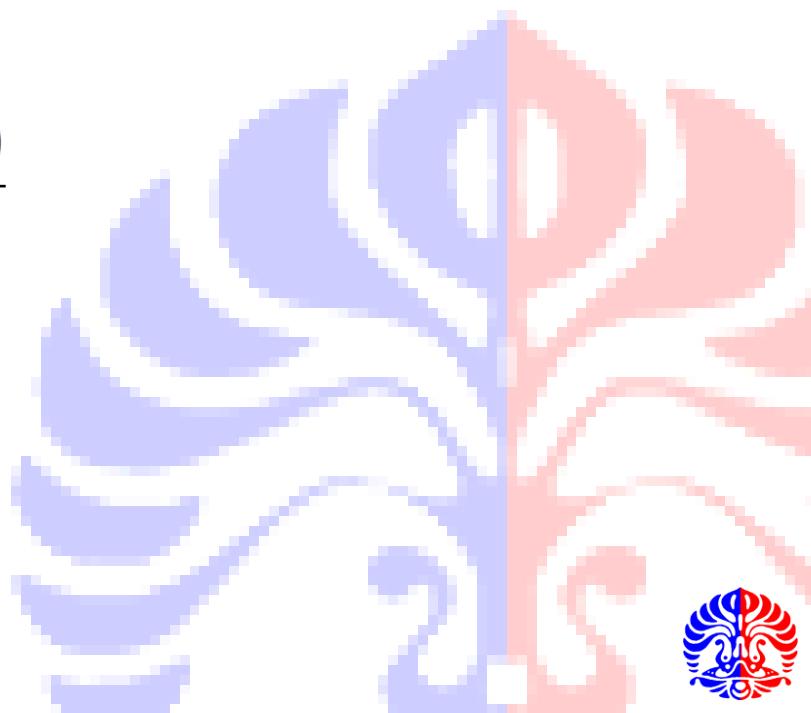
$$F_{H+3} = \frac{g^{H+3}}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{H+3}}{\sqrt{5}}$$

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

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

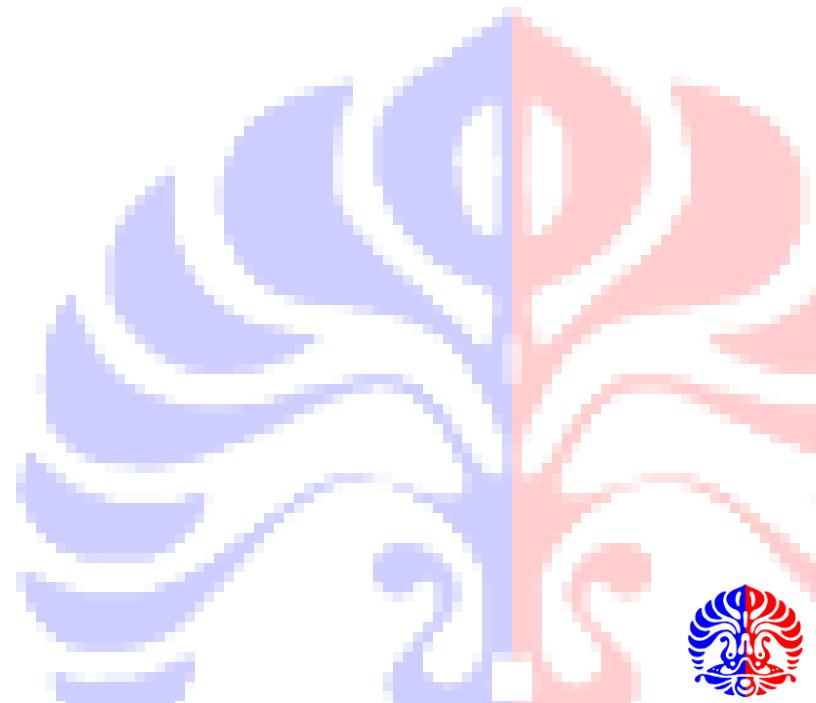
$$(H+3) = \frac{\log F_{H+3} - \frac{\log(5)}{2}}{-\log \left( \frac{2}{1+\sqrt{5}} \right)}$$

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



# 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.



- 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.



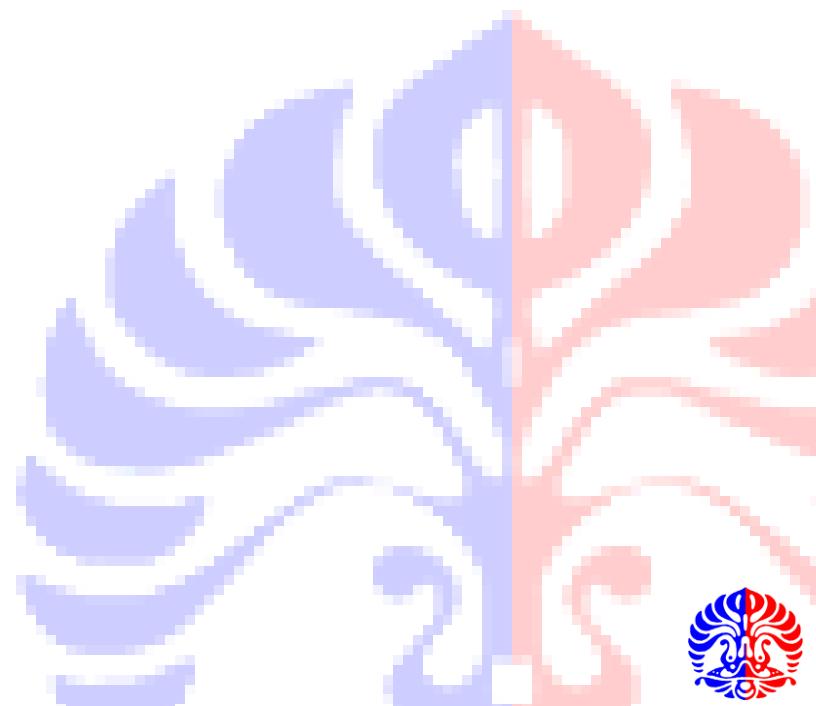
# Algoritma insertion (lanj.)

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

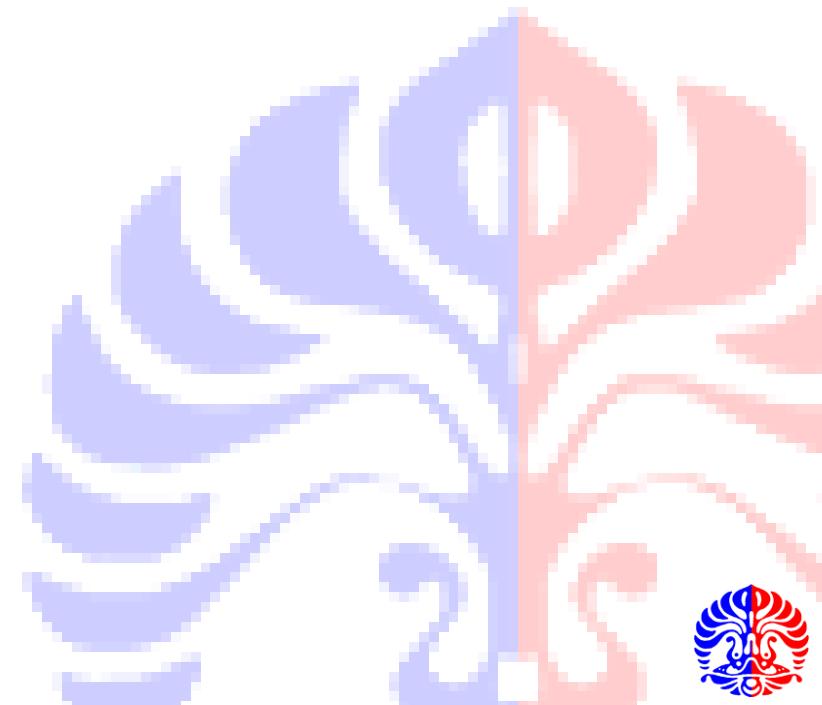
```
public static <A extends Comparable<A>> AvlNode<A>
insert( A x, AvlNode<A> t ){

    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 );
}
```

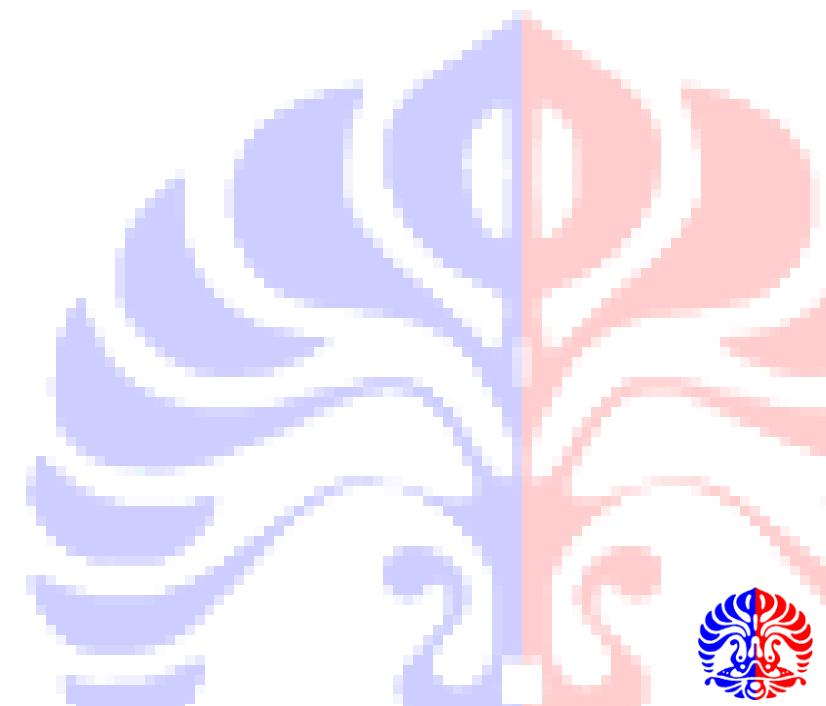
# AVL Trees: Latihan

- Coba simulasi 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



# IKI10400 • Struktur Data & Algoritma: AVL Tree Example Operation

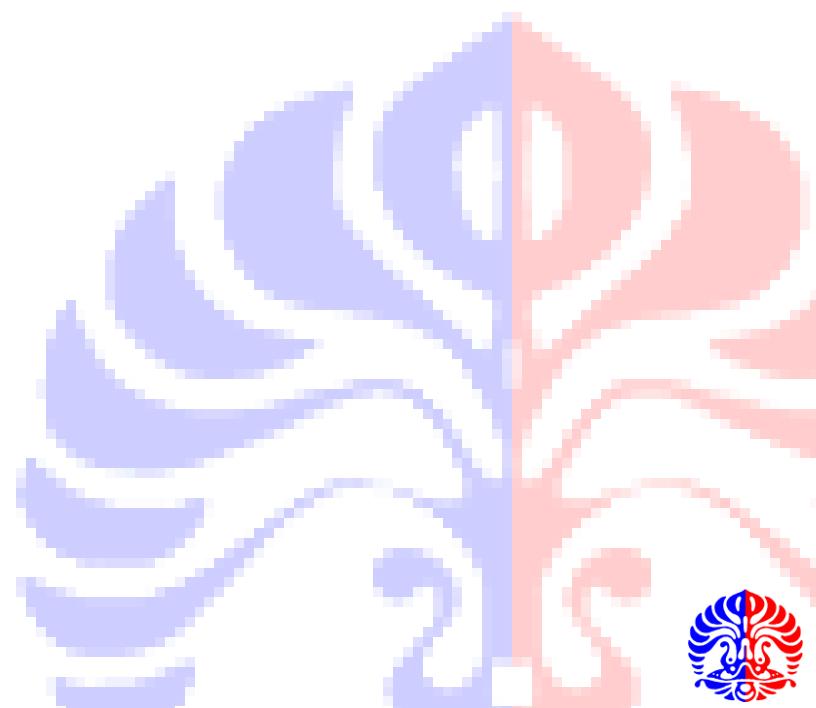
Fakultas Ilmu Komputer • Universitas Indonesia

*Slide acknowledgments:*  
Bayu Distiawan



# Gambarkan step by step hasil operasi berikut:

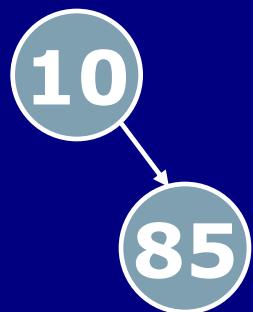
- Insert: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



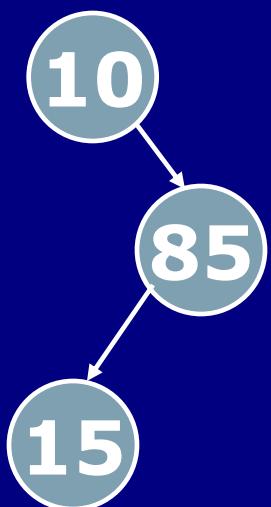
Insert 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55

10

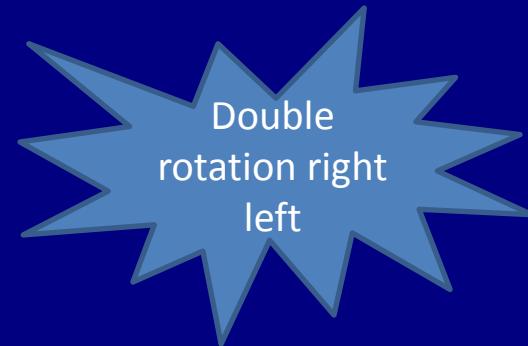
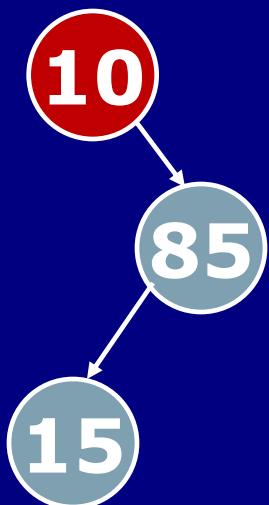
Insert 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



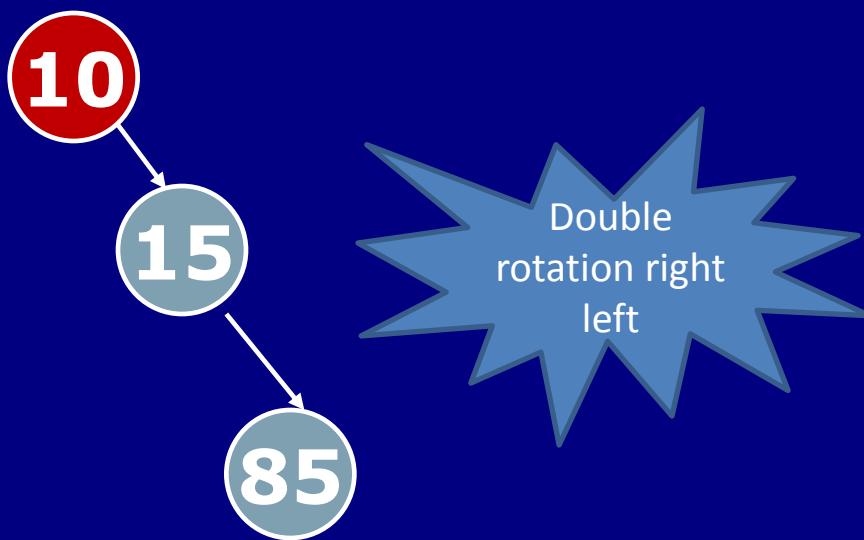
Insert 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



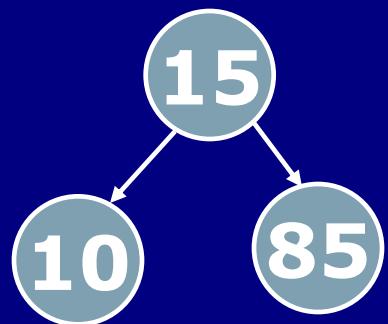
Insert 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



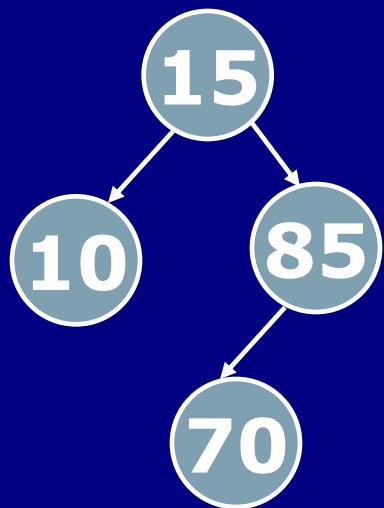
Insert 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



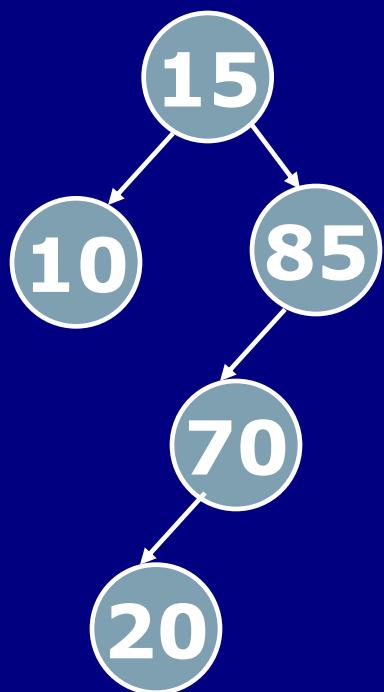
Insert 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



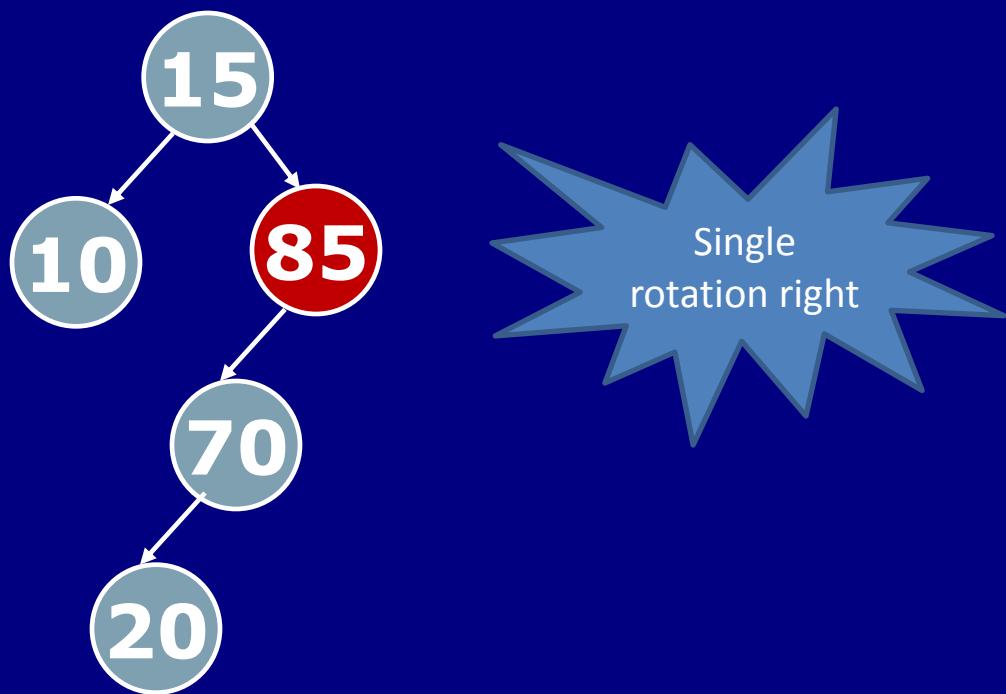
Insert 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



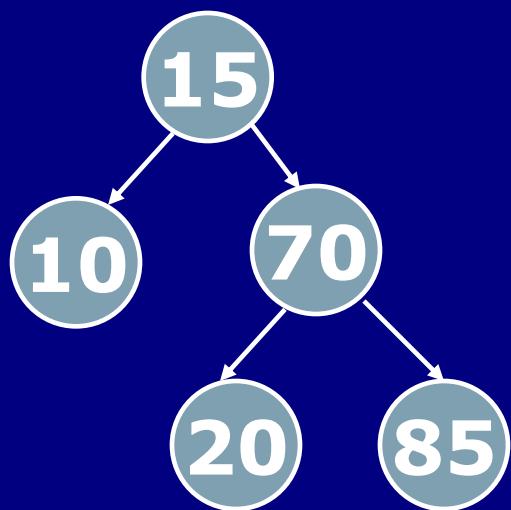
Insert 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



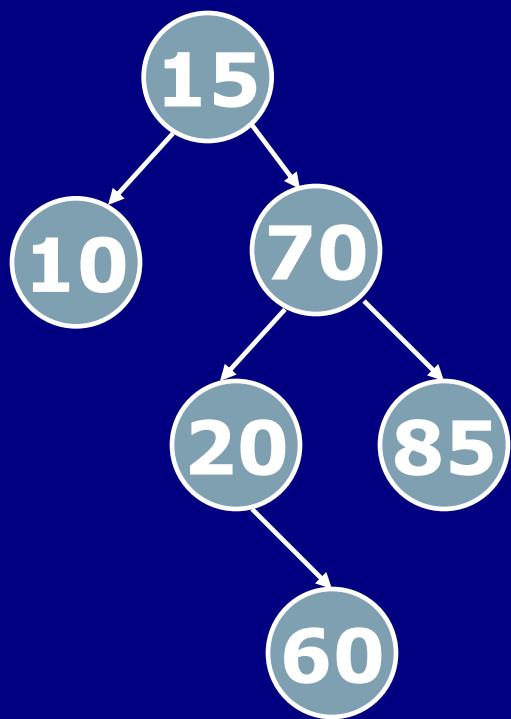
Insert 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



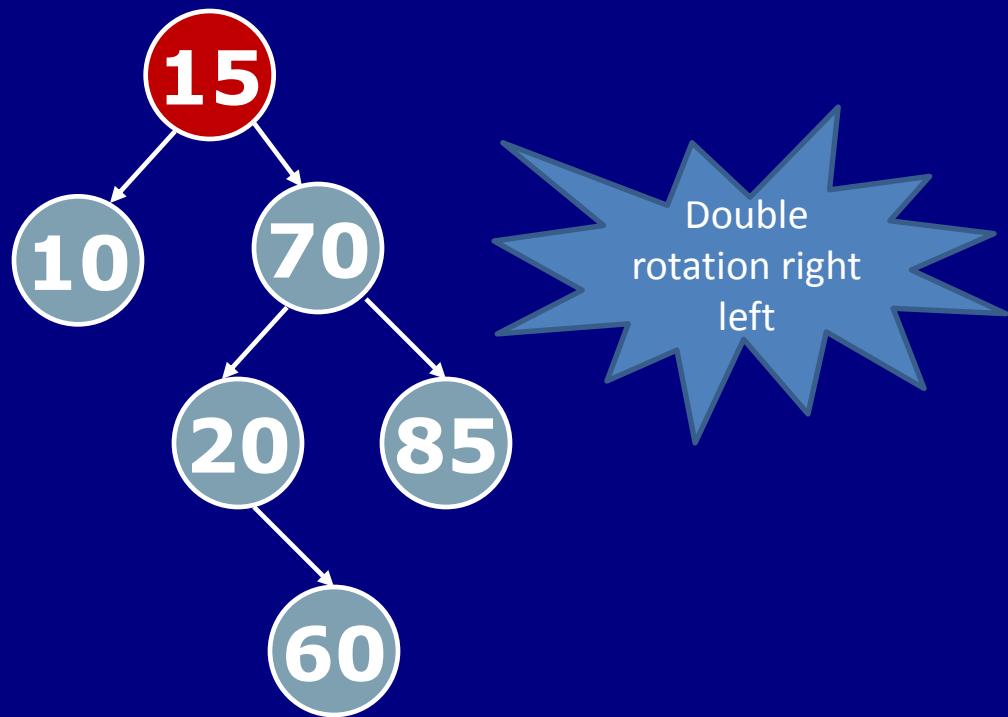
Insert 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



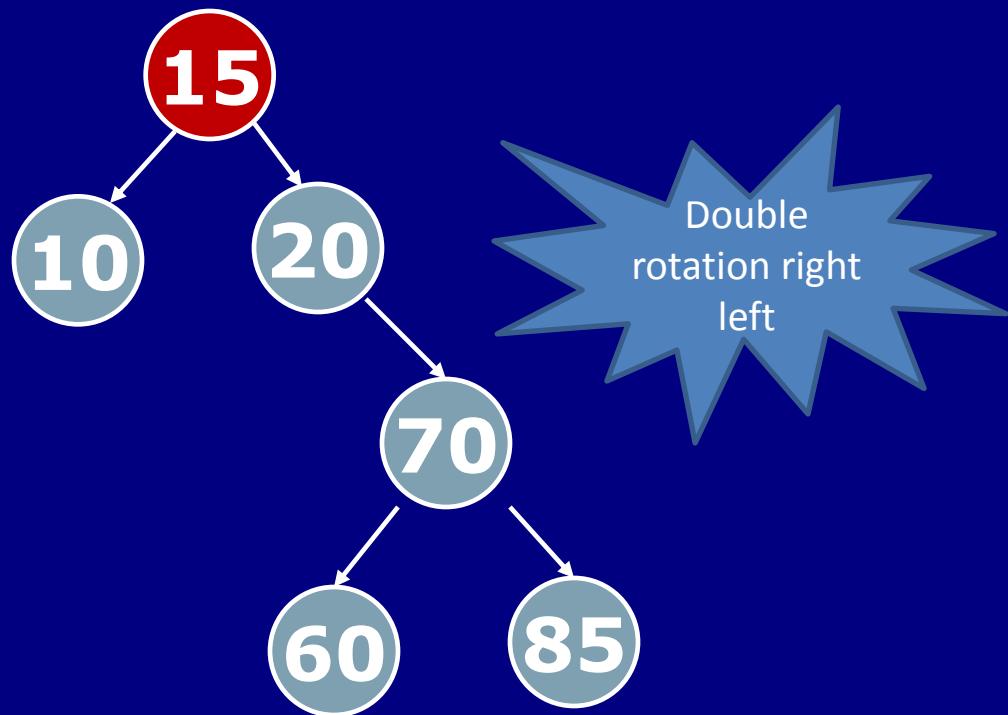
Insert 60, 30, 50, 65, 80, 90, 40, 5, 55



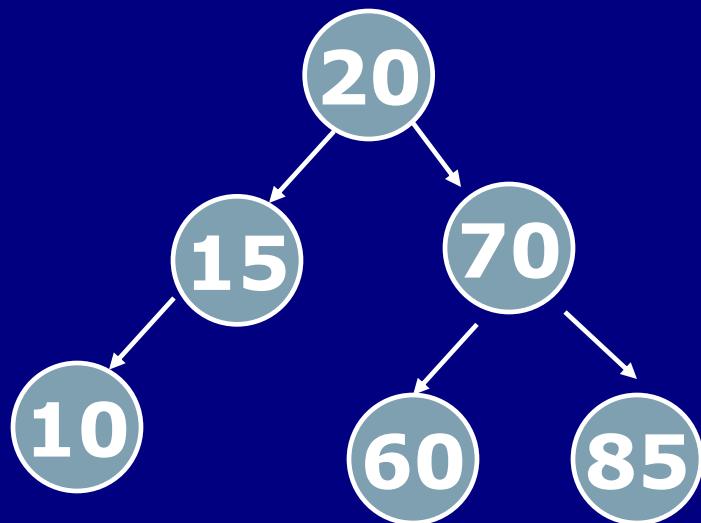
Insert 60, 30, 50, 65, 80, 90, 40, 5, 55



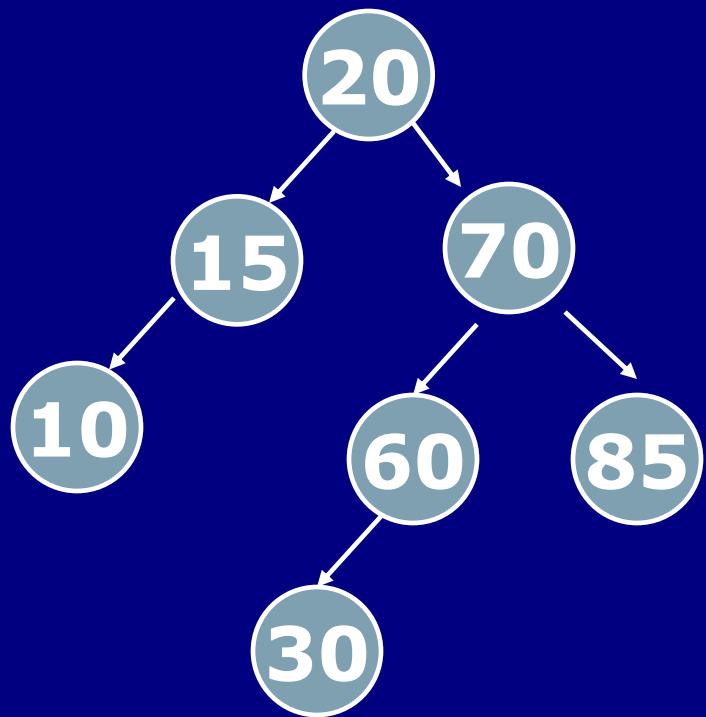
Insert 60, 30, 50, 65, 80, 90, 40, 5, 55



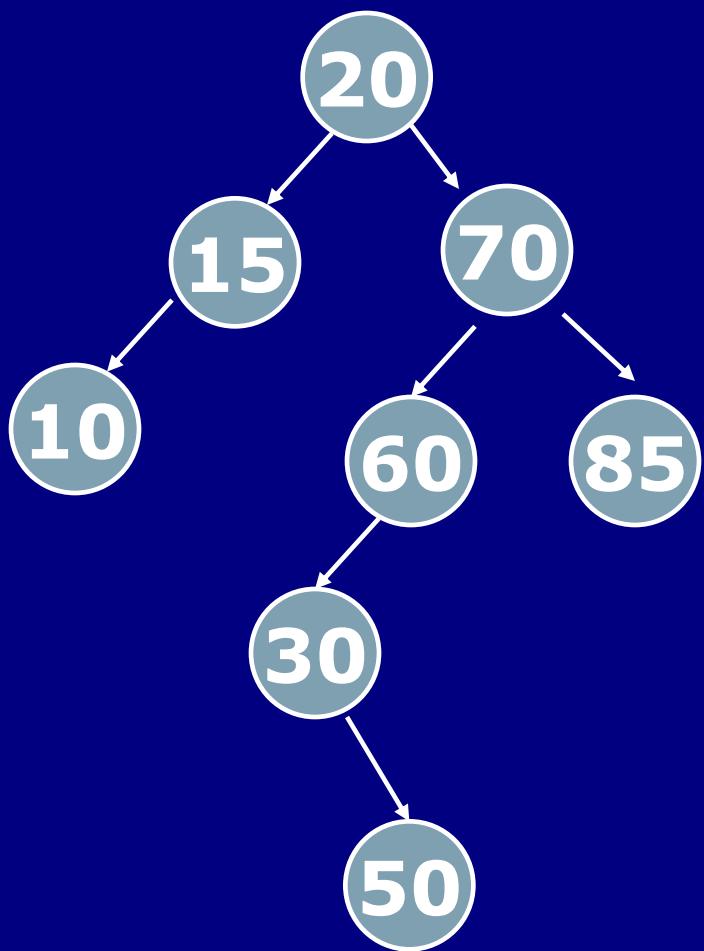
Insert 60, 30, 50, 65, 80, 90, 40, 5, 55



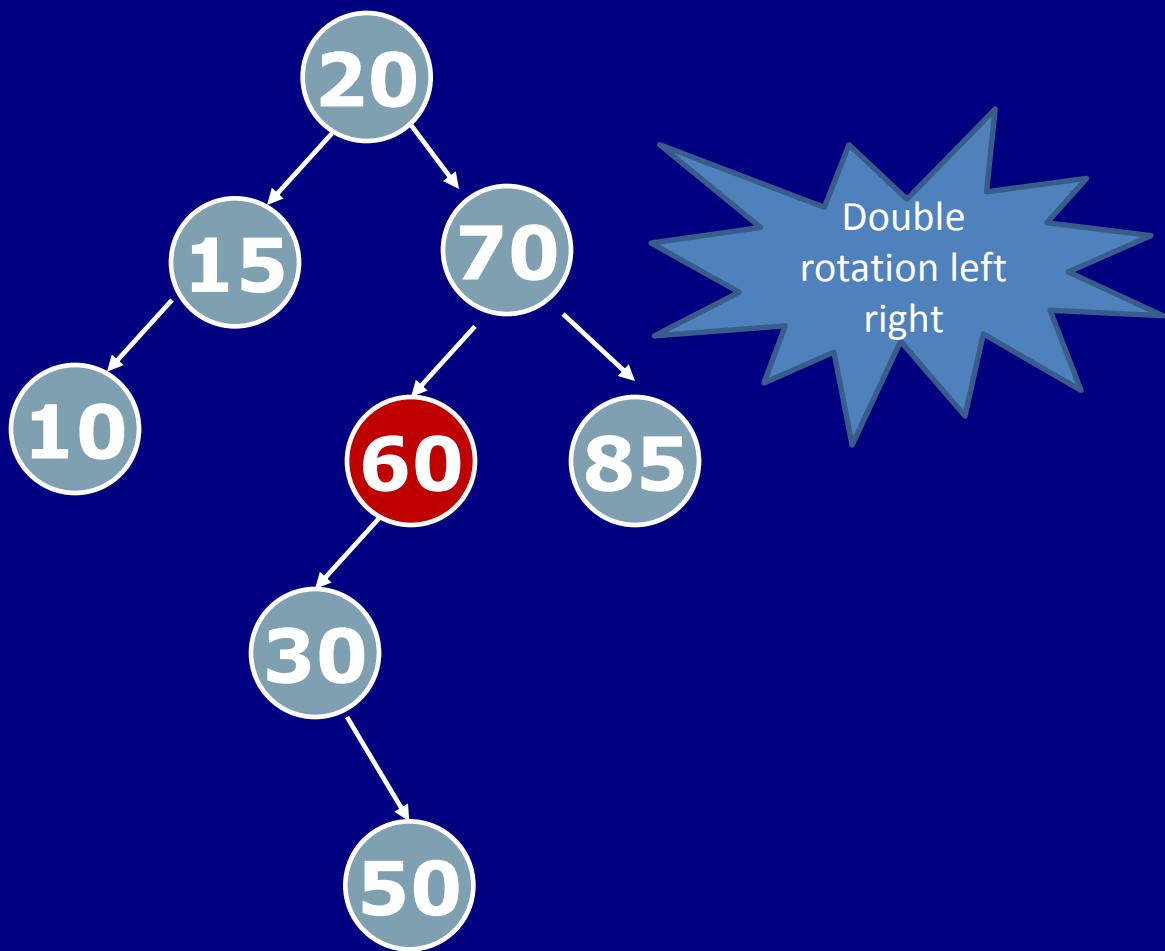
Insert 30, 50, 65, 80, 90, 40, 5, 55



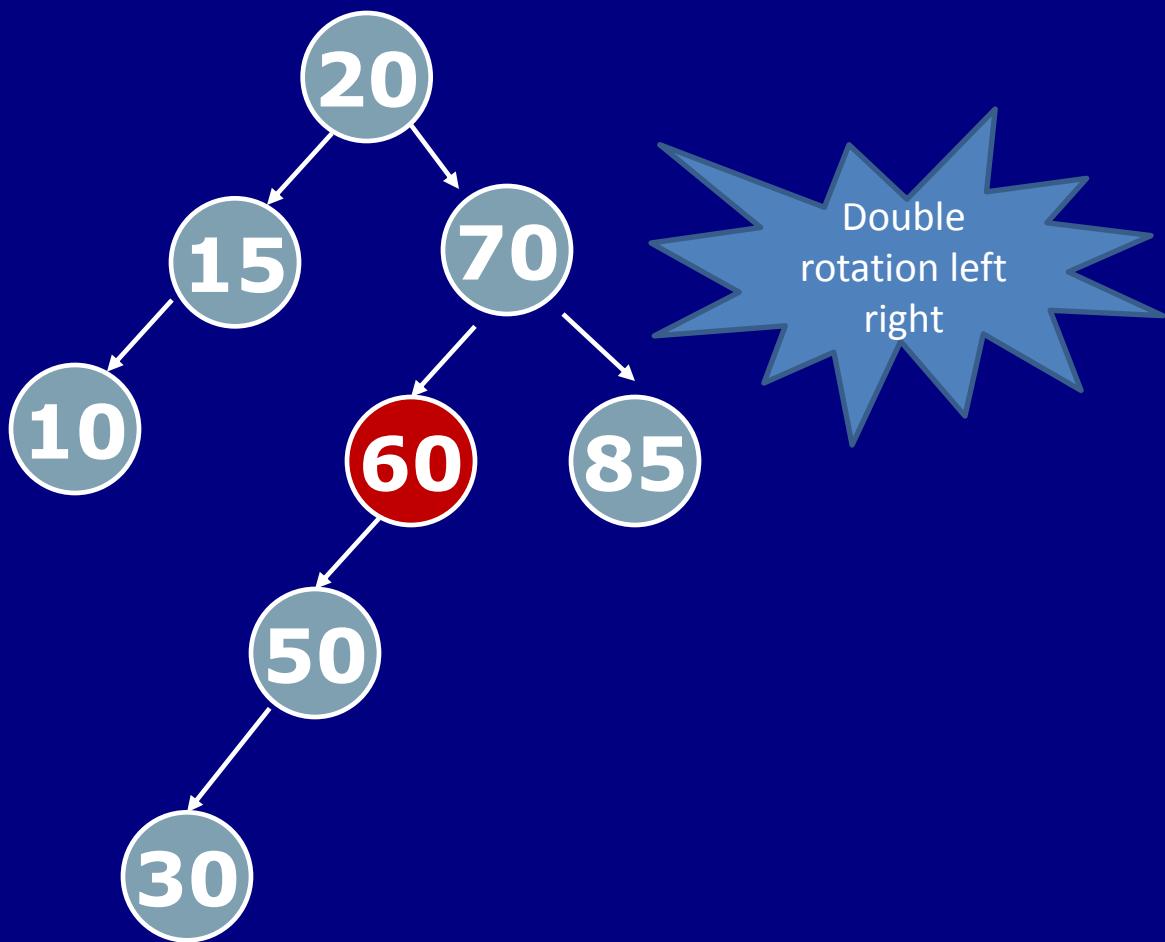
Insert 50, 65, 80, 90, 40, 5, 55



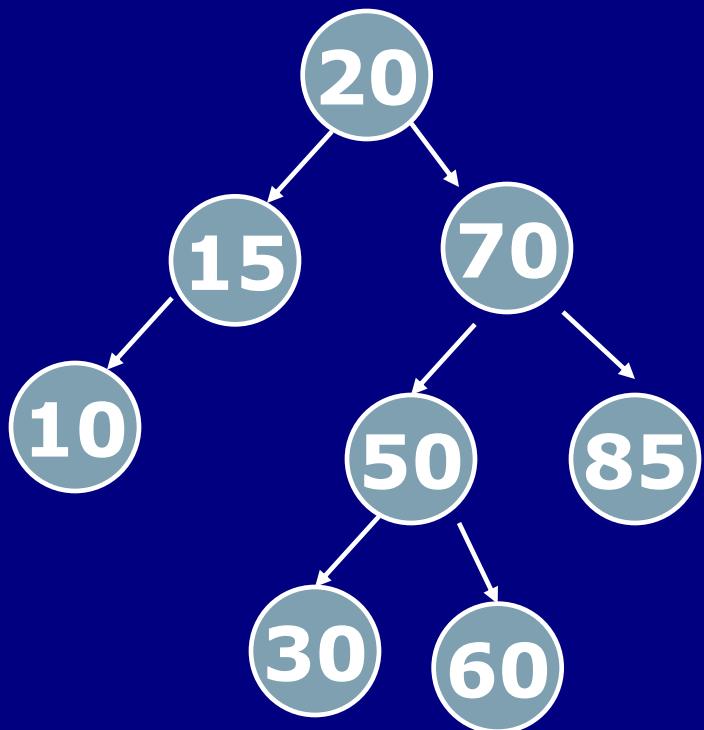
Insert 50, 65, 80, 90, 40, 5, 55



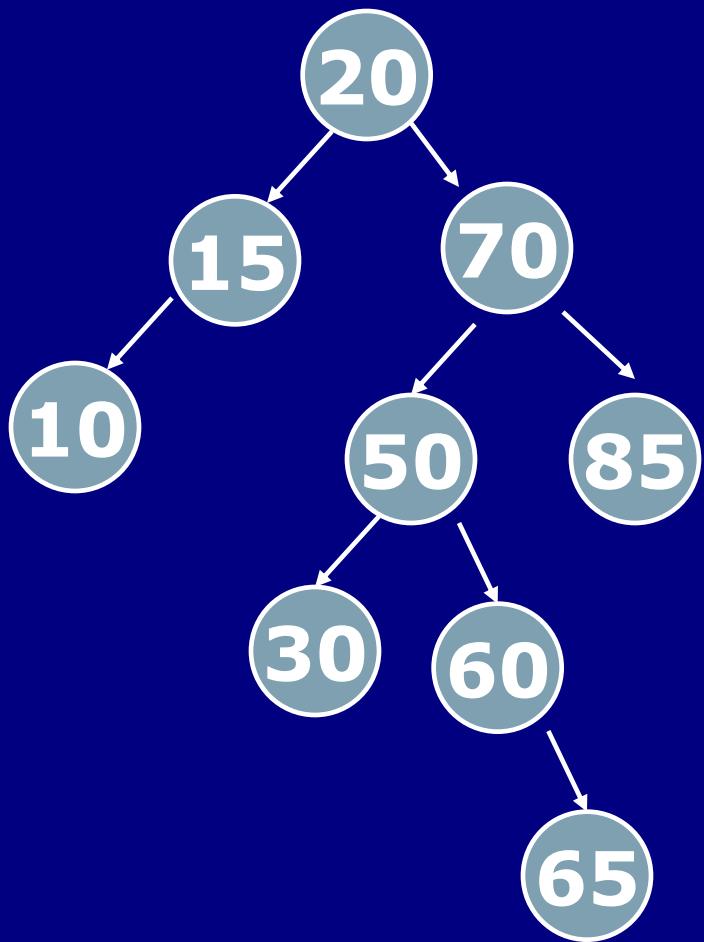
Insert 50, 65, 80, 90, 40, 5, 55



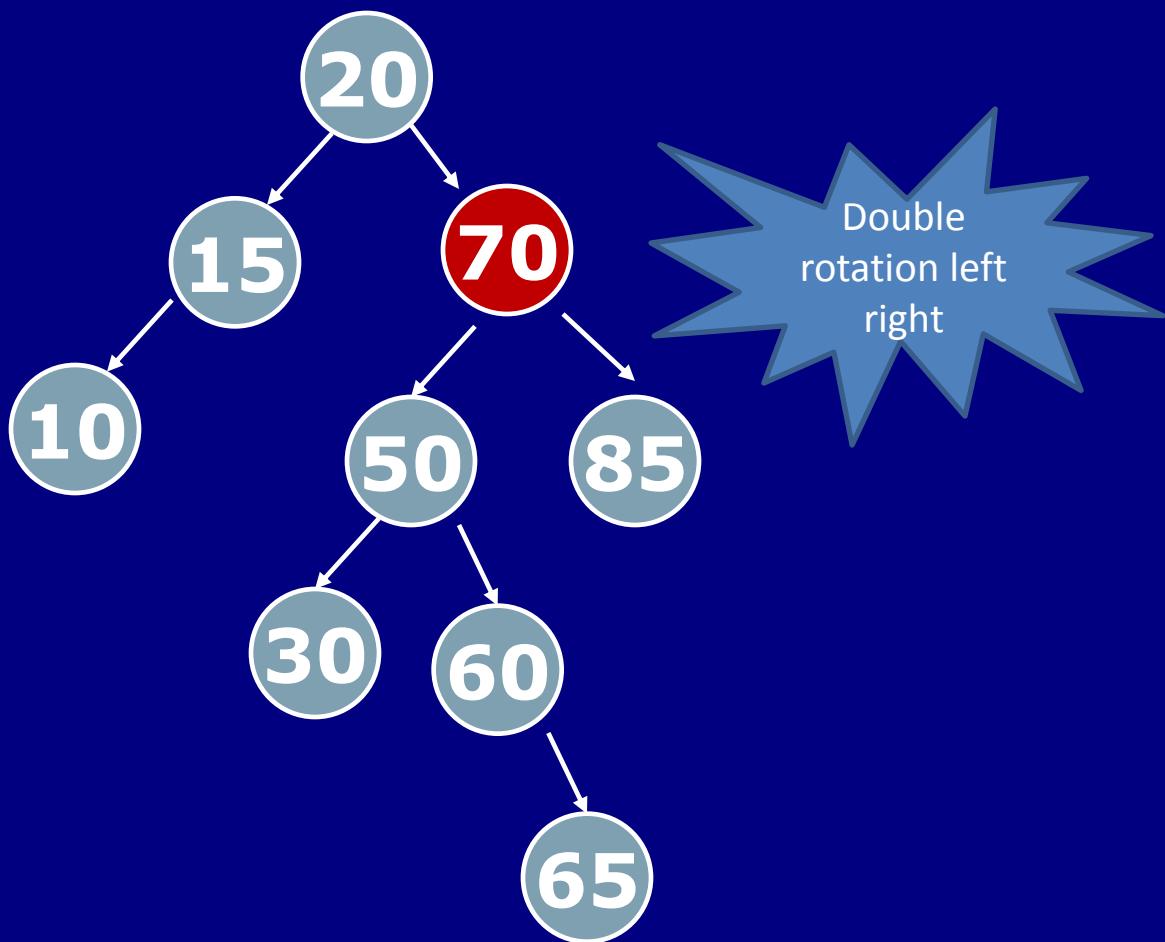
Insert 50, 65, 80, 90, 40, 5, 55



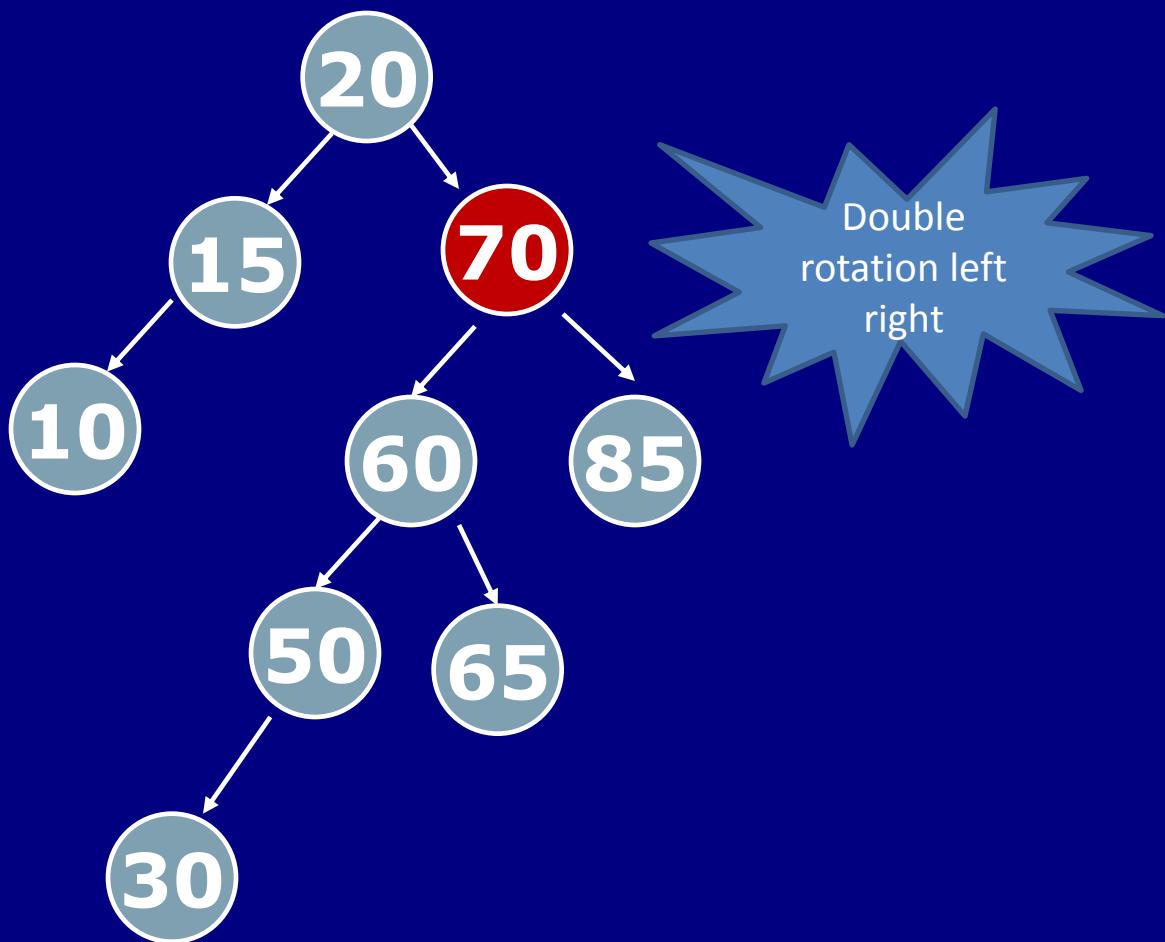
Insert 65, 80, 90, 40, 5, 55



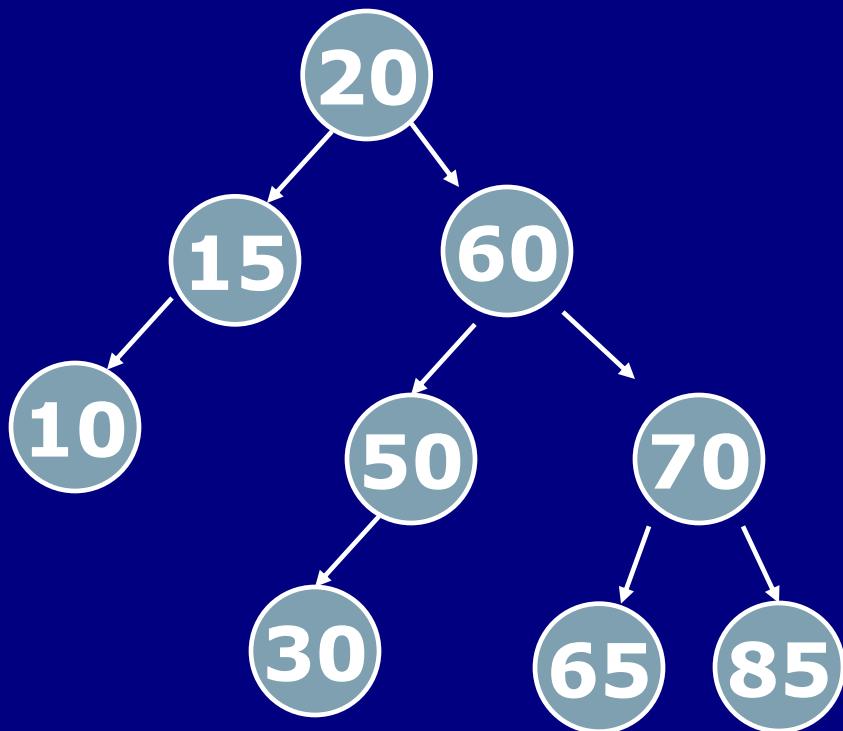
Insert 65, 80, 90, 40, 5, 55



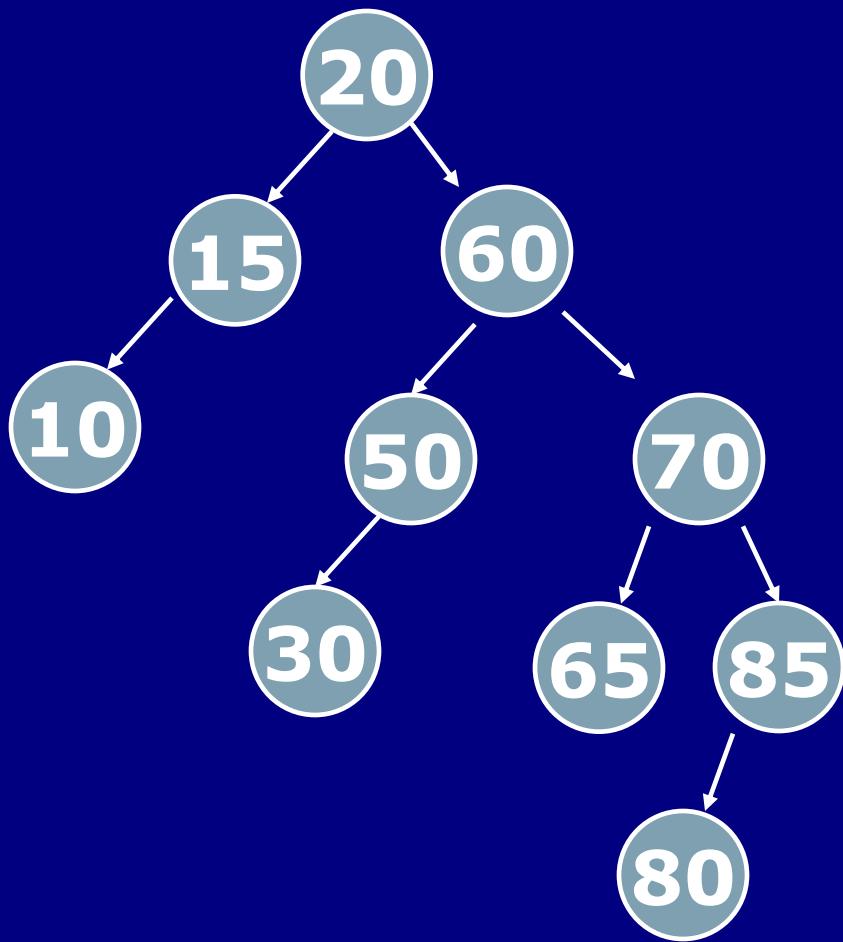
Insert 65, 80, 90, 40, 5, 55



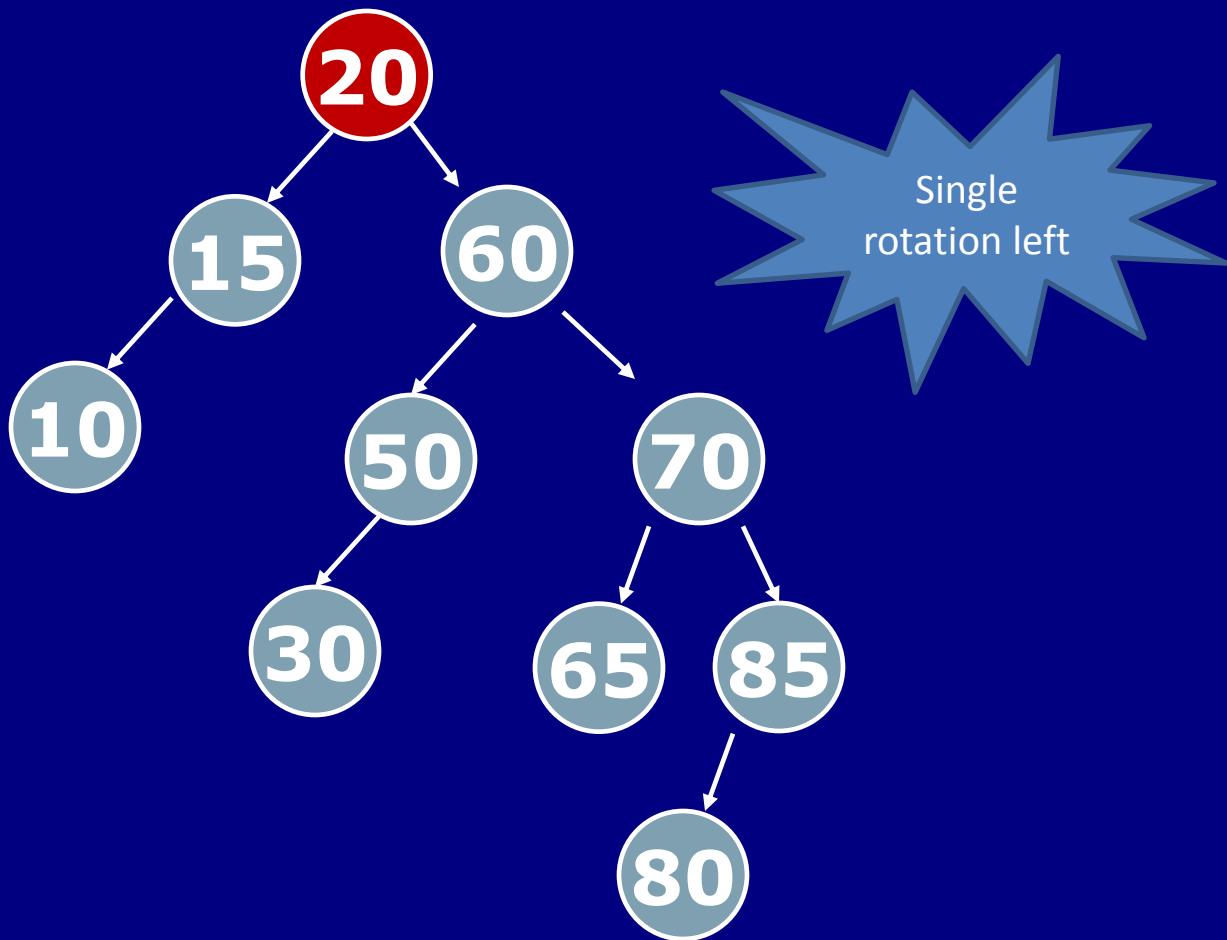
Insert 65, 80, 90, 40, 5, 55



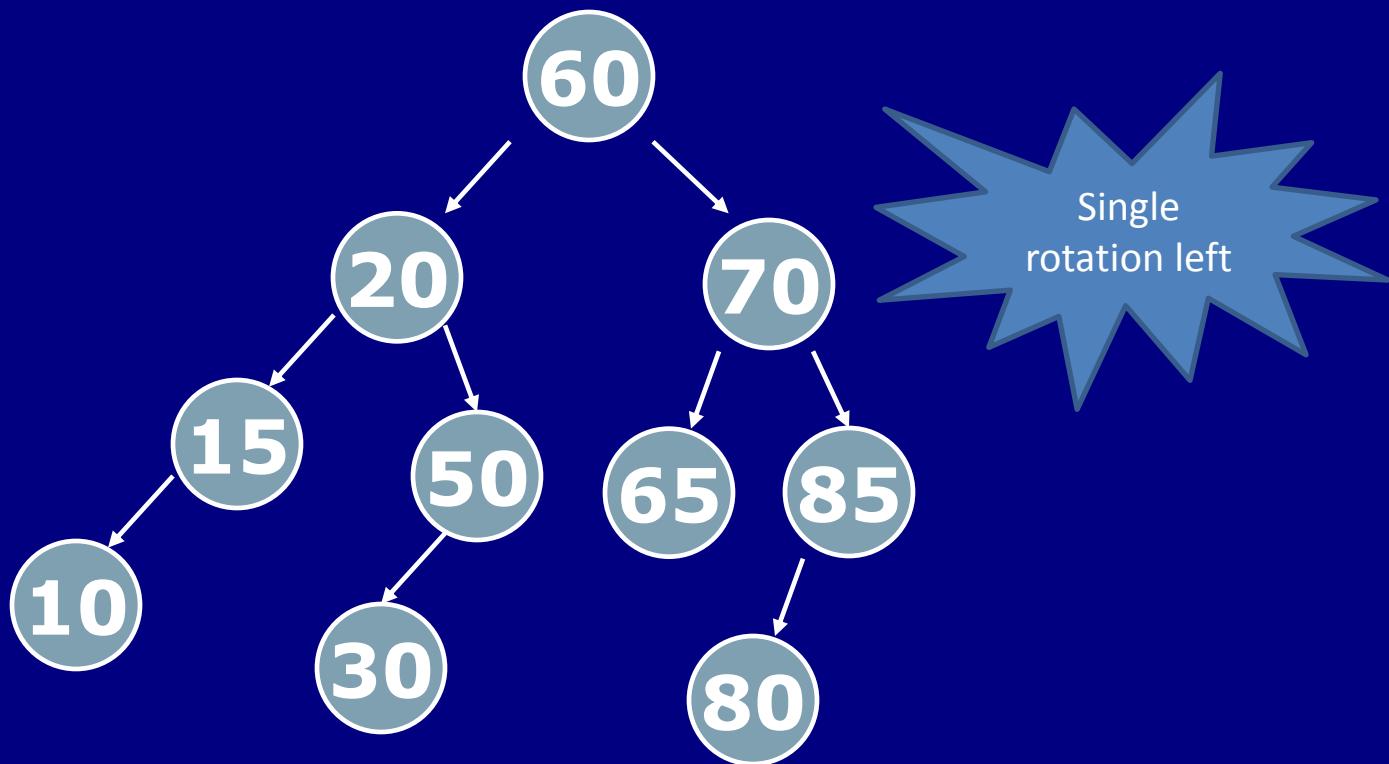
Insert 80, 90, 40, 5, 55



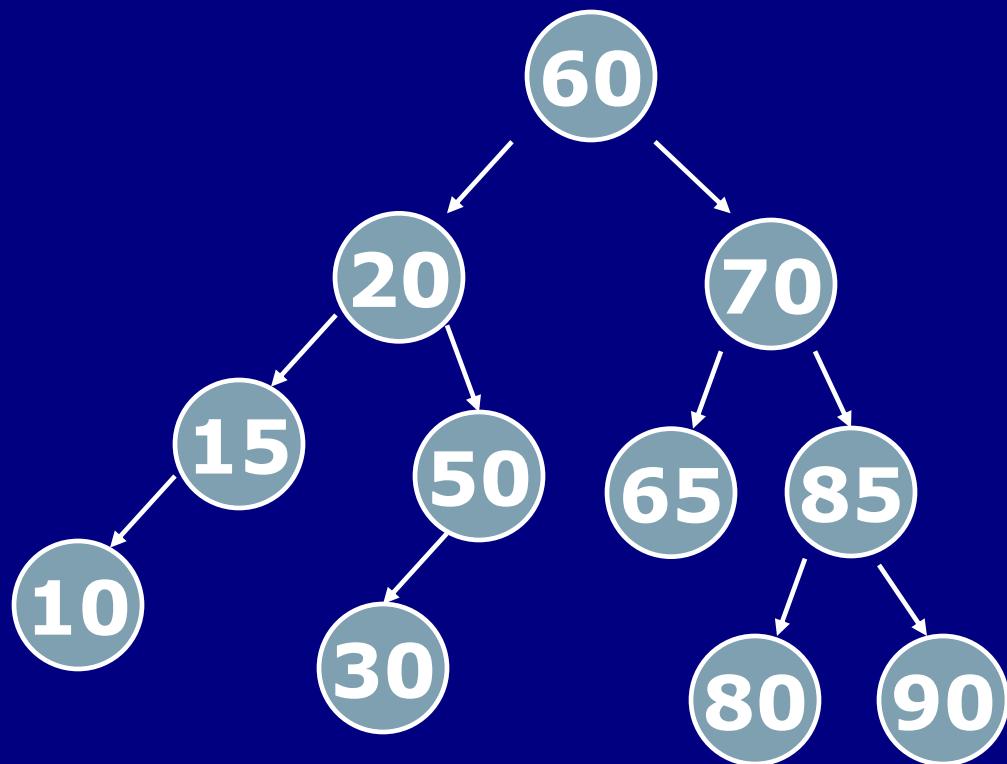
Insert 80, 90, 40, 5, 55



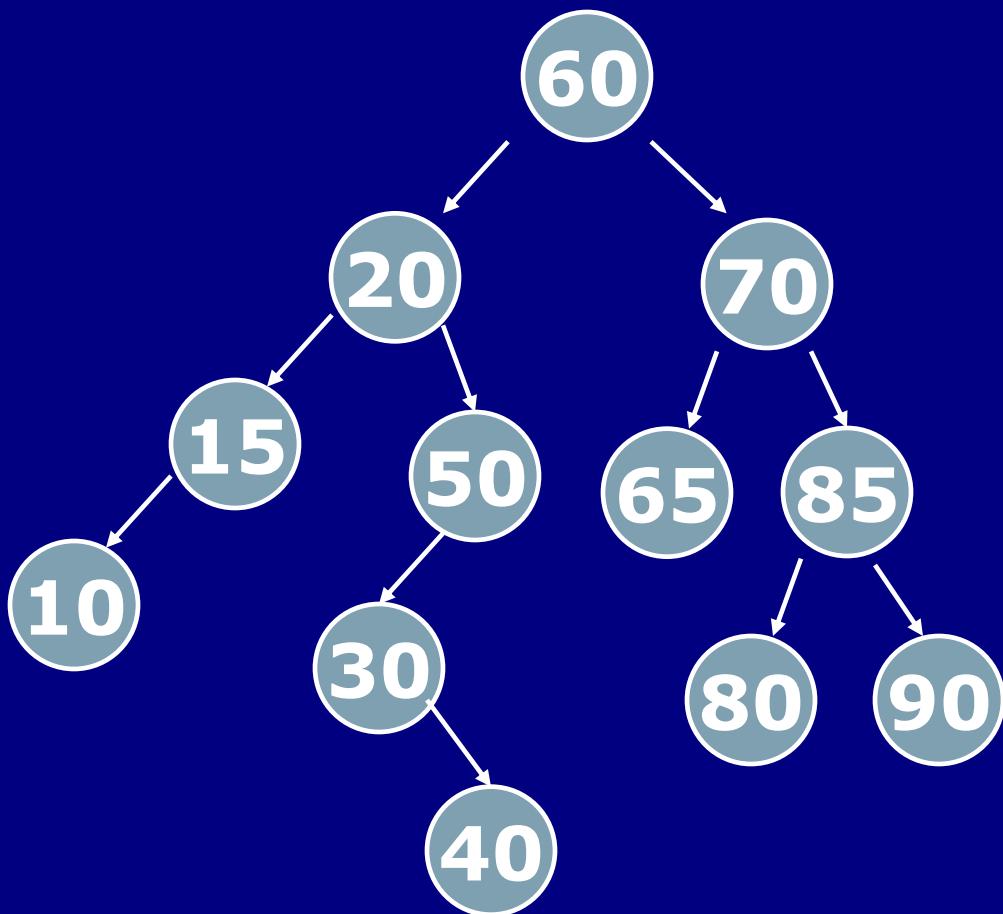
Insert 80, 90, 40, 5, 55



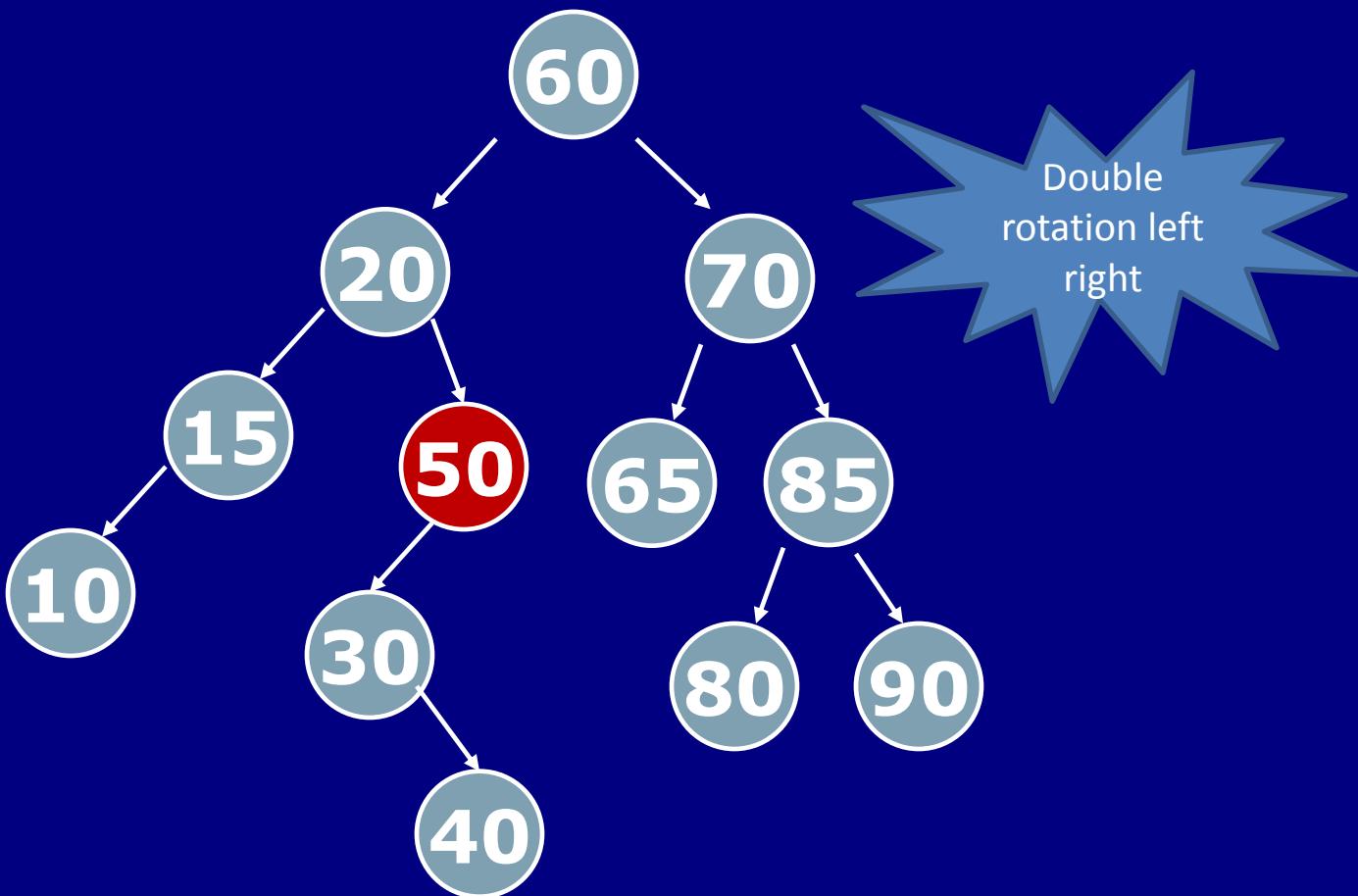
Insert 90, 40, 5, 55



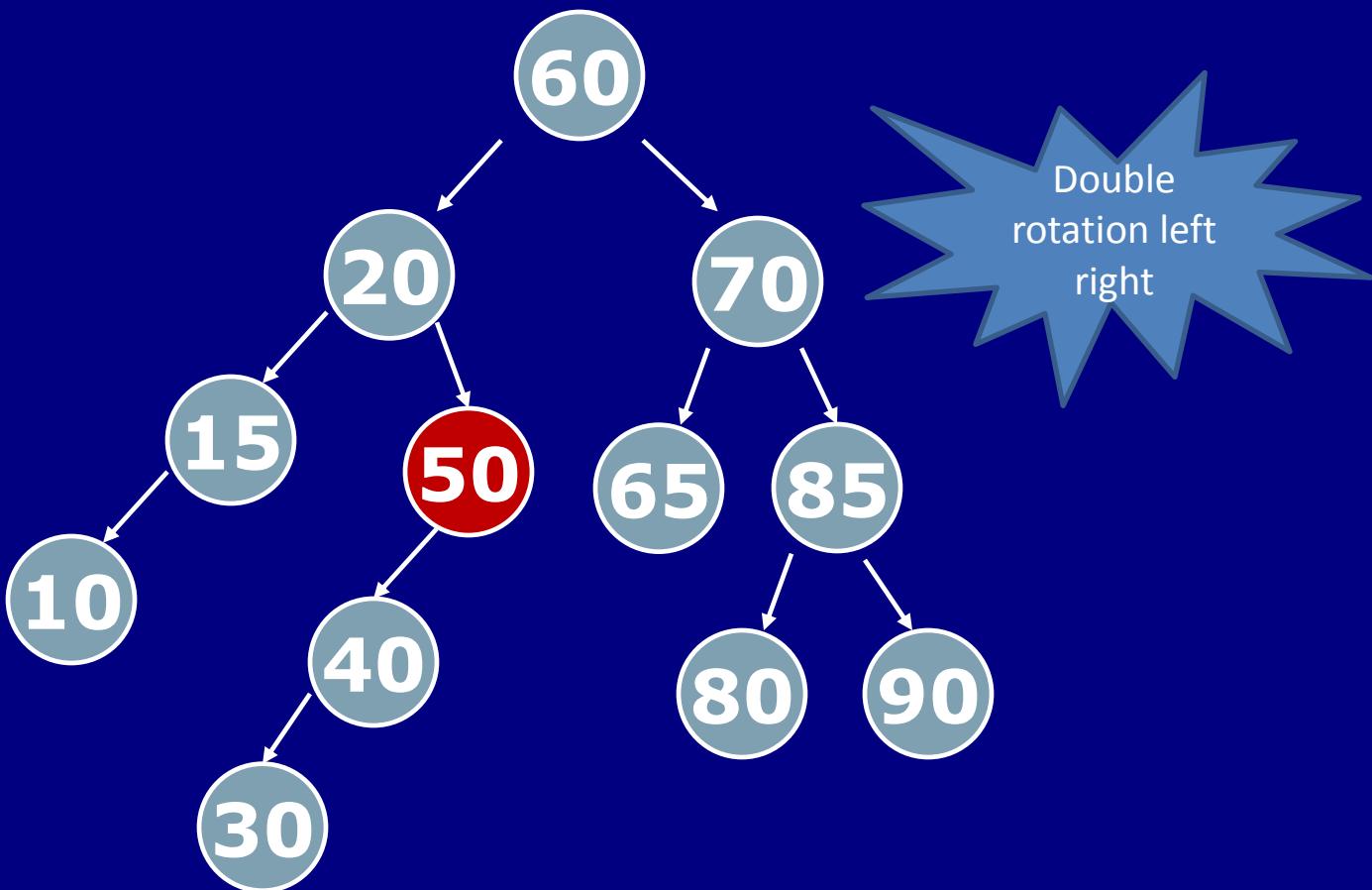
Insert 40, 5, 55



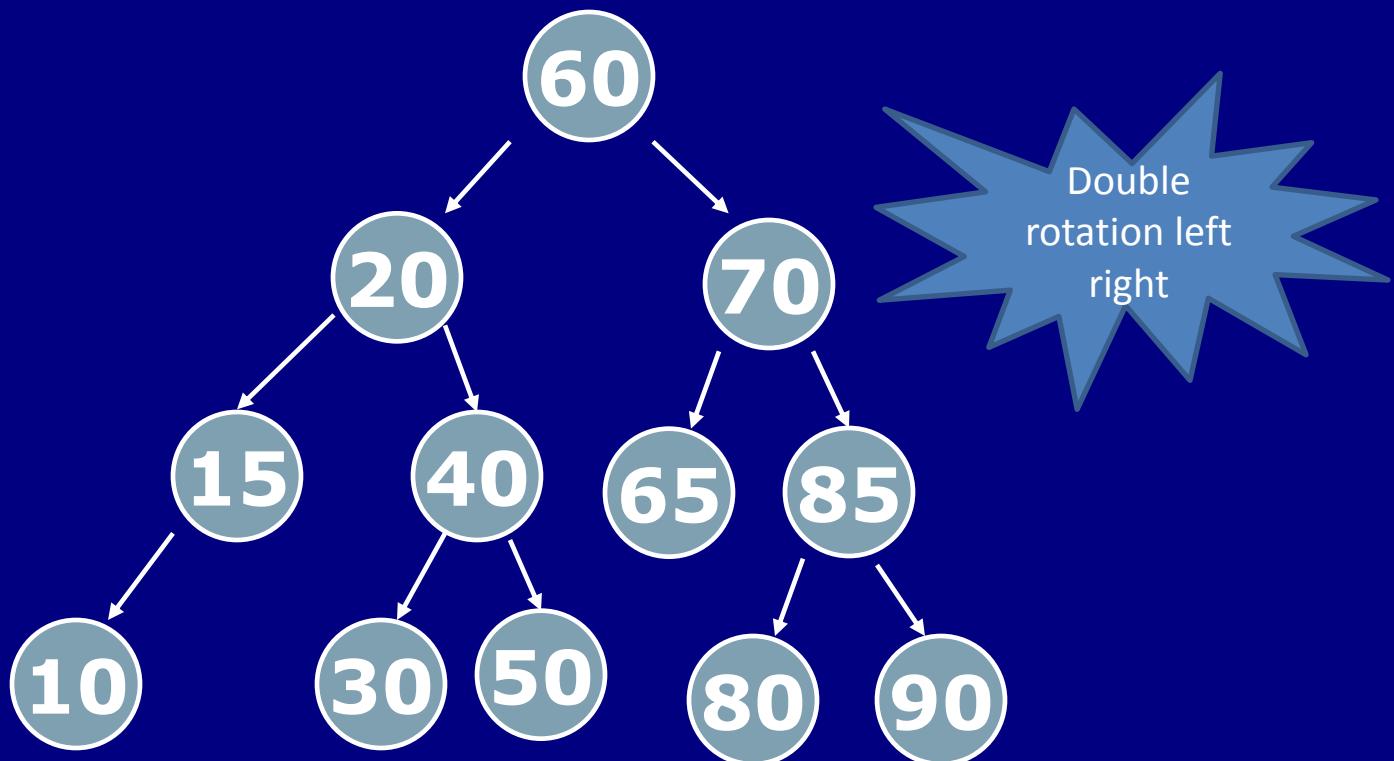
Insert 40, 5, 55



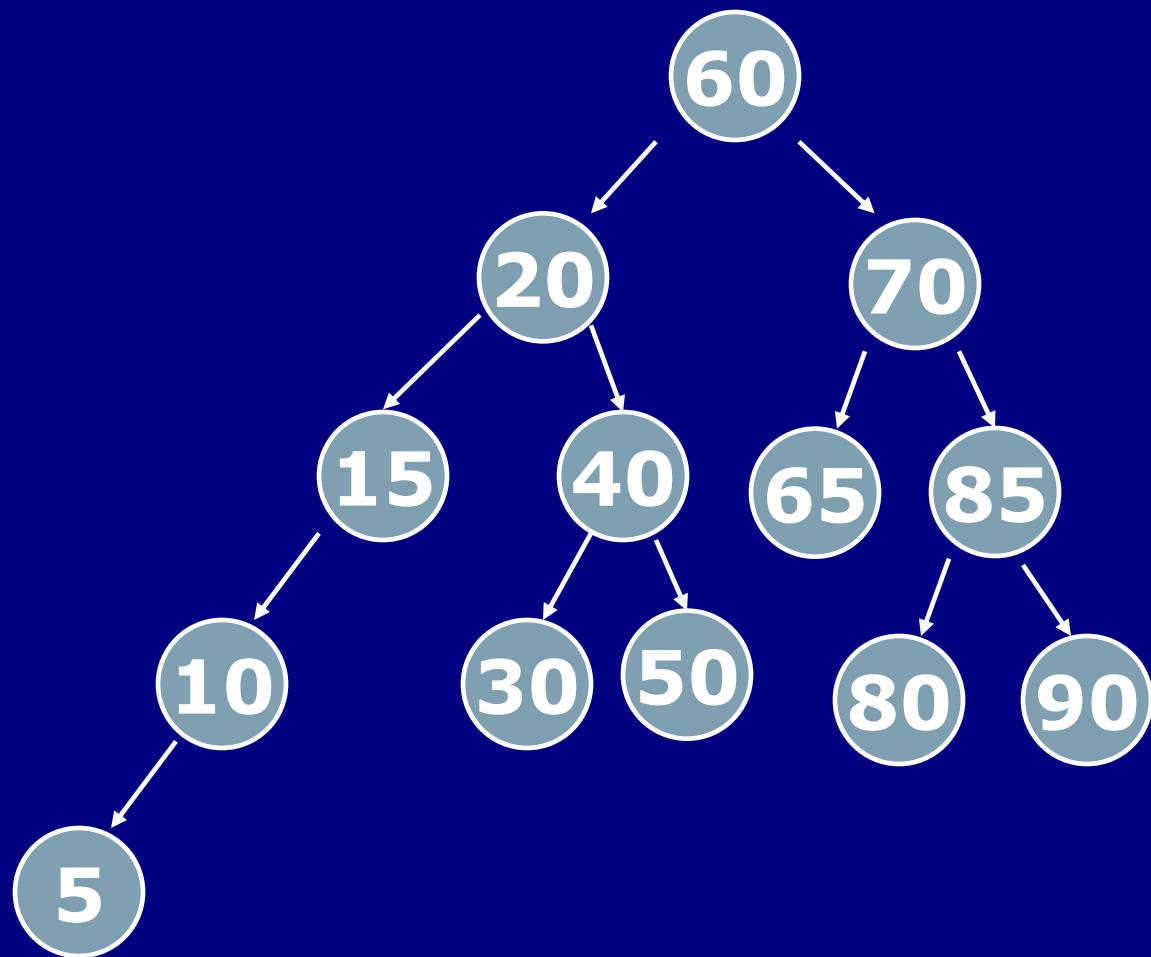
# Insert 40, 5, 55



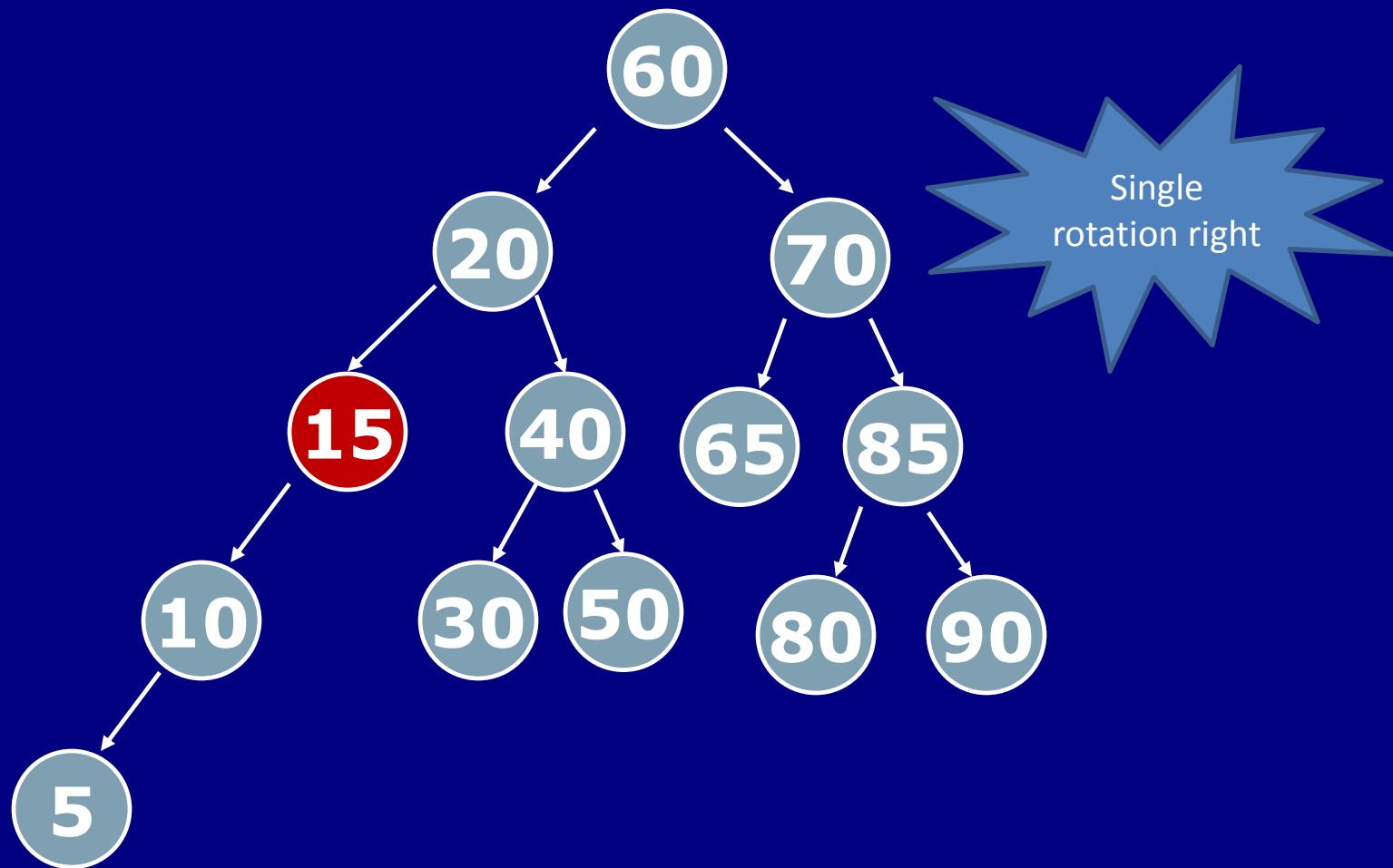
Insert 40, 5, 55



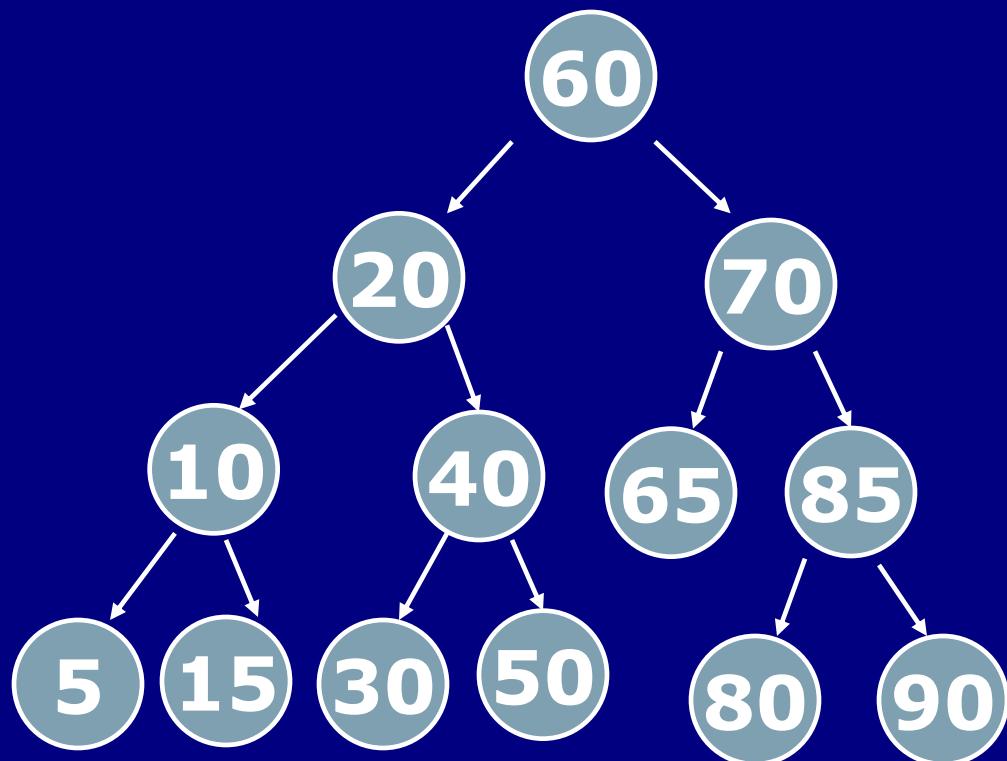
Insert 5, 55



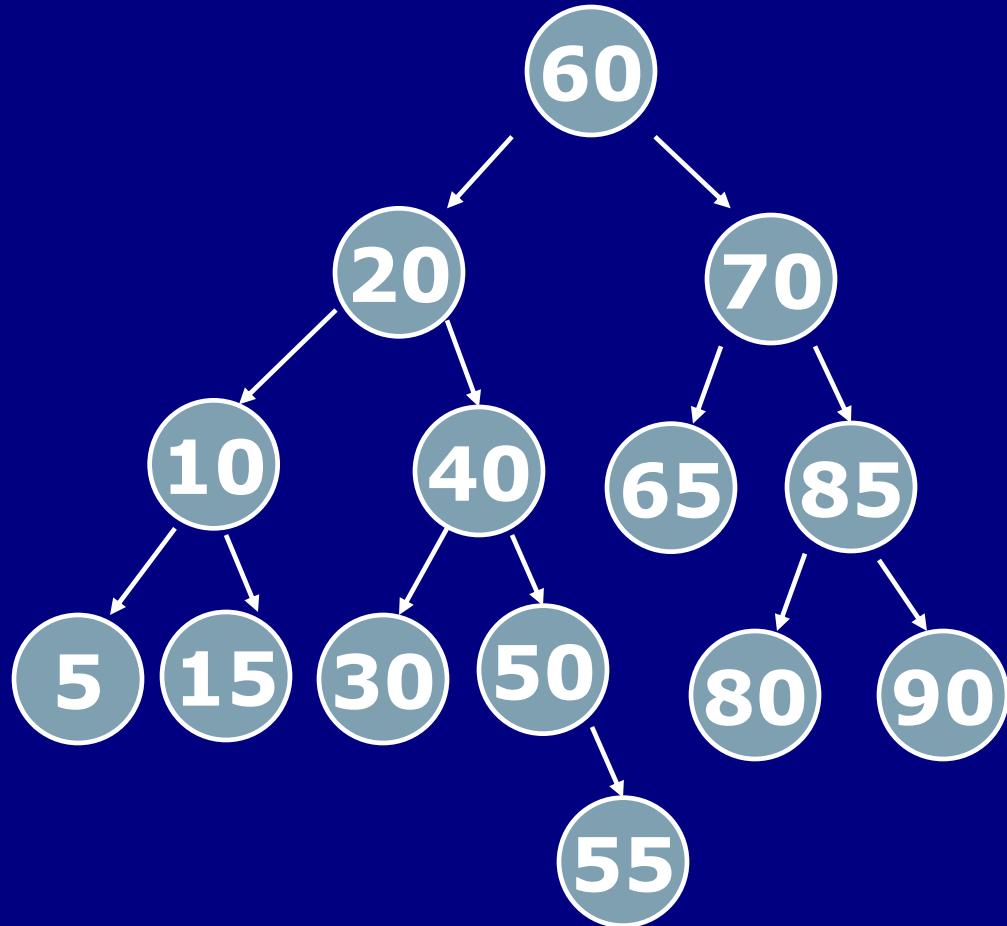
# Insert 5, 55



Insert 5, 55



# Insert 55



# AVL Trees: Latihan

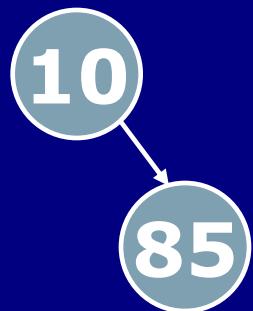
- Coba simulasi urutan proses pada sebuah AVL Tree berikut ini
  - insert 10, 85, 15, 70, 20, 60, 30,
  - delete 15, 10,
  - insert 50, 65, 80,
  - delete 20, 60,
  - insert 90, 40, 5, 55
  - delete 70
- Gambarkan kondisi akhir dari AVL Tree tersebut.



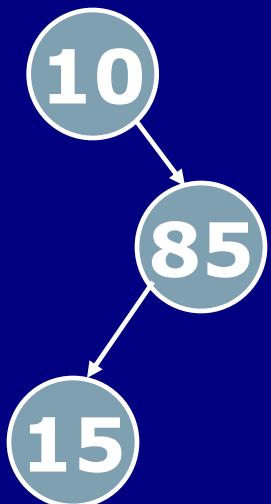
Insert 10, 85, 15, 70, 20, 60, 30

10

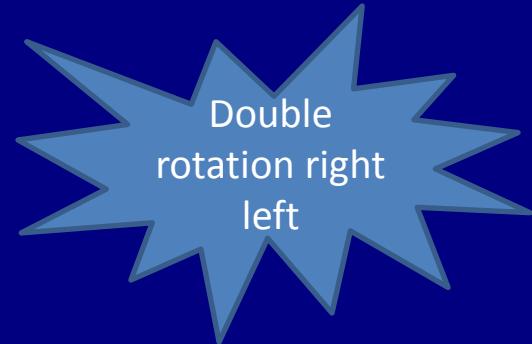
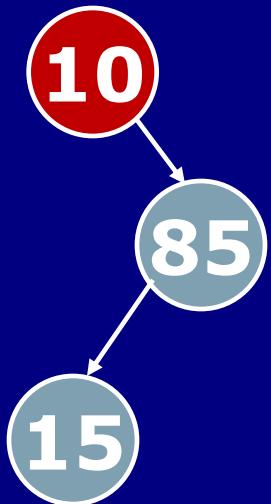
Insert 85, 15, 70, 20, 60, 30



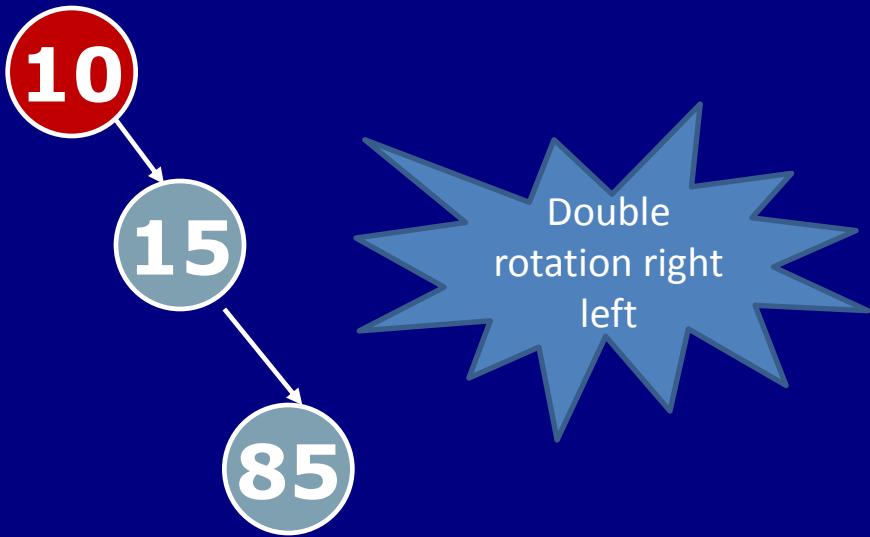
Insert 15, 70, 20, 60, 30



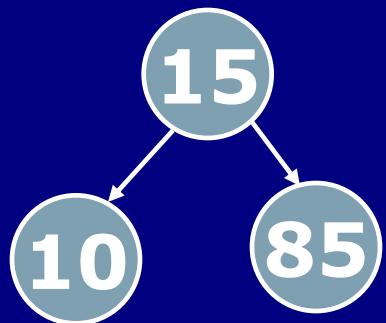
Insert 15, 70, 20, 60, 30



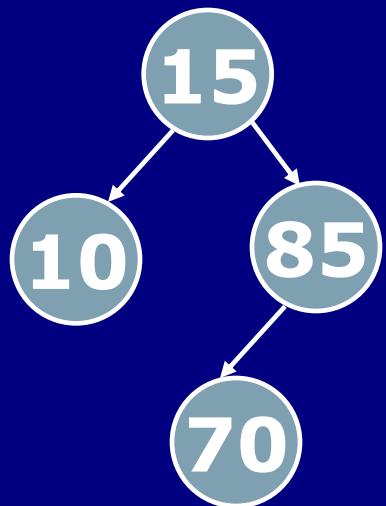
Insert 15, 70, 20, 60, 30



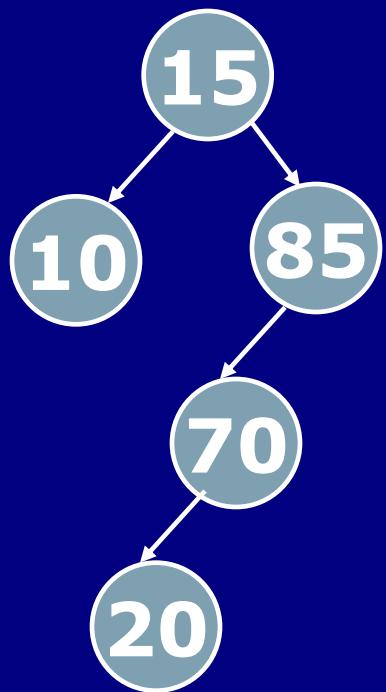
Insert 15, 70, 20, 60, 30



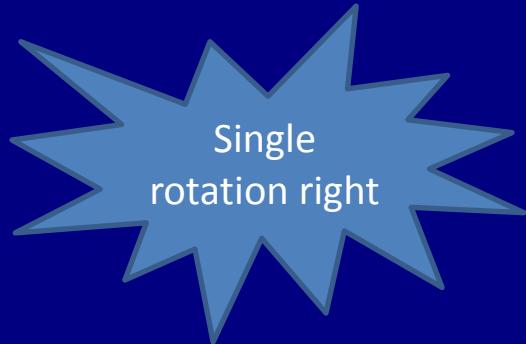
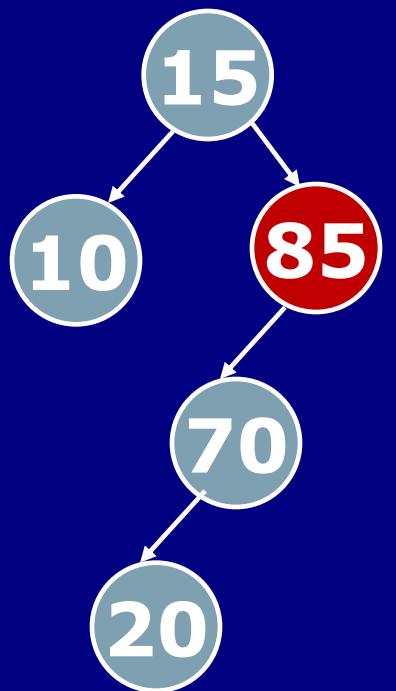
Insert 70, 20, 60, 30



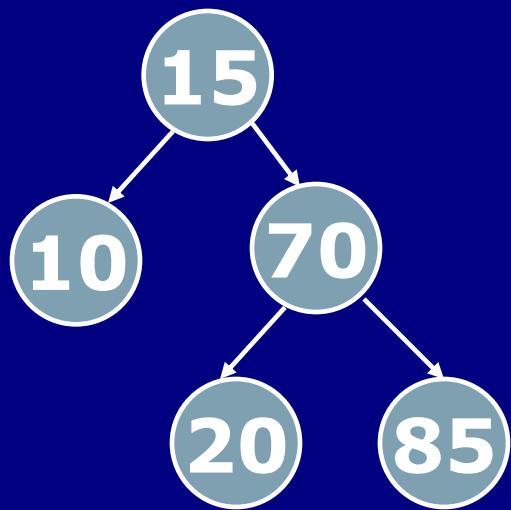
Insert 20, 60, 30



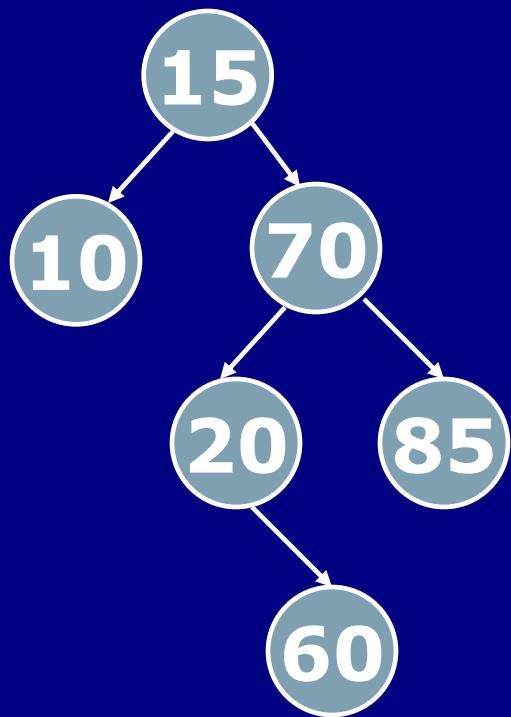
Insert 20, 60, 30



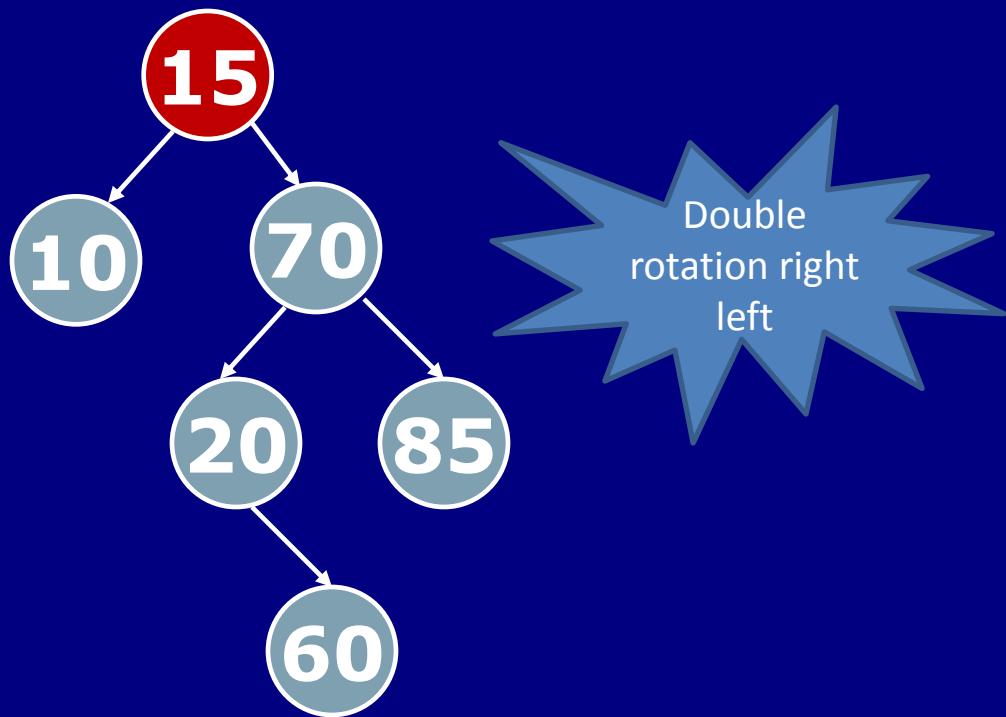
Insert 20, 60, 30



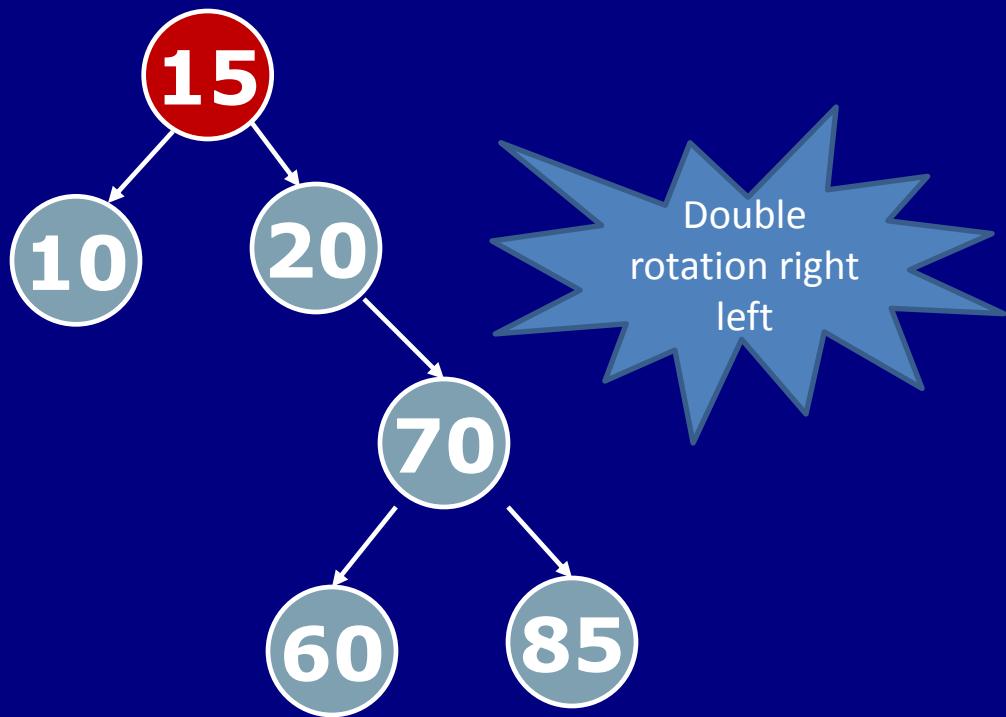
Insert 60, 30



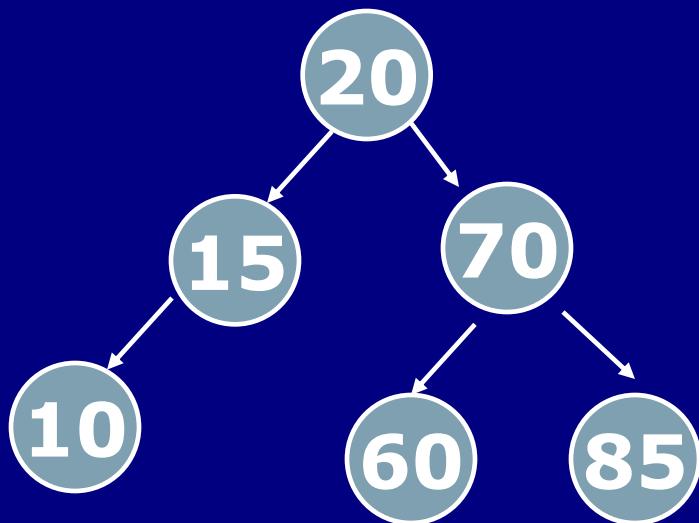
# Insert 60, 30



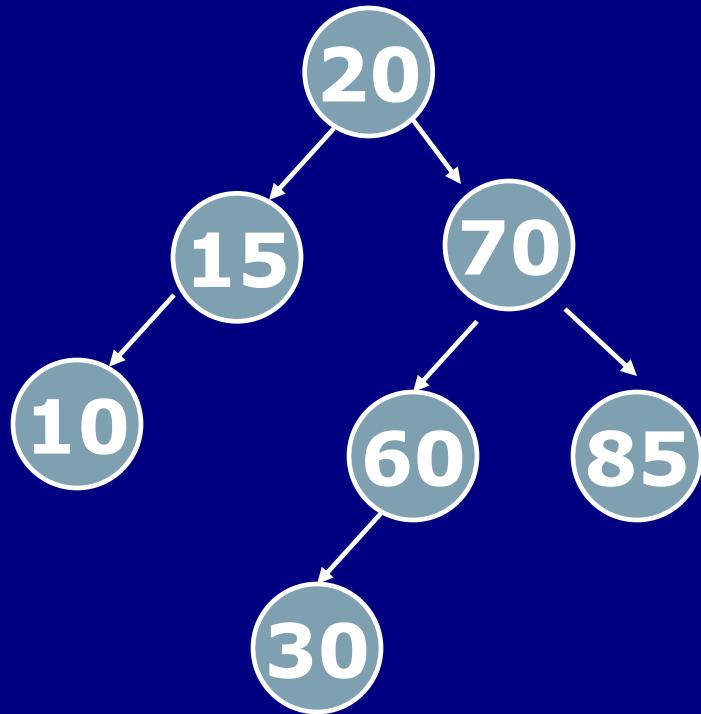
# Insert 60, 30



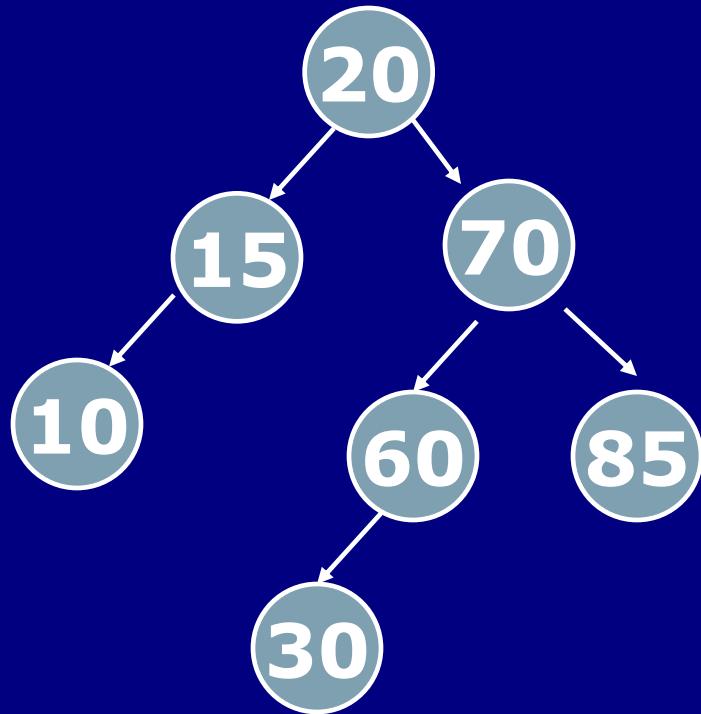
Insert 60, 30



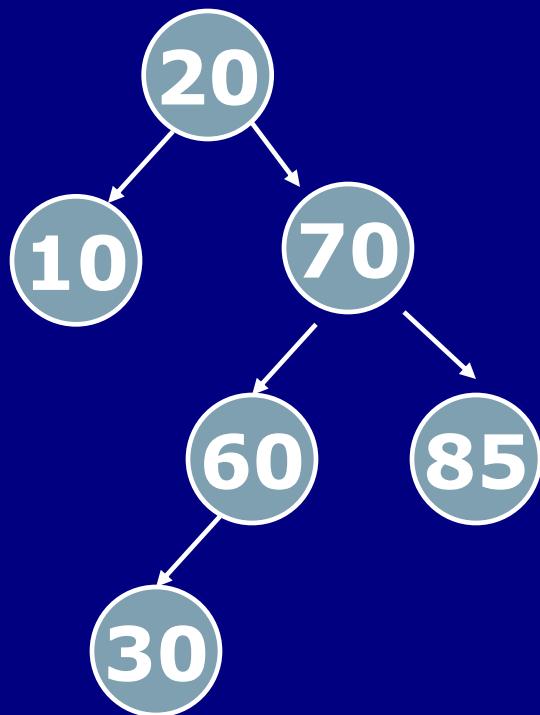
# Insert 30



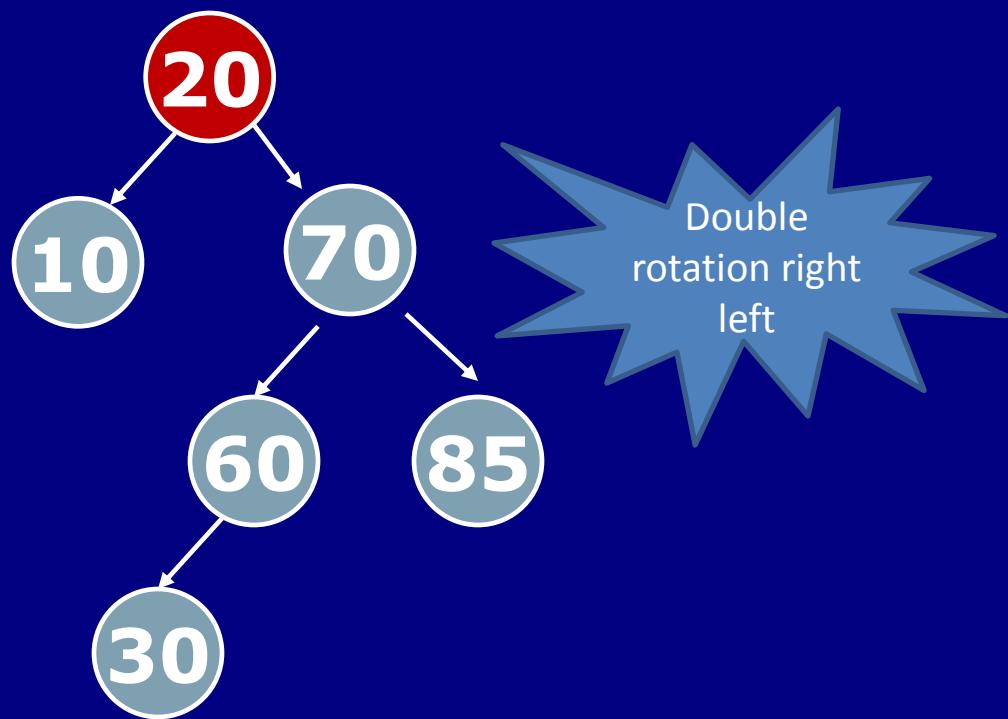
# Delete 15, 10



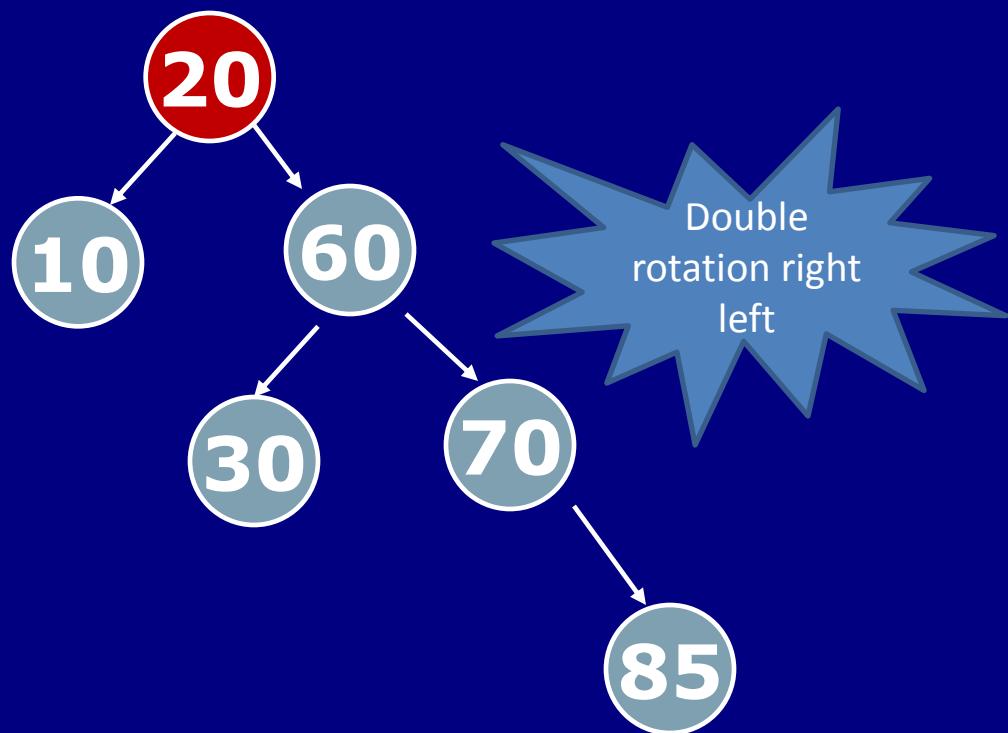
Delete 15, 10



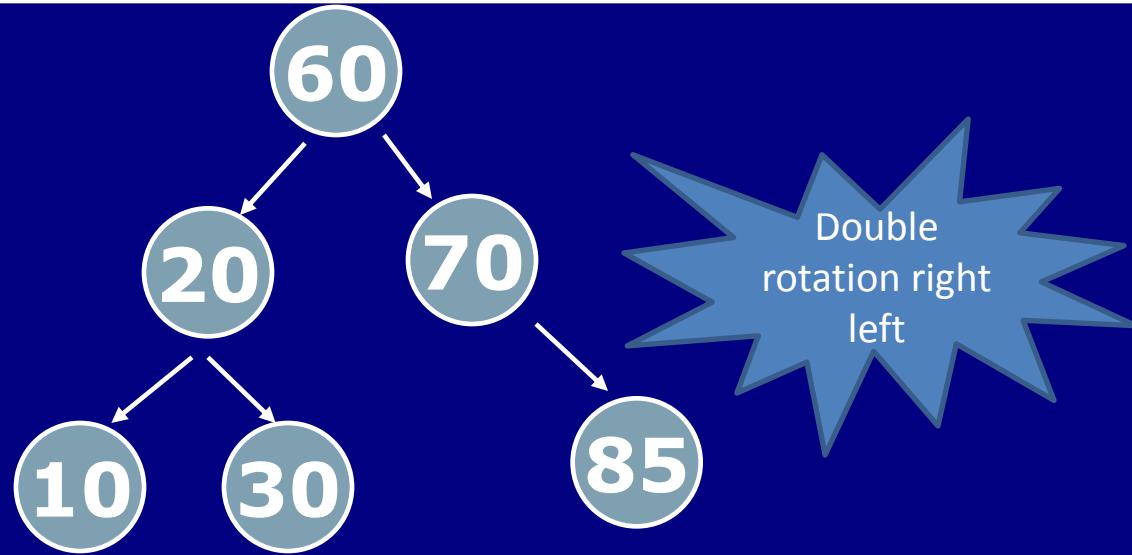
# Delete 15, 10



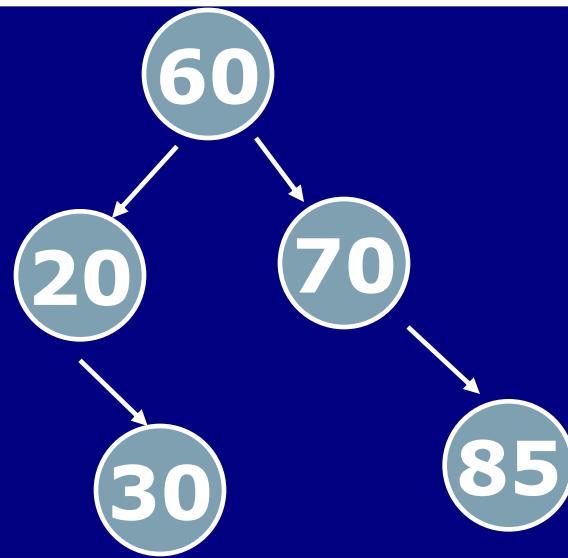
# Delete 15, 10



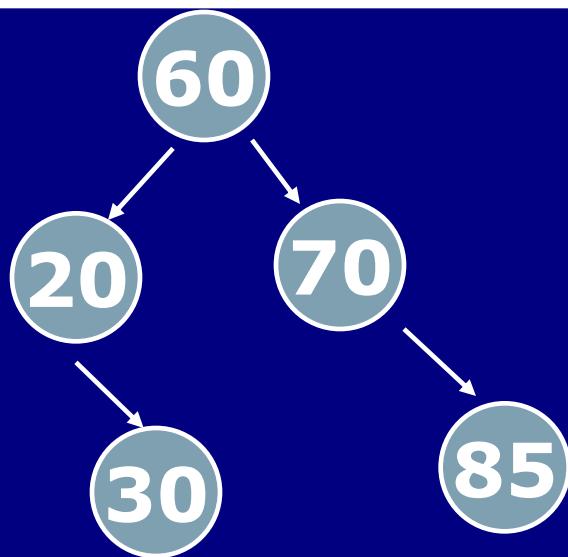
# Delete 15, 10



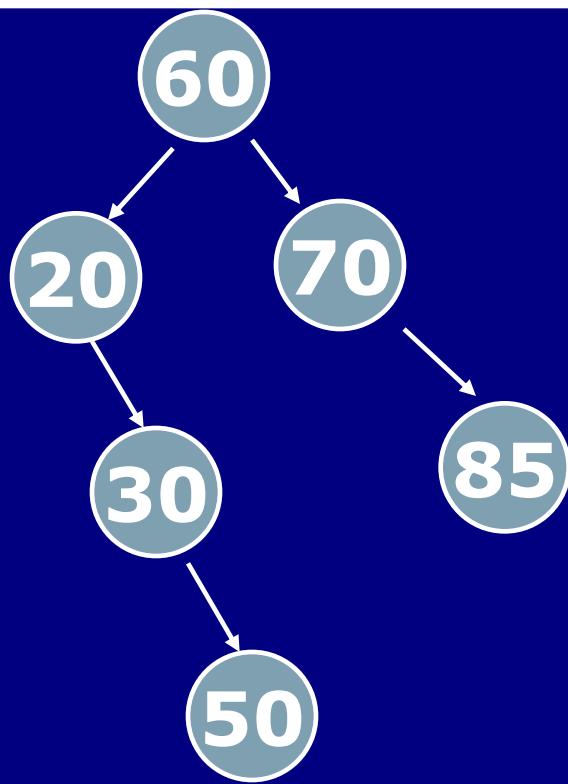
# Delete 10



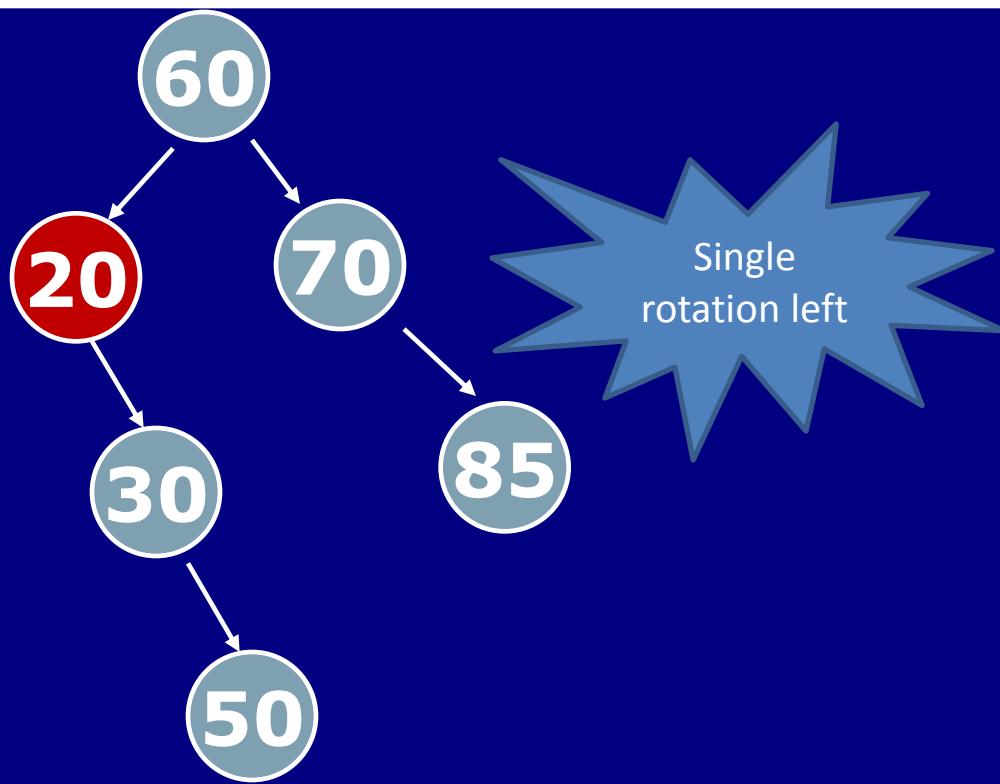
# Insert 50, 65, 80



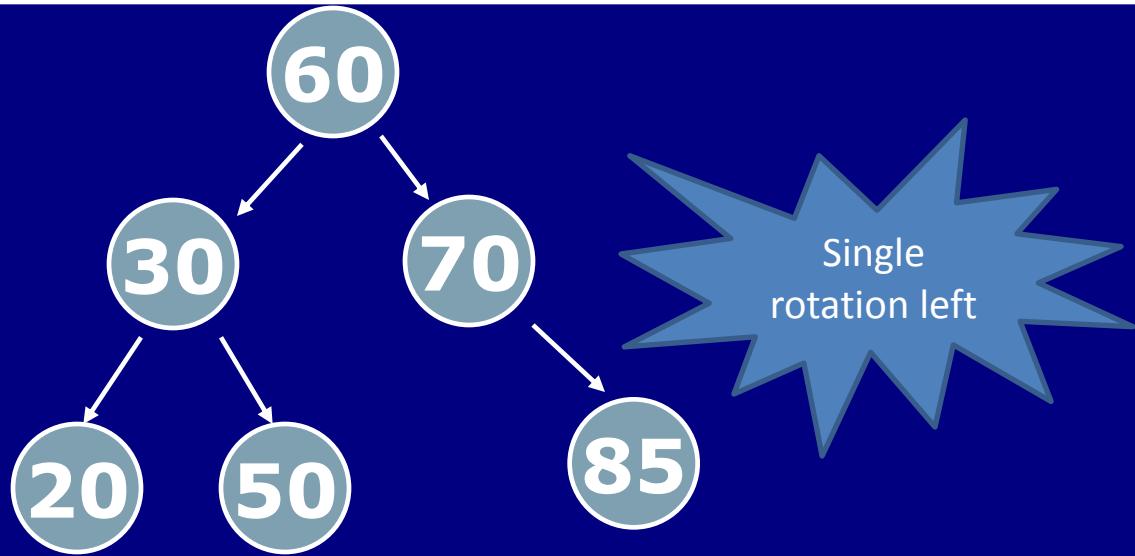
Insert 50, 65, 80



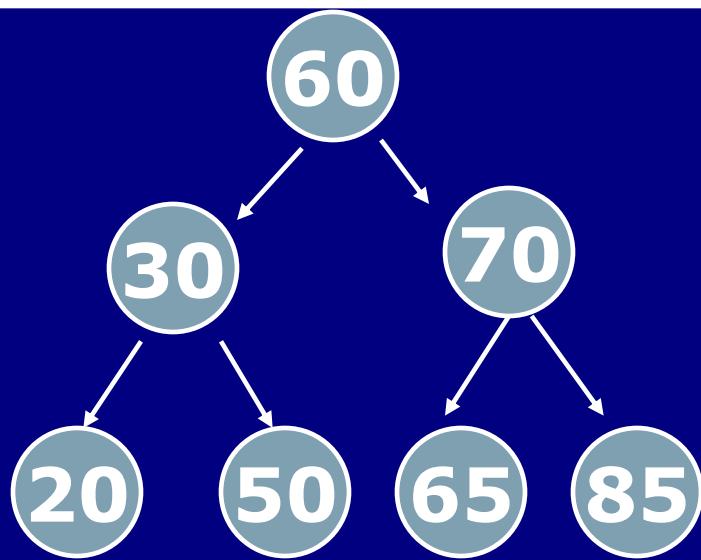
# Insert 50, 65, 80



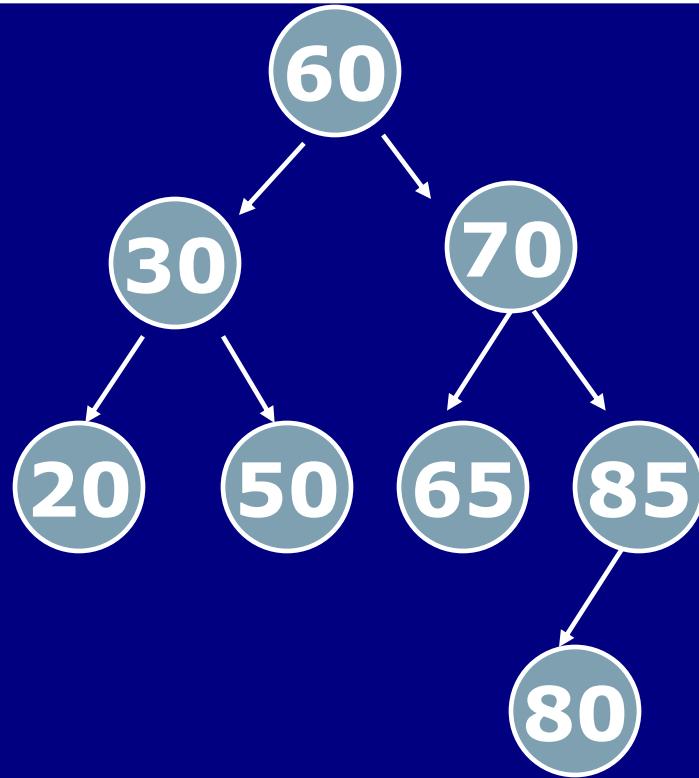
# Insert 50, 65, 80



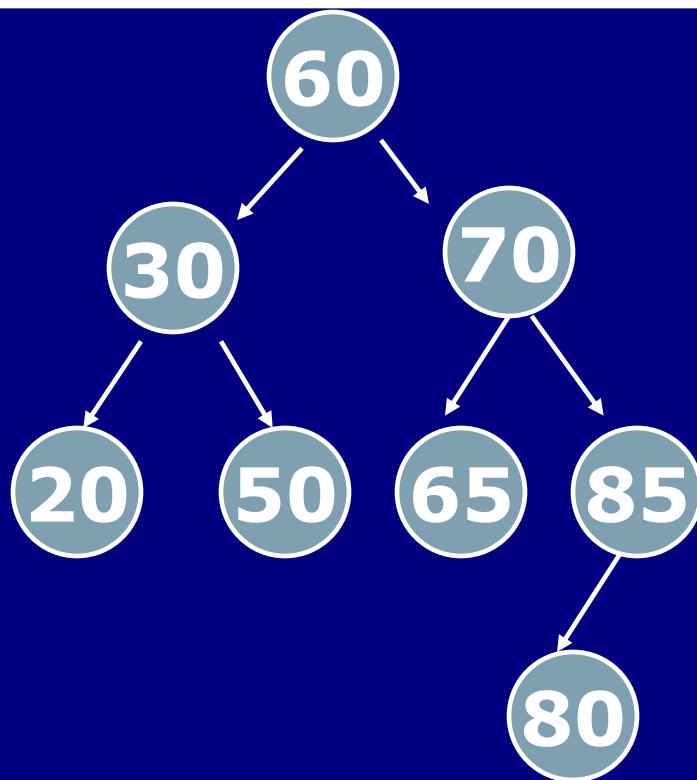
Insert 65, 80



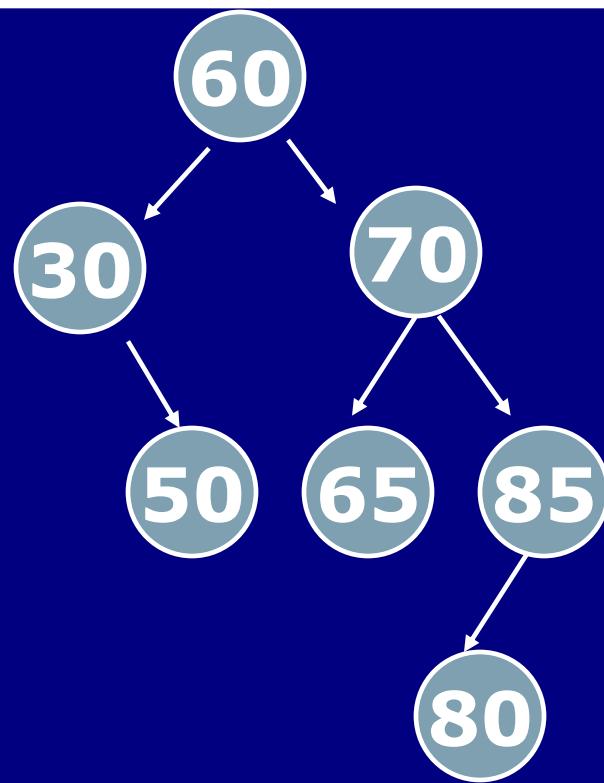
# Insert 80



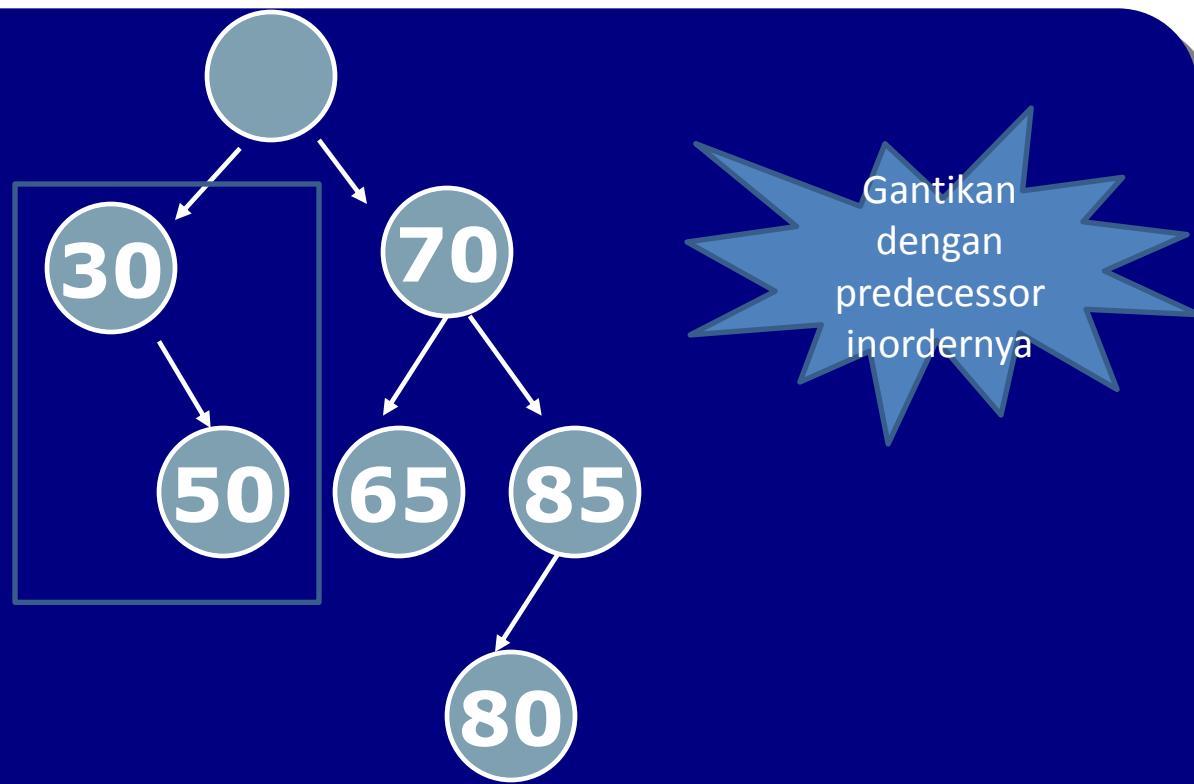
# Delete 20, 60



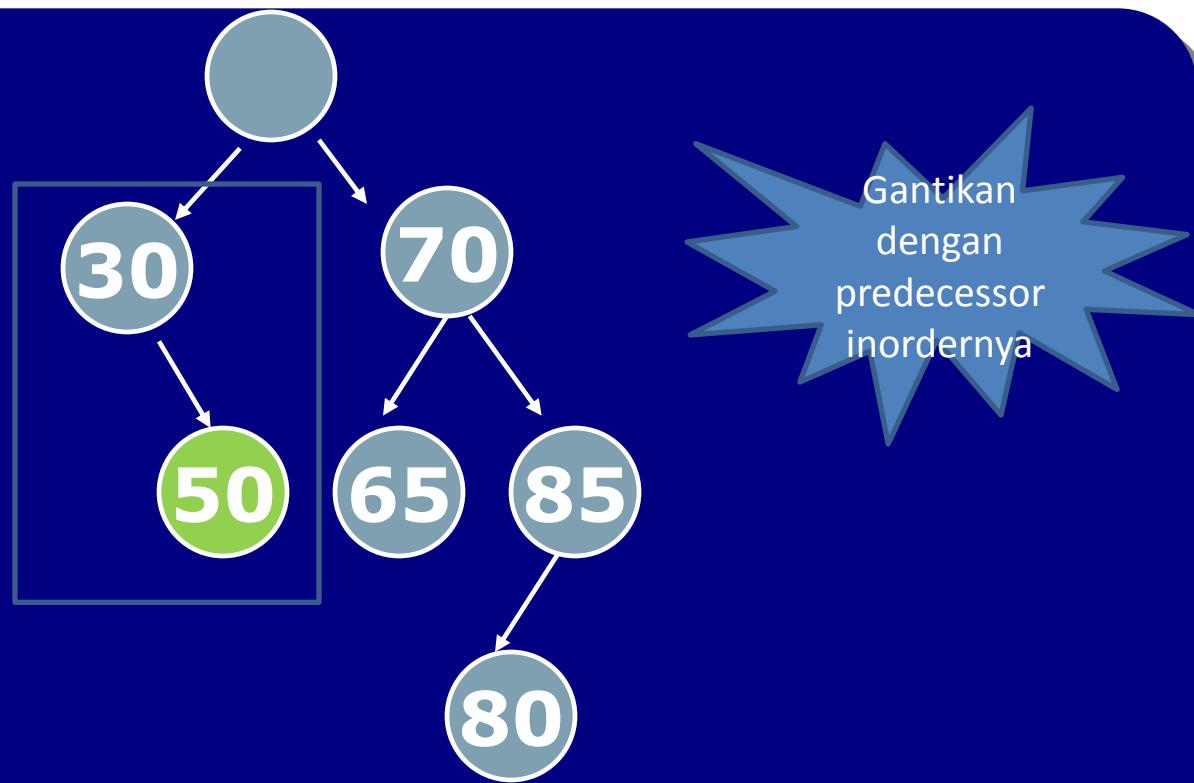
Delete 20, 60



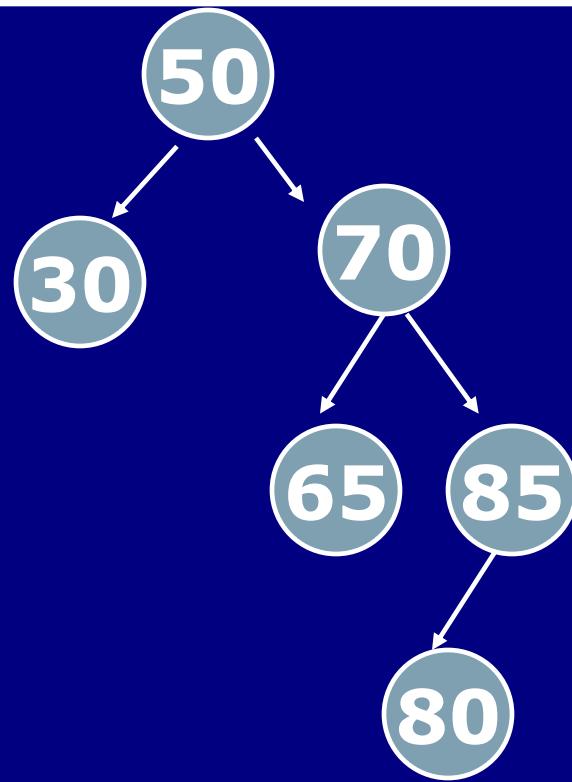
# Delete 60



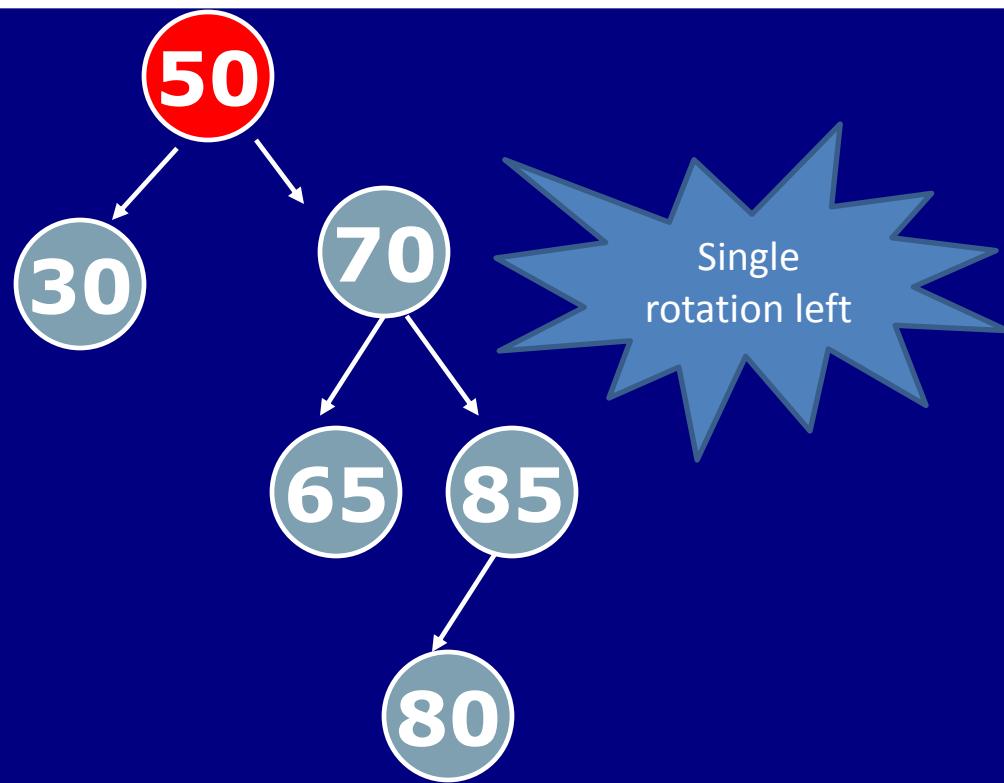
# Delete 60



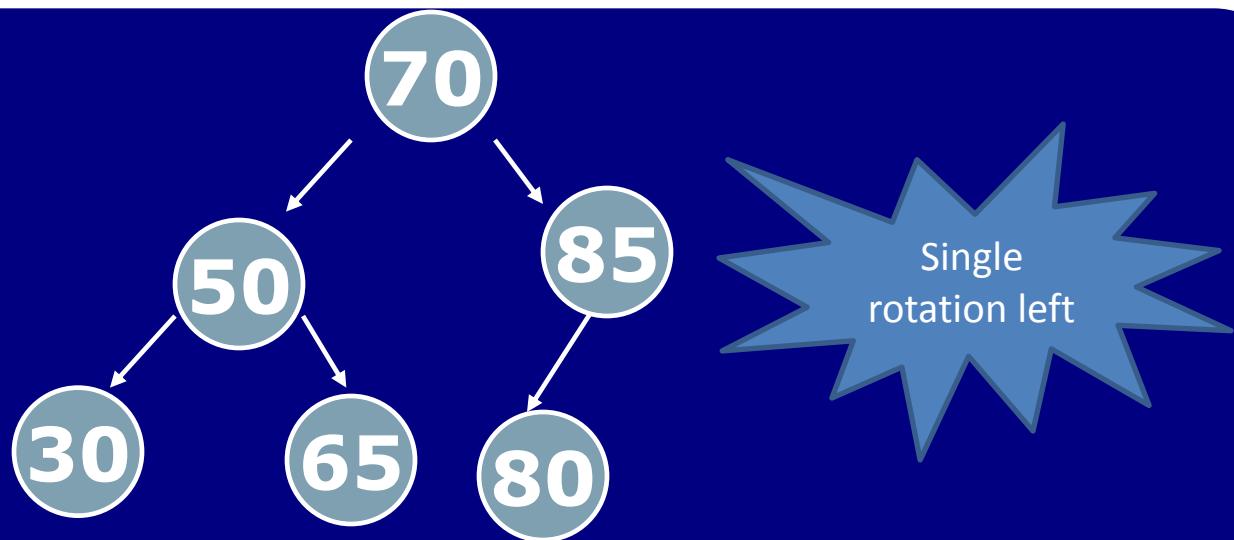
# Delete 60



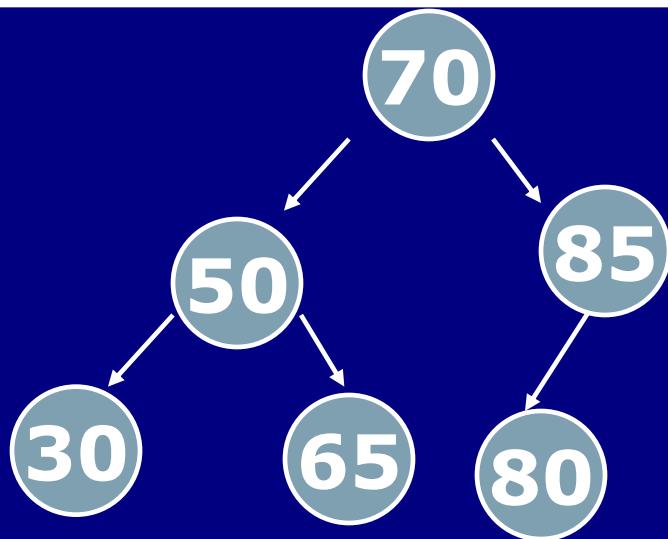
# Delete 60



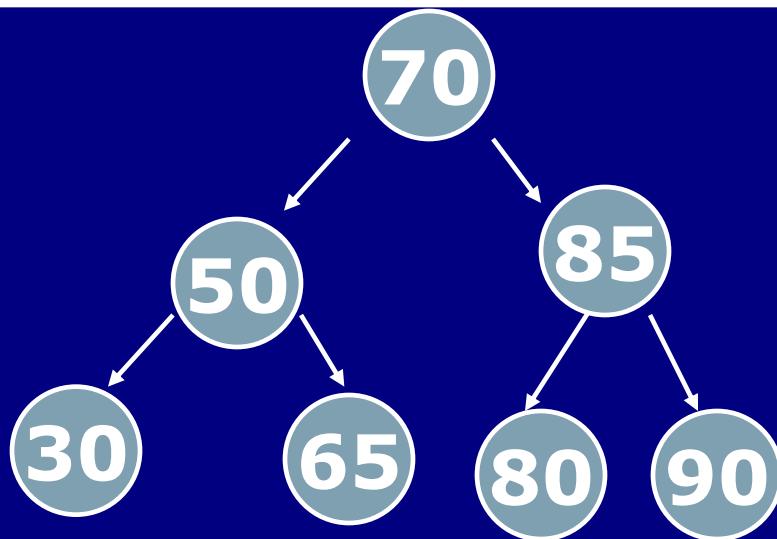
# Delete 60



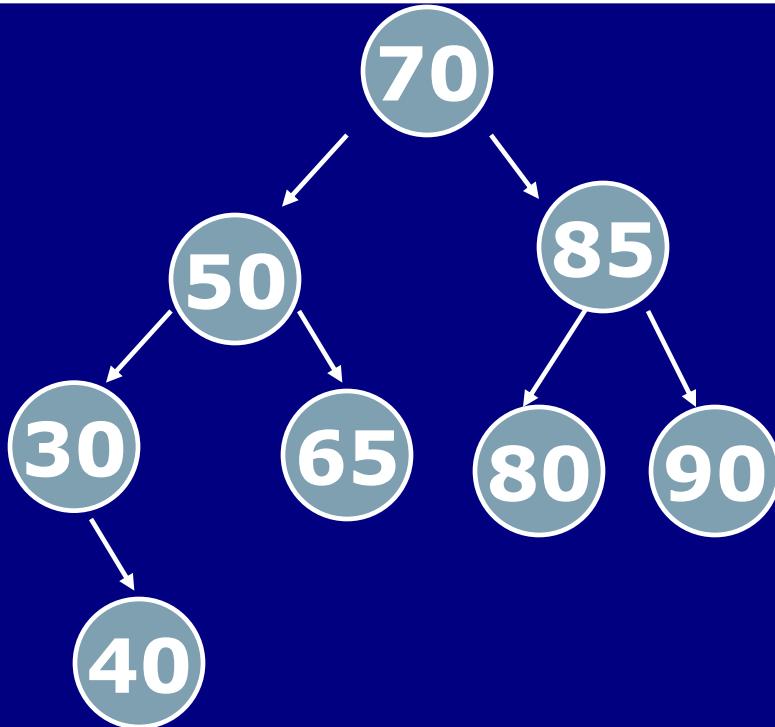
Insert 90, 40, 5, 55



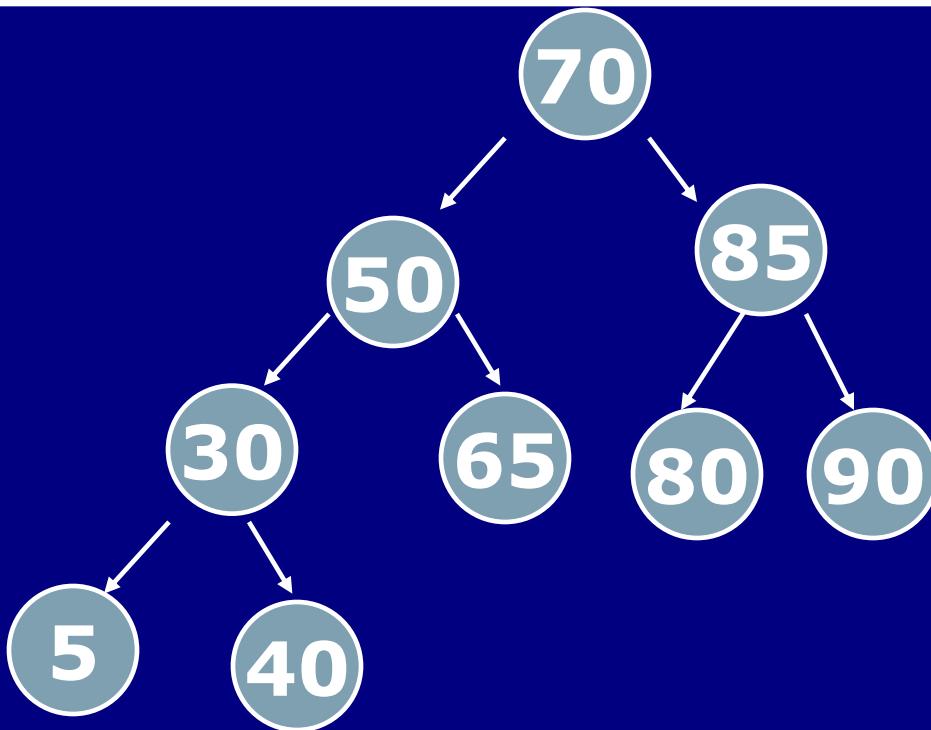
Insert 90, 40, 5, 55



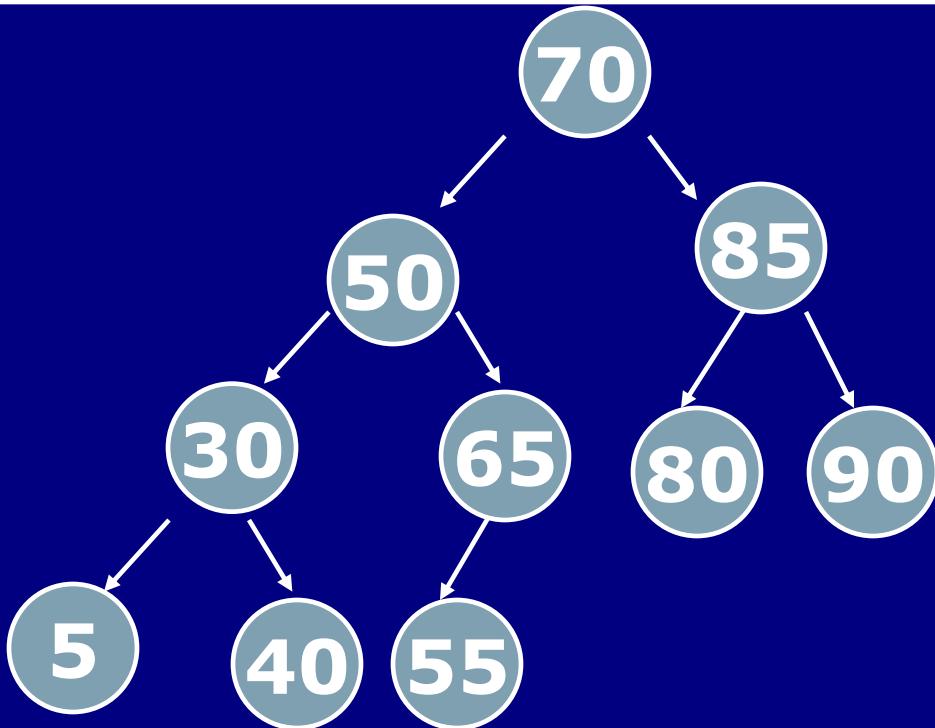
Insert 40, 5, 55



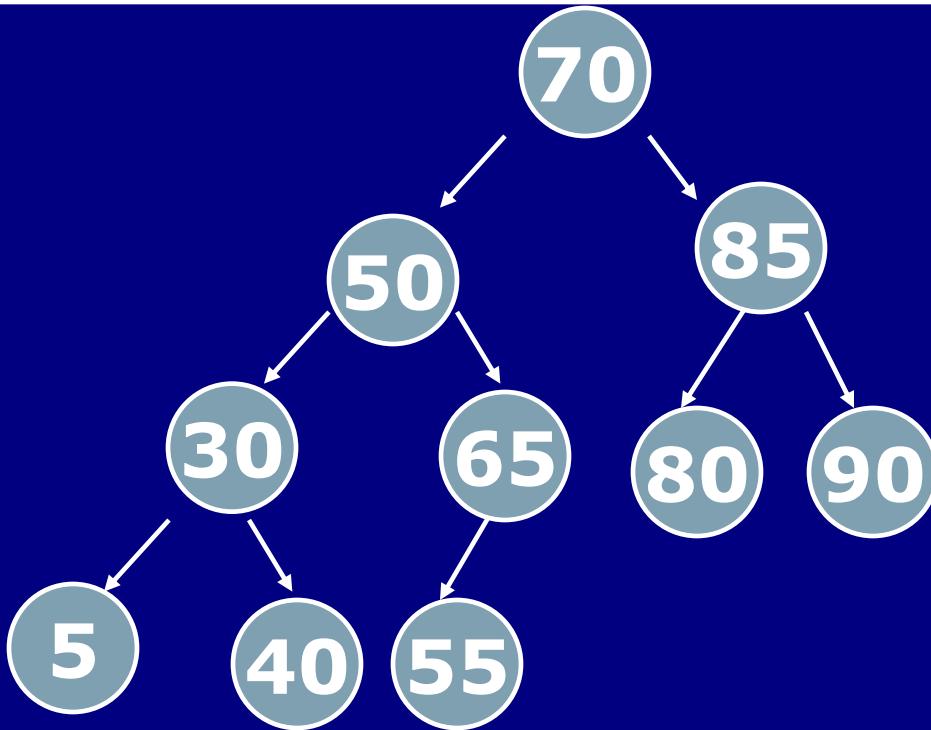
Insert 5, 55



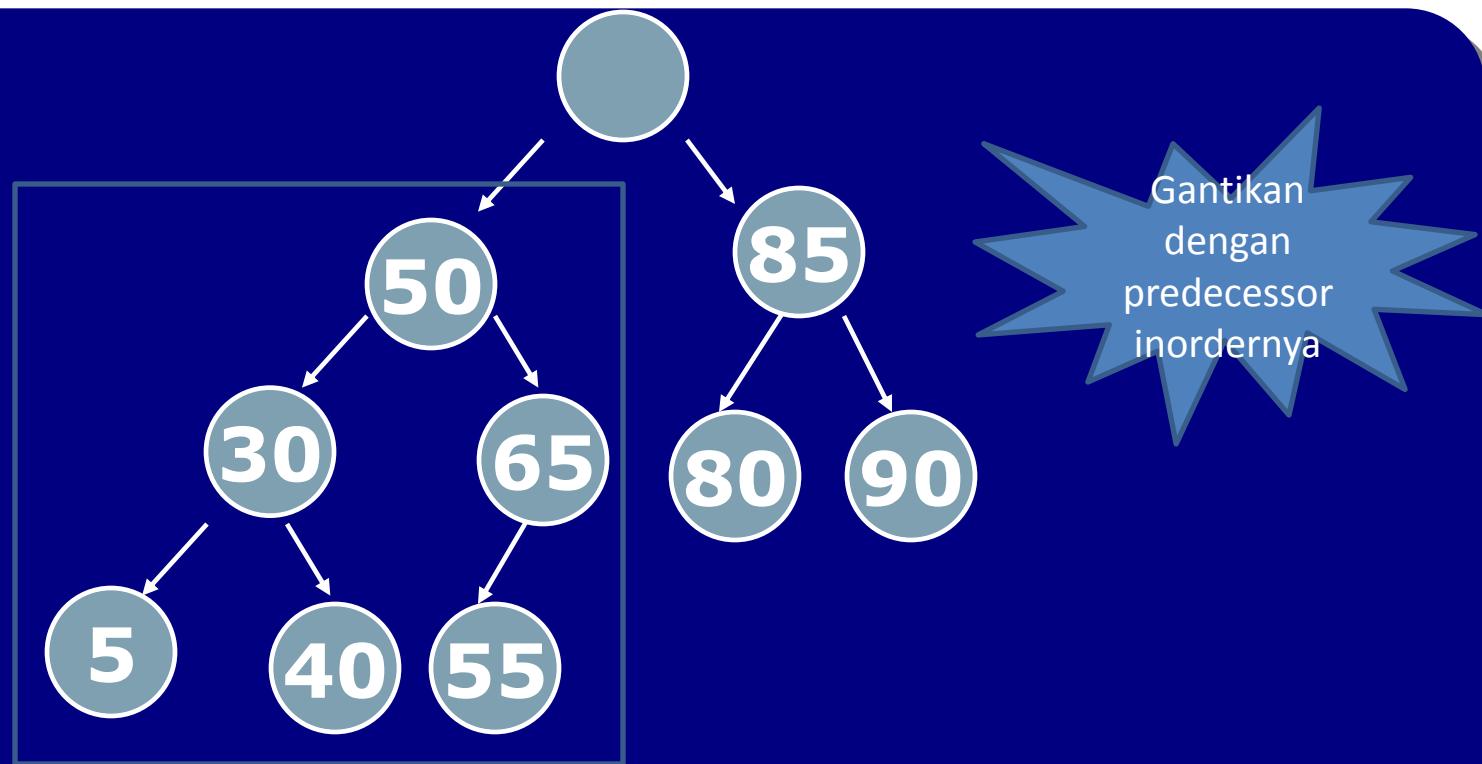
# Insert 55



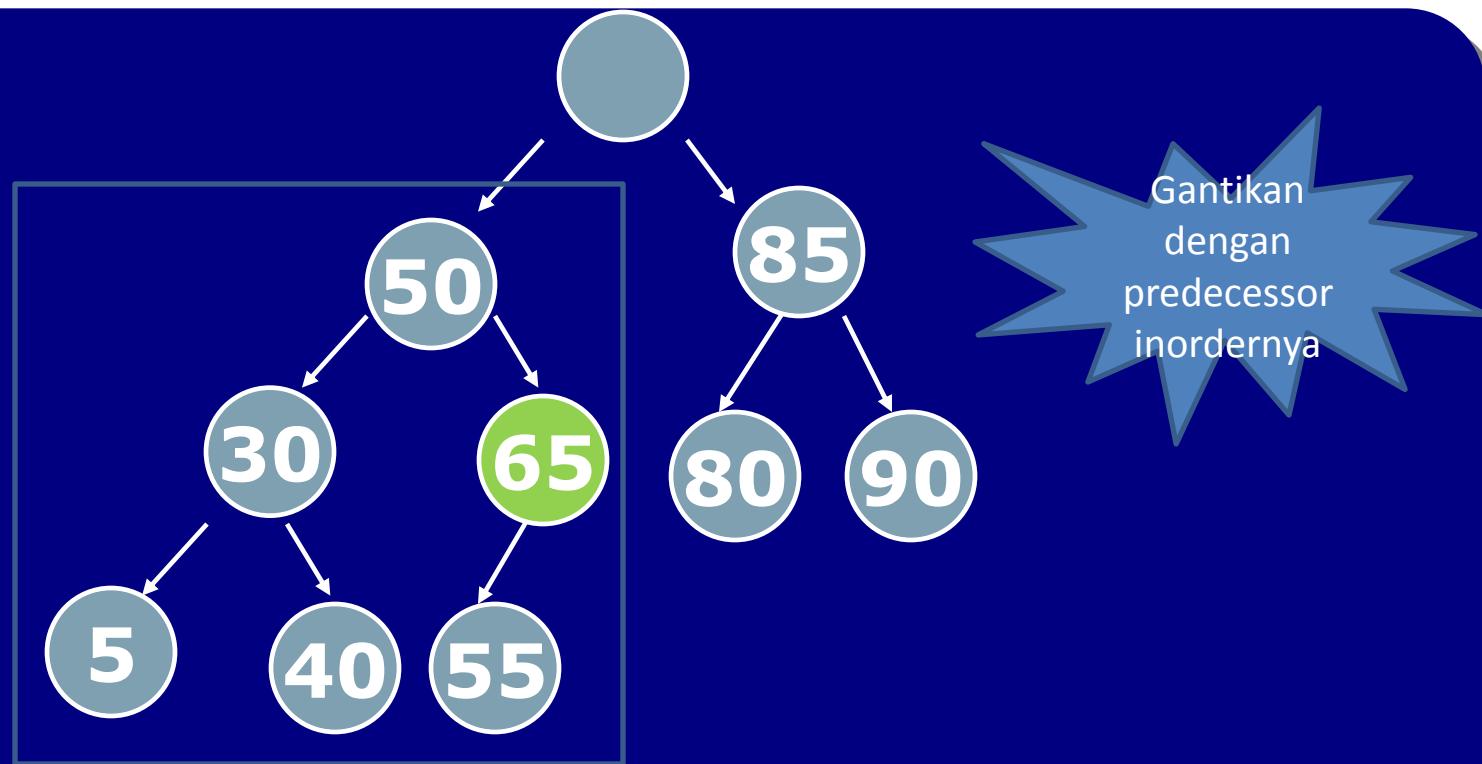
# Delete 70



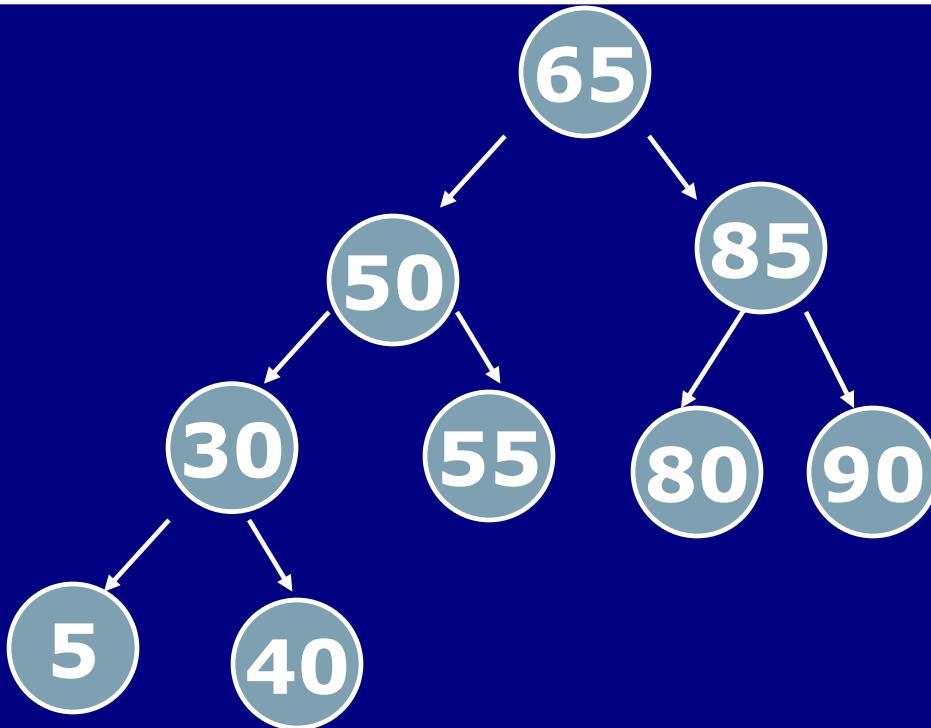
# Delete 70



# Delete 70



# Delete 70



# IKI10400 • Struktur Data & Algoritma: B-Tree

**Fakultas Ilmu Komputer • Universitas Indonesia**

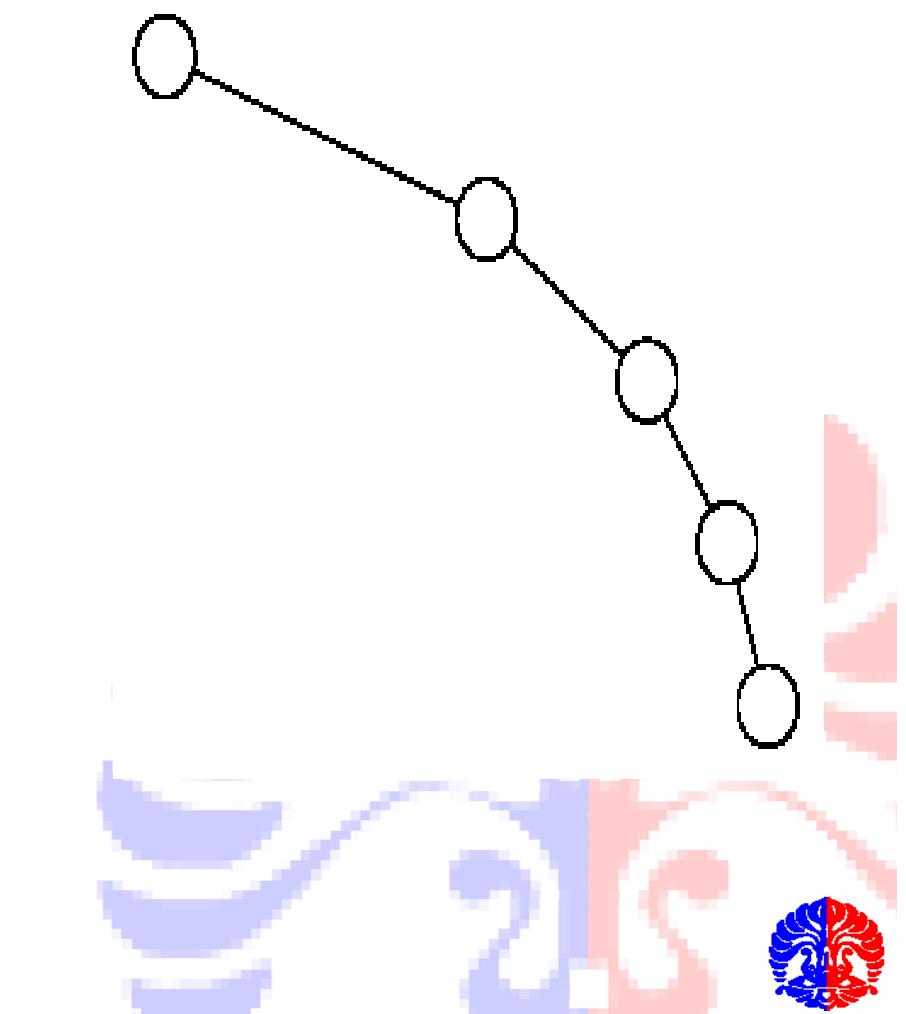
*Slide acknowledgments:*

Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung, Tisha Melia



# Motivasi B Trees

- The problem with Binary Trees is balance, the tree can easily deteriorate to a linked list. Consequently, the reduced search times are lost, this problem is overcome in B-Trees.
  - B stands for Balanced, where all the leaves are the same distance from the root. B-Trees guarantee a predictable efficiency.
- There are several varieties of B-Trees, most applications use the B+Tree.

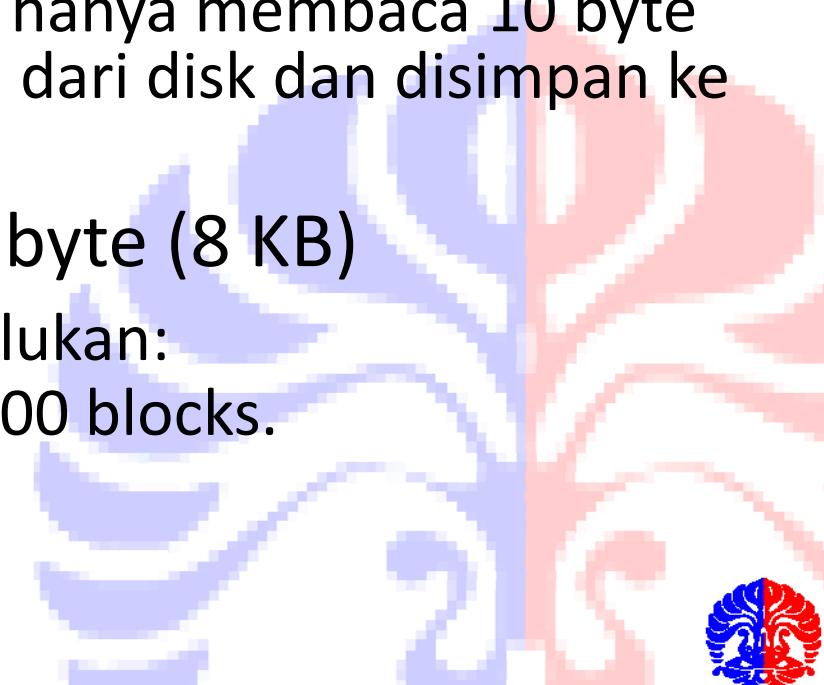


# Motivasi

- Perhatikan kasus berikut ini:
  - Kita harus membuat program basisdata untuk menyimpan data di yellow pages daerah Jakarta, misalnya ada 2.000.000 data.
  - Setiap entry terdapat nama, alamat, nomor telepon, dll. Asumsi setiap entry disimpan dalam sebuah record yang besarnya 512 byte.
  - Total file size =  $2,000,000 * 512 \text{ byte} = 1 \text{ GB}$ .
    - terlalu besar untuk disimpan dalam memory (primary storage)
    - perlu disimpan di disk (secondary storage)

# Motivasi

- Jika kita menggunakan disk untuk penyimpanan, kita harus menggunakan struktur blok pada disk untuk menyimpan basis data tsb.
  - Secondary storage dibagi menjadi blok-blok yang ukurannya sama. Umumnya 512 byte, 2 KB, 4 KB, 8 KB.
  - Block adalah satuan unit transfer antar disk dengan memory. Walaupun program hanya membaca 10 byte dari disk, 1 block akan dibaca dari disk dan disimpan ke memory.
- Misalnya 1 disk block 8.192 byte (8 KB)
  - Maka jumlah blok yang diperlukan:  
 $1 \text{ GB} / 8 \text{ KB per block} = 125,000 \text{ blocks.}$
  - Setiap blok menyimpan:  
 $8,192 / 512 = 16 \text{ records.}$



# Motivasi

- Karena keterbatasan mekanik, sebuah akses ke harddisk (storage) membutuhkan waktu yang relatif sangat lama.
  - Akses ke disk diperkirakan 10,000 kali lebih lambat dari pada akses ke main memory.
  - Sebuah akses ke disk dapat disetarakan dengan 200,000 buah instruksi.
- Dengan demikian jumlah akses ke disk akan mendominasi running time.
- Kita membutuhkan:  
*Sebuah teknik pencarian “multiway search tree” yang dapat meminimalkan jumlah akses ke disk.*



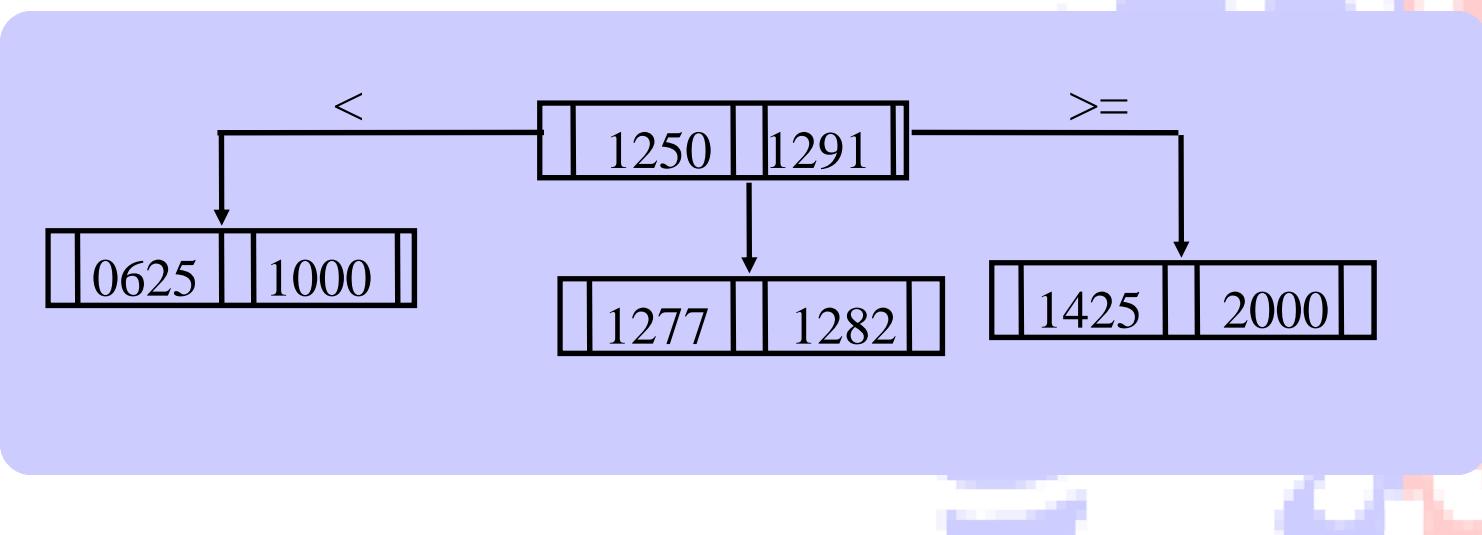
# B-Tree

- B-tree banyak digunakan untuk external data structure.
  - Setiap node berukuran sesuai dengan ukuran block pada disk, misalnya 1 block = 8 KB.
  - Tujuannya: meminimalkan jumlah block transfer.
- Ada beberapa variasi dari B-Trees, salah satu contoh yang banyak digunakan adalah B+Tree.



# B Tree

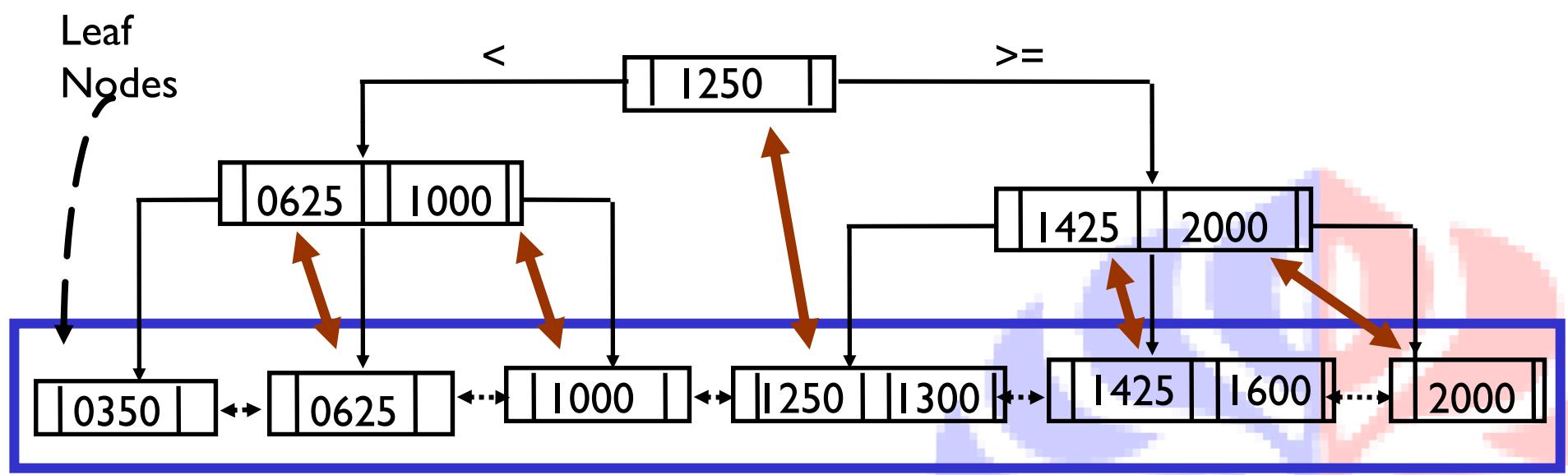
- B Tree dengan *degree m* memiliki karakteristik sebagai berikut:
  - Setiap non-leaf (internal) nodes (kecuali root) jumlah anaknya (yang tidak null) antara  $\lceil m/2 \rceil$  dan *m*.
  - Sebuah non-leaf (internal) node yang memiliki *m* cabang memiliki sejumlah *m-1* keys.
  - Setiap leaves berada pada *level* yang sama, (dengan kata lain, memiliki *depth* yang sama dari *root*).



# B+Tree

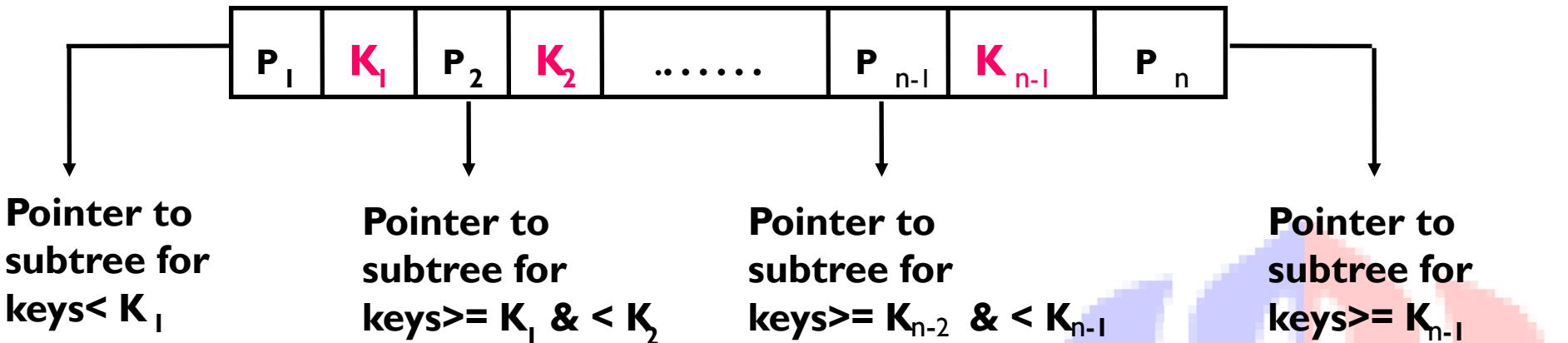
- B+Tree adalah variant dari B-tree, dengan aturan:
  - semua key value sebagai reference terhadap data disimpan dalam leaf.
  - disertakan suatu pointer tambahan untuk menghubungkan setiap leaf node tersebut sebagai suatu linear linked-list.
    - Struktur ini memungkinkan akses sikuensial data dalam B-tree tanpa harus turun-naik pada struktur hirarkisnya.
  - node internal digunakan sebagai ‘indeks’ (dummy key).
  - beberapa *key value* dapat muncul dua kali di dalam tree (sekali pada leaf sebagai reference thd data kedua sebagai indeks pada internal node).

# B+Tree

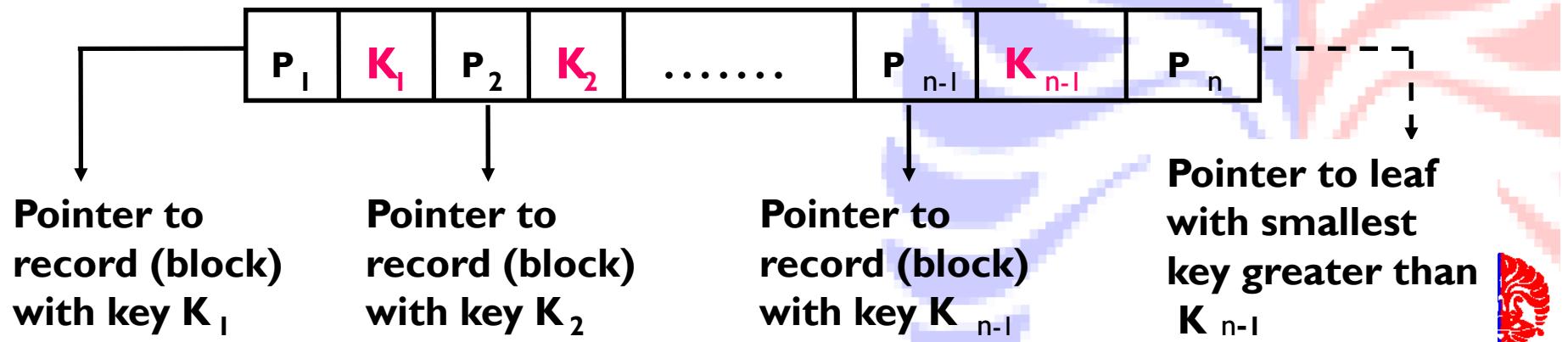


# Struktur: B+Tree Node

## A high level node (internal node)



## A leaf node (Every key value appears in a leaf node)

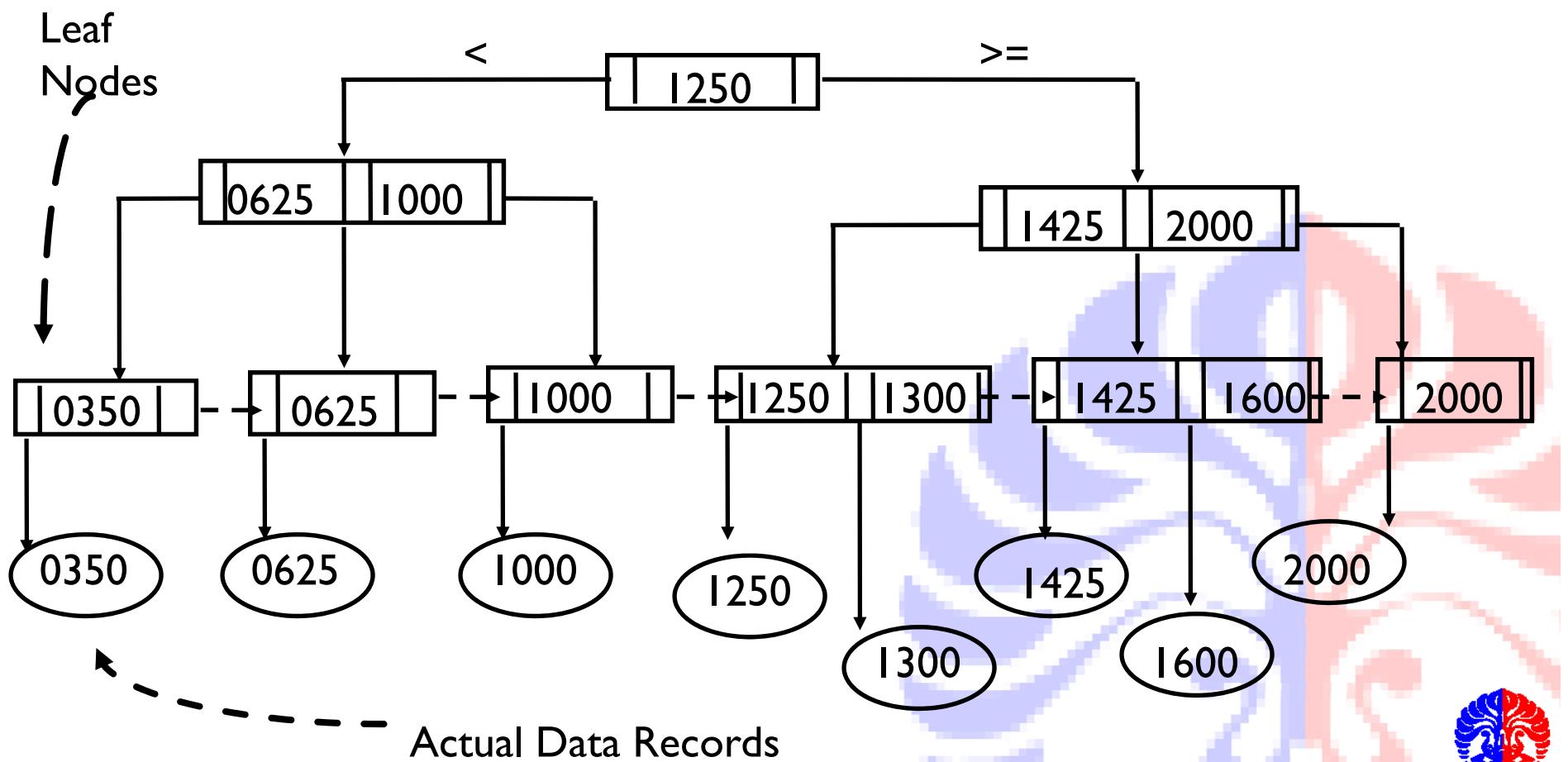


# Struktur: B+Tree Node

- Perhatikan bahwa:
  - jumlah pointer/children pada leaf nodes == jumlah key
    - karena pointer terakhir menunjuk ke leaf node berikutnya
  - jumlah pointer/children pada internal nodes == jumlah key + 1



# Contoh sebuah B+Tree



# Proses pada B+Trees

- Mencari records dengan *search-key value* :  $k$ .
  1. Mulai dari root.
    - a. Periksa node tersebut, dan tentukan nilai key terkecil pada node tersebut yang lebih besar dari  $k$ .
    - b. Jika key tersebut ditemukan, ada  $K_i$ , key terkecil pada node yang memenuhi syarat:  $K_i > k$ , maka ikuti  $P_i$  (rekursif terhadap node yang ditunjukkan oleh  $P_i$ )
    - c. Jika tidak ditemukan dan  $k \geq K_{m-1}$ , serta ada  $m$  pointers pada node. Maka ikuti  $P_m$  (rekursif terhadap node yang ditunjukkan oleh  $P_i$ )
  - Bila node yang ditunjukkan oleh pointer tersebut di atas internal node (non-leaf node), ulangi prosedure a-c.
  - Bila mencapai sebuah *leaf node*. Jika ada  $i$  sehingga nilai key  $K_i = k$ , maka ikuti pointer  $P_i$  untuk mendapatkan record (atau *bucket*) yang diinginkan. Jika tidak, maka tidak ada data dengan nilai key  $k$ .

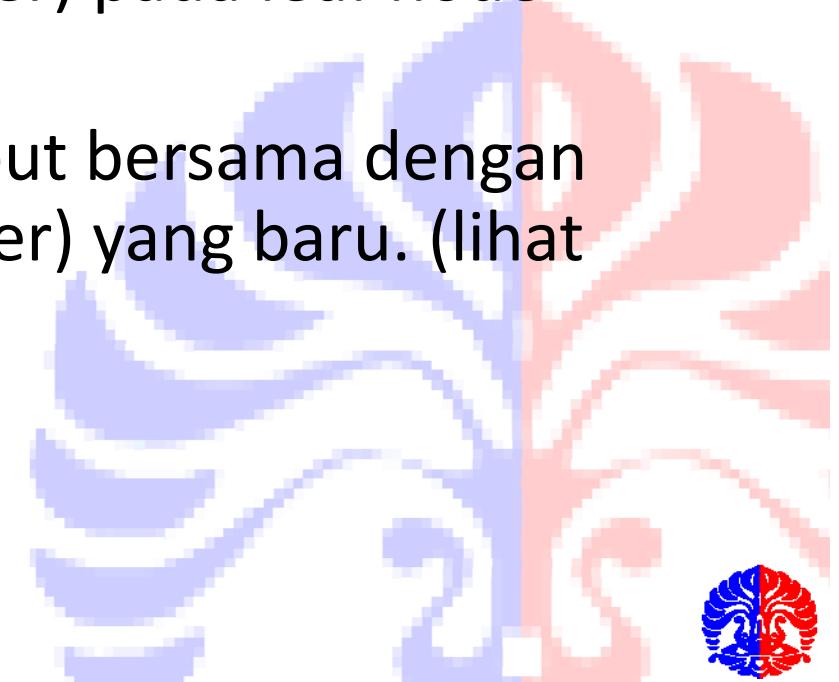


# Proses pada B+Trees: *Range Query*

- Mencari seluruh record yang berada diantara  $k$  dan  $l$  (range query).
  - Mencari record dengan search-key value =  $k$ .
  - selama nilai search-key value selanjutnya yang ditunjukkan oleh pointer lebih kecil dari  $l$ , ikuti pointer untuk mendapatkan records yang diinginkan
    - jika search-key yang ditemukan adalah search-key yang terakhir dalam node, ikuti pointer yang terakhir ( $P_n$ ) untuk menuju leaf node selanjutnya.
- Bandingkan dengan proses yang sama pada binary search tree

# Insertion on B+Trees

- Cari posisi leaf node yang sesuai agar nilai search-key yang baru dapat diletakkan secara terurut.
  - Jika masih ada tempat pada leaf node, tambahkan pasangan (key-value, pointer) pada leaf node secara terurut
  - Bila tidak, split node tersebut bersama dengan pasangan (key-value, pointer) yang baru. (lihat slide selanjutnya)



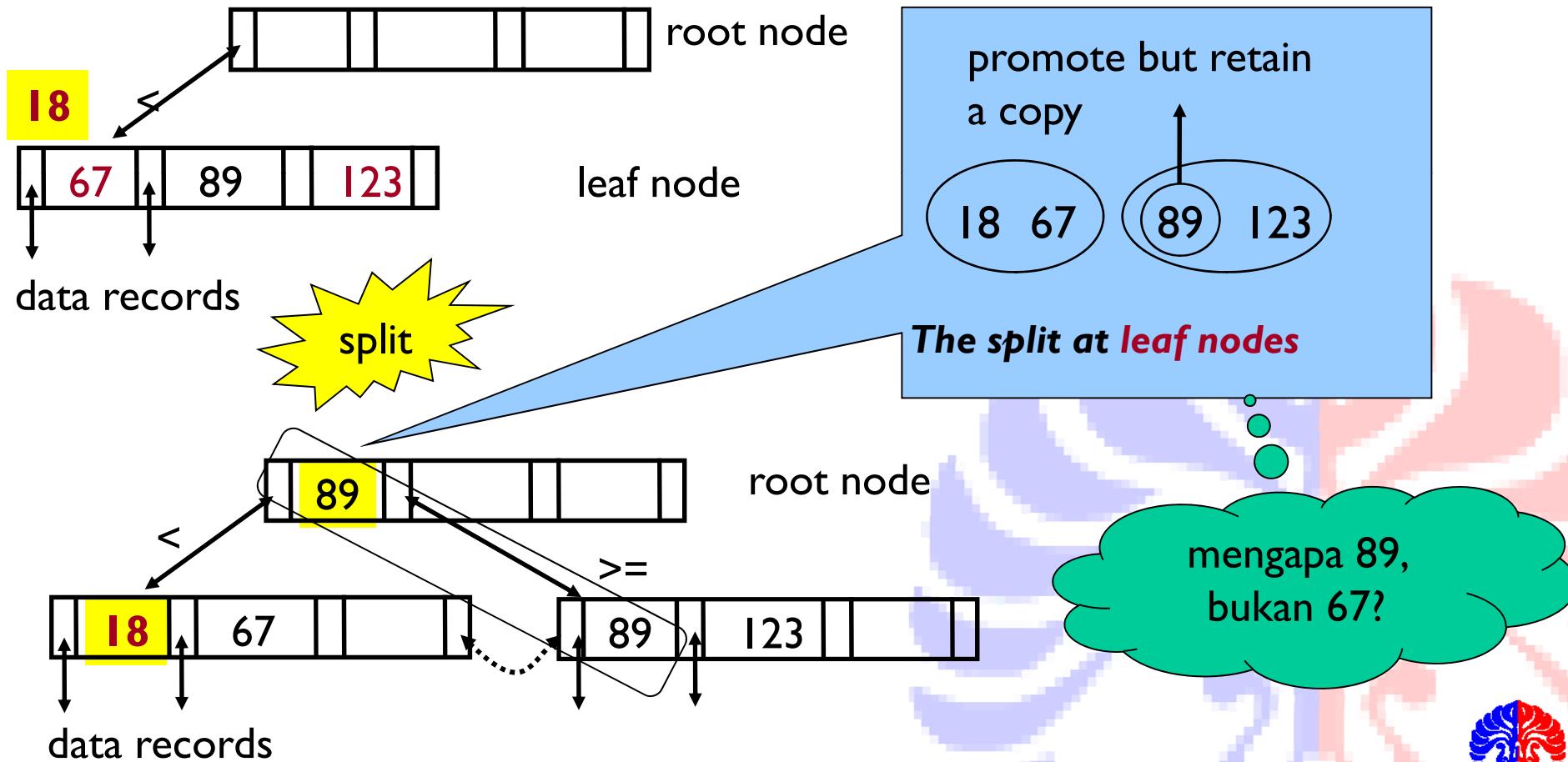
# Insertion on B+Trees

- Proses Splitting pada sebuah node:
  - ambil pasangan (search-key value, pointer) **termasuk yang baru ditambahkan**, letakkan search-key  $\lfloor \frac{m}{2} \rfloor$  yang pertama pada node awal, dan pindahkan sisanya pada sebuah node baru.
  - ketika melakukan splitting **pada sebuah leaf**, promosikan middle/median key dari node yang akan di split kepada parent node, **tanpa menghapus pada leaf** tersebut (meng-copy).
  - ketika melakukan splitting **pada internal node**, promosikan the middle/median key dari node yang akan displit kepada parent node, dan **hapus pada node** sebelumnya (tidak membuat copy).
  - Jika hasil promosi membuat parent menjadi full, lakukan proses splitting serupa secara rekursif.

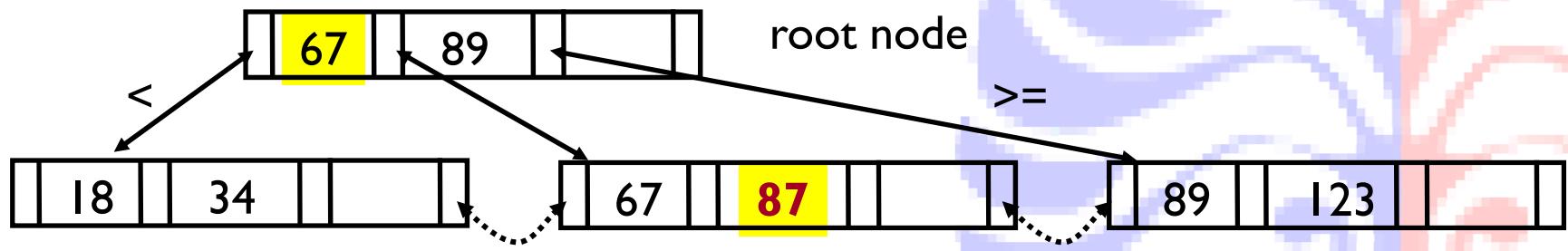
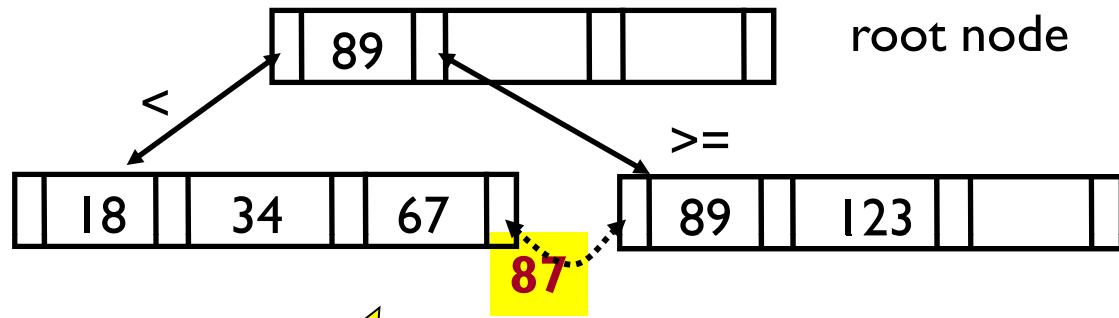


# Building a B+Tree (m=4)

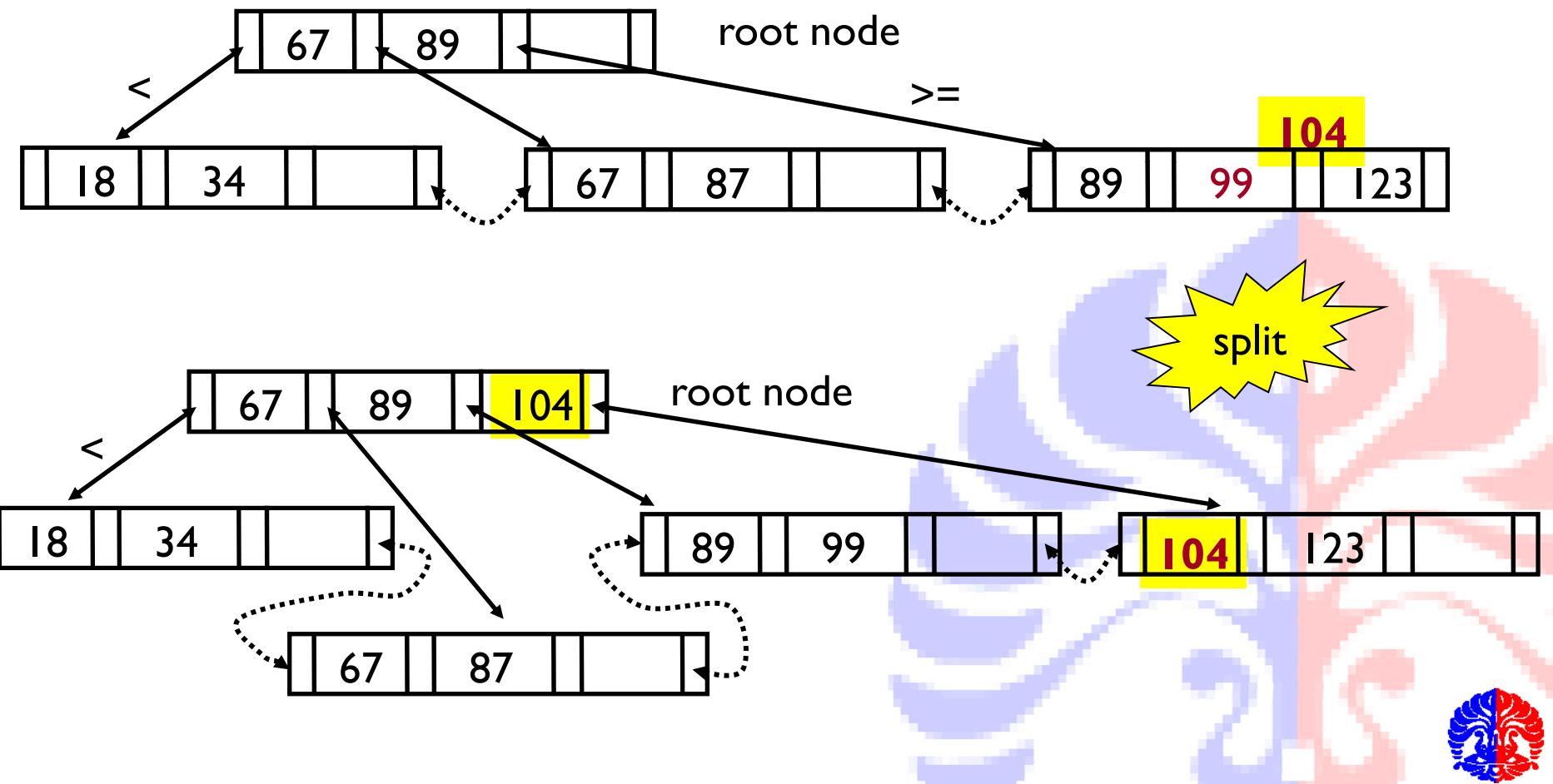
67, 123, 89, **18**, 34, 87, 99, 104, 36, 55, 78, 9



67, 123, 89, 18, 34, **87**, 99, 104, 36, 55, 78, 9

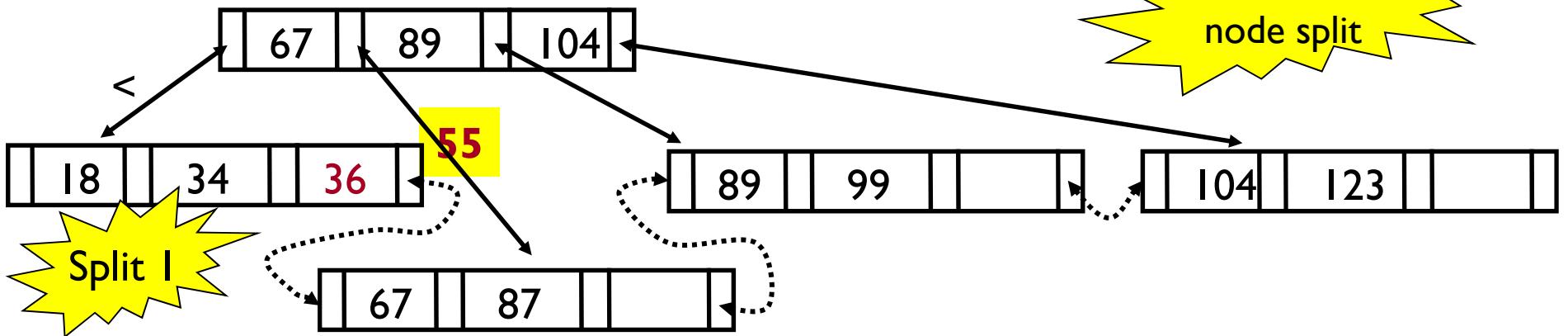


67, 123, 89, 18, 34, 87, 99, **104**, 36, 55, 78, 9

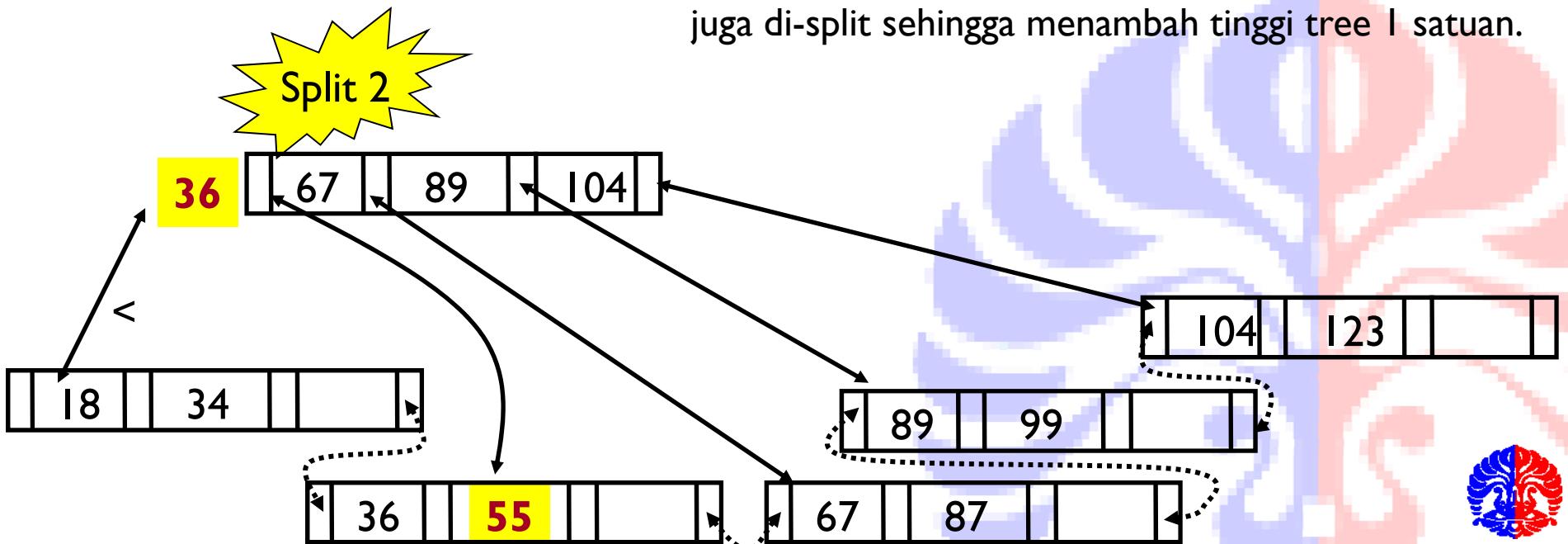


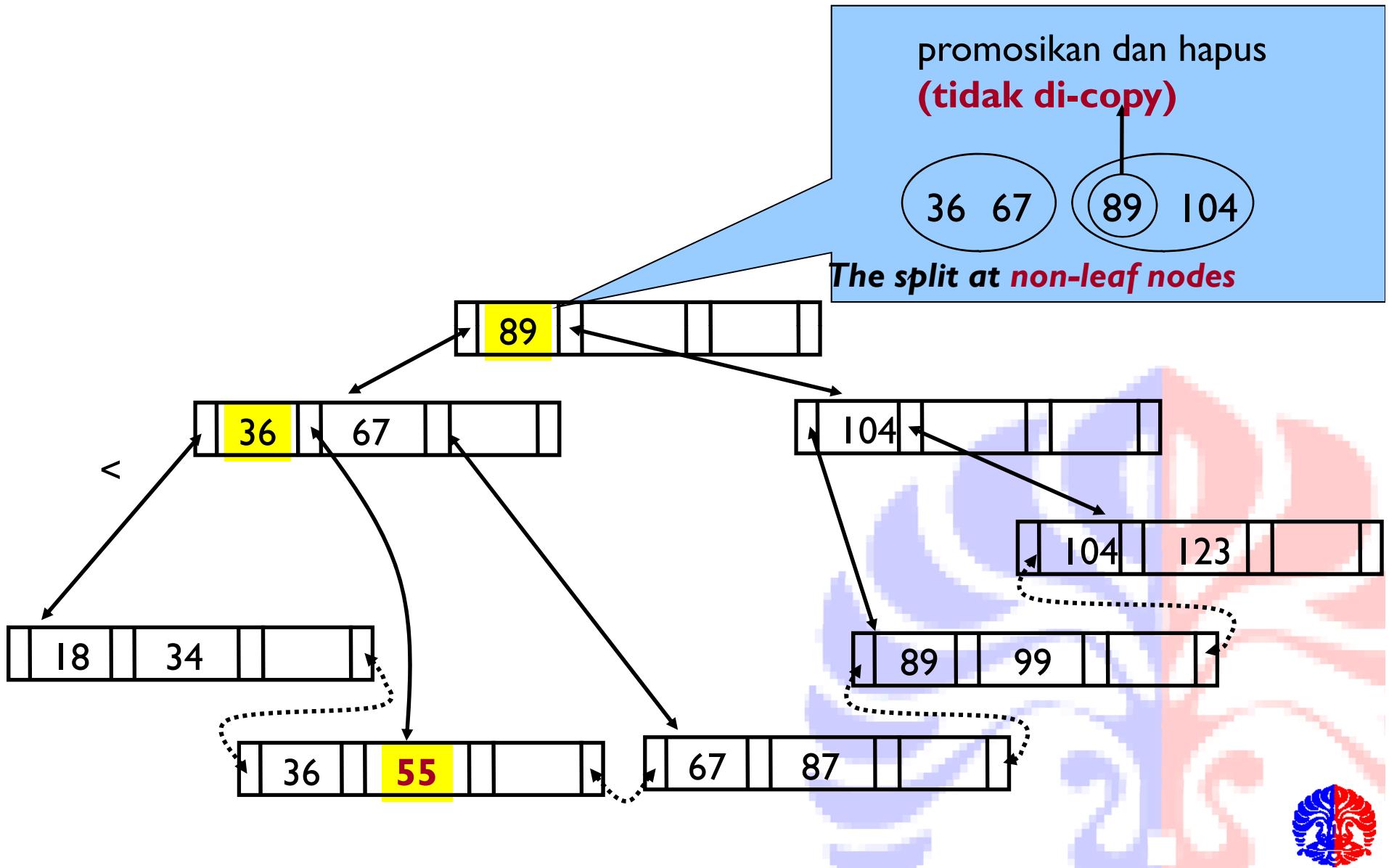
67, 123, 89, 18, 34, 87, 99, 104, 36, **55**, 78, 9

double  
node split



Proses splitting diteruskan ke atas hingga node tersebut tidak full. Pada worst case, root node bisa juga di-split sehingga menambah tinggi tree 1 satuan.





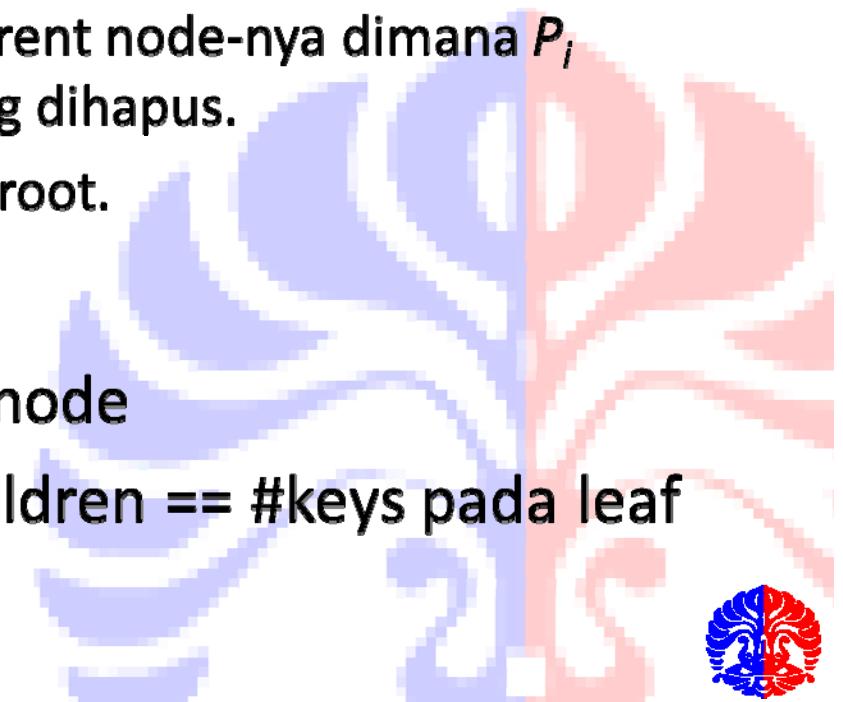
# Observasi mengenai B+Trees

- B+Tree umumnya memiliki jumlah levels yang rendah (logarithmic pada jumlah data), sehingga proses pencarian dapat dilakukan dengan effisien.
  - Saat memproses sebuah query, sebuah jalur (path) dijalani pada tree dari root hingga leaf node.
  - jika terdapat sejumlah  $K$  search-key pada sebuah file, maka jalur pencarian tidak akan lebih besar dari  $\lceil \log_{m/2}(K) \rceil$ ,  $m$  adalah degree B+ Tree.
    - pertanyaan: Mengapa “ $\log_{m/2}$ ”, bukan “ $\log_m$ ” ?



# Deletion on B+Trees

- hapus pasangan (search-key value, pointer) dari leaf node
- jika node ternyata memiliki pasangan yang terlalu sedikit, dan jika jumlah pasangan pada node dan sibling-nya **dapat digabungkan** pada sebuah node, maka
  - gabungkan kedua node tersebut
  - hapus pasangan  $(K_{i-1}, P_i)$ , pada parent node-nya dimana  $P_i$  adalah pointer terhadap node yang dihapus.
  - Lakukan secara recursively hingga root.
- minimum requirement:
  - $\lceil \frac{m}{2} \rceil$  children pada internal node
  - $\lceil \frac{m}{2} \rceil$  children pada leaf, #children == #keys pada leaf

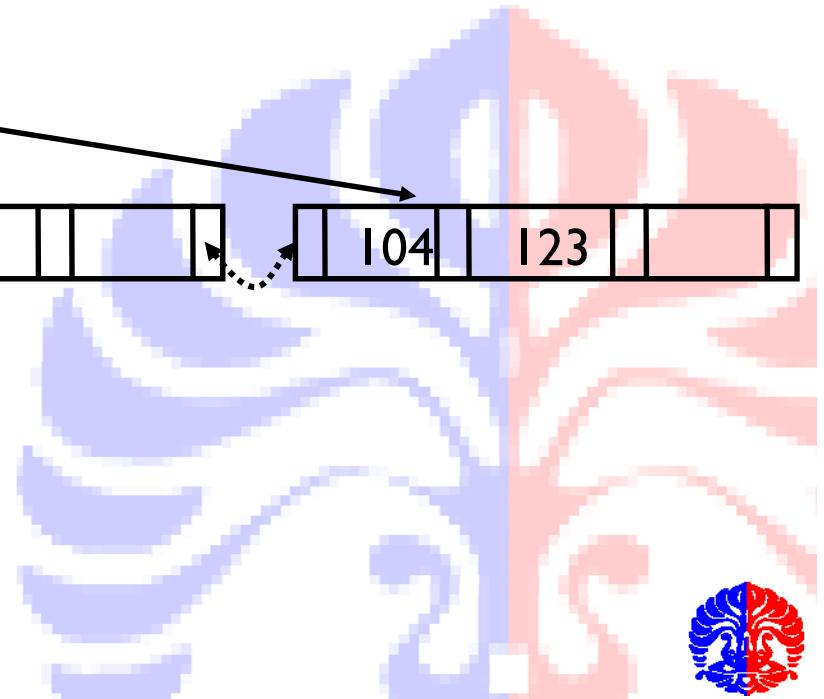
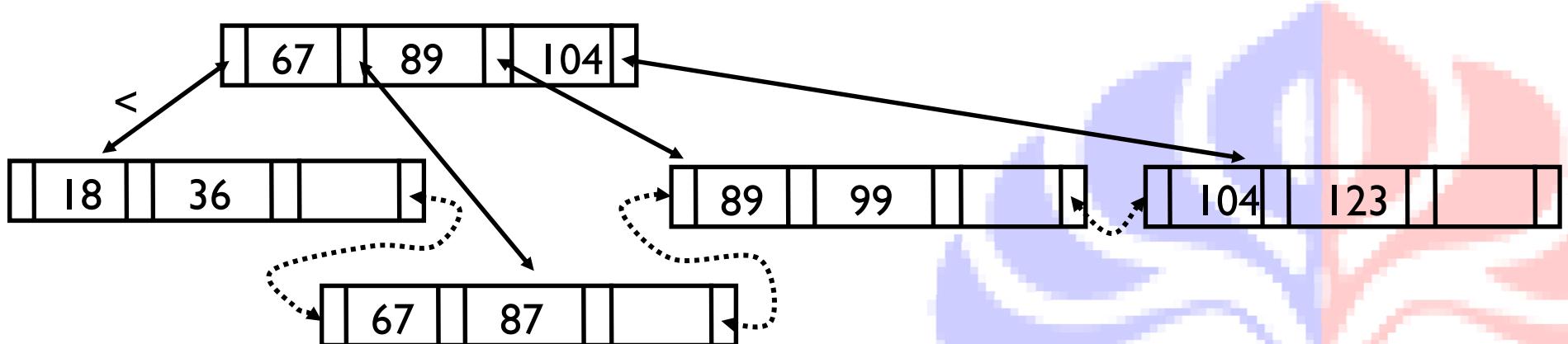
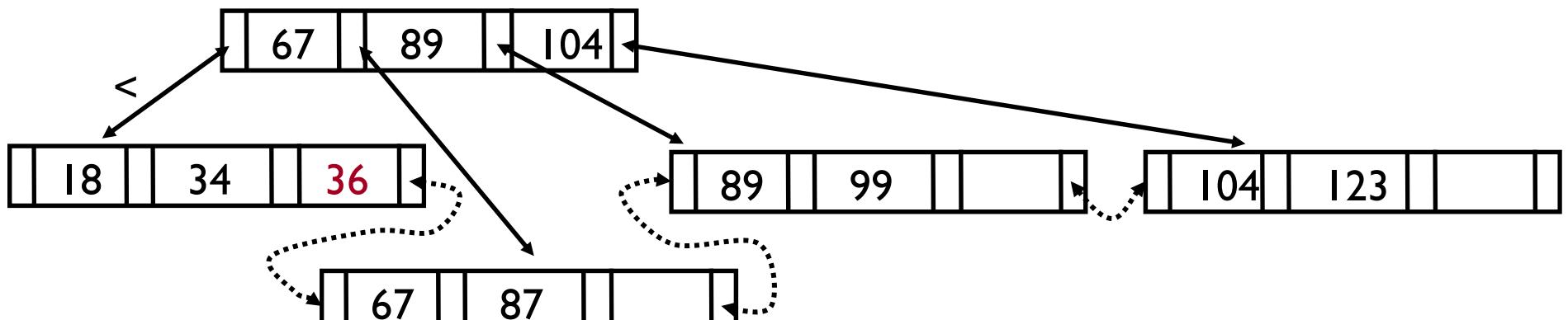


# Deletion on B+Trees

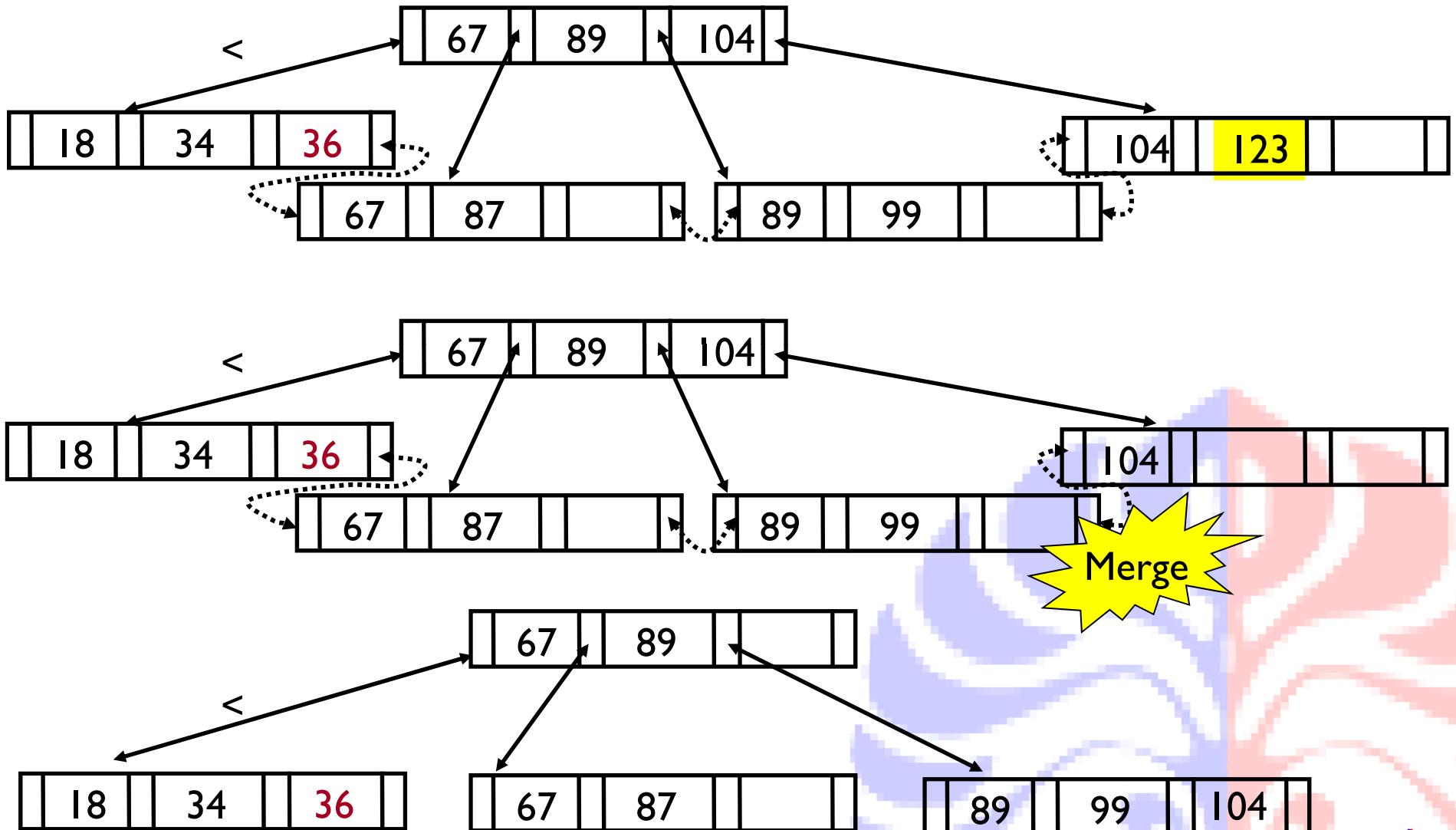
- jika jumlah pasangan pada node dan sibling-nya **tidak dapat** digabungkan pada sebuah node, maka
- re-distribusikan pointers diantara node tersebut dan sibling-nya sehingga mereka memiliki jumlah element yang lebih dari minimum.
  - Update search-key yang berkesesuaian pada parent .
  - sibling → memiliki parent yang sama
- proses ini dapat terus berlaku rekursif ke parent hingga ditemukan node yang memiliki pasangan sejumlah  $\lceil \frac{m}{2} \rceil$  atau lebih.



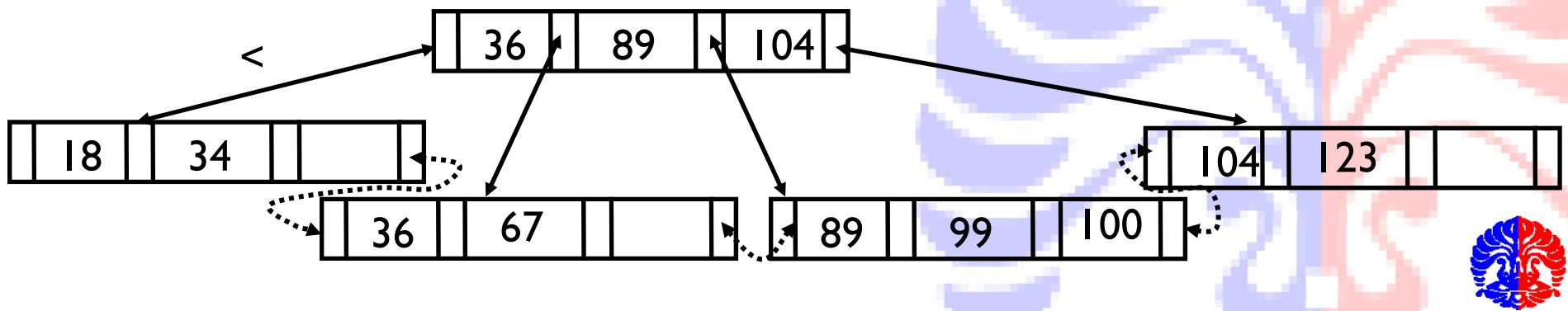
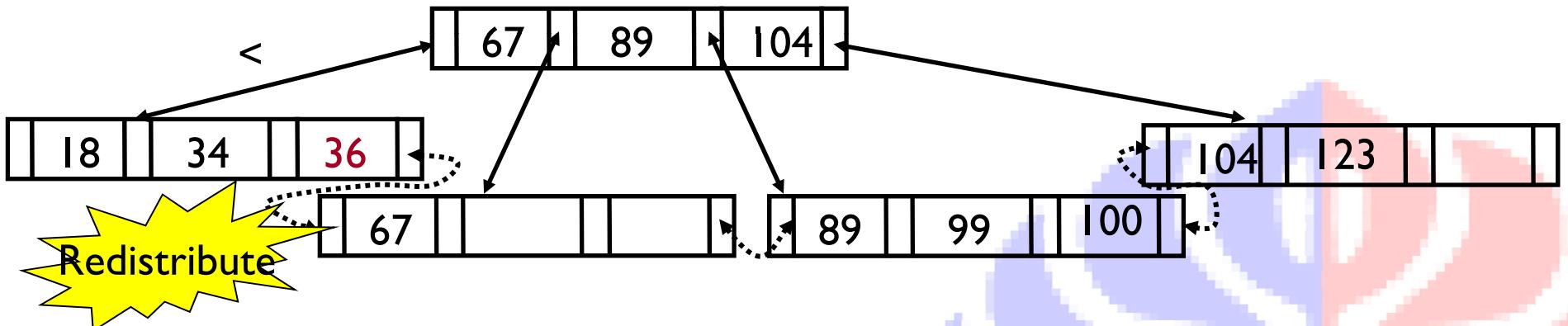
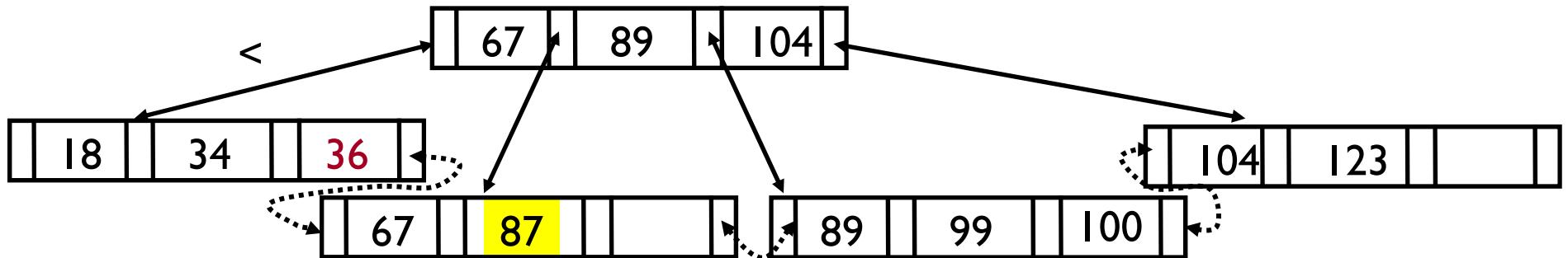
Hapus 34 → OK



## Hapus 123 → Merge



## Hapus 87 → Redistribute



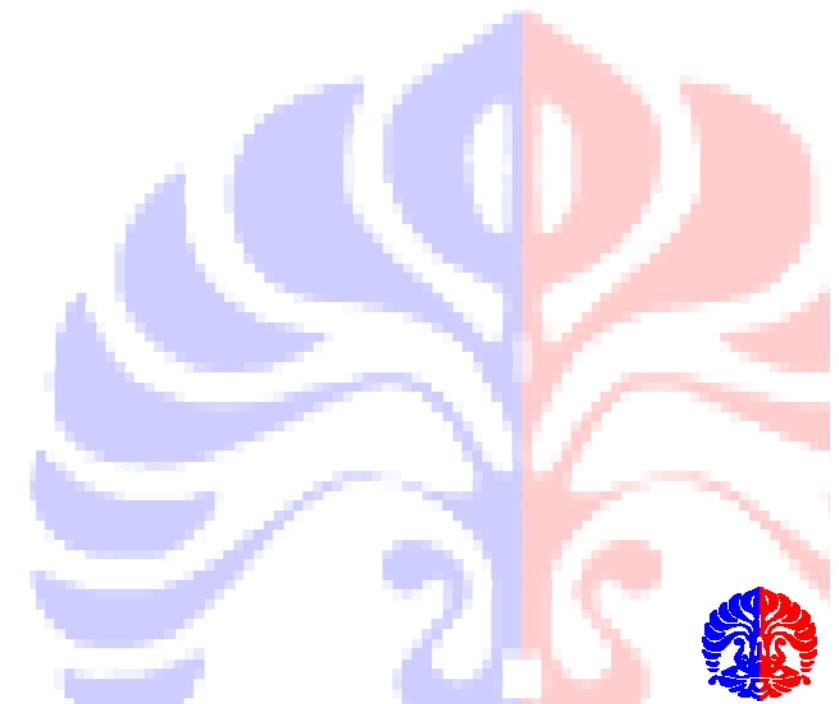
# Rangkuman

- B Tree biasanya digunakan sebagai struktur data external pada databases.
- B Tree dengan degree  $m$  memiliki aturan seperti berikut:
  - Setiap non-leaf (internal) nodes (kecuali root) jumlah anaknya (yang tidak null) antara  $\lceil m/2 \rceil$  dan  $m$ .
  - Sebuah non-leaf (internal) node yang memiliki  $m$  cabang memiliki sejumlah  $m-1$  keys.
  - Setiap leaves berada pada *level* yang sama, (dengan kata lain, memiliki *depth* yang sama dari *root*).
- B+Tree adalah variant dari B Tree dimana seluruh key terletak pada leaves.



# Tambahan informasi dan latihan

- applet simulasi B+Tree
  - <http://slady.net/java/bt/view.php?w=600&h=450>



# IKI10400 • Struktur Data & Algoritma: Binary Heap & Huffman Code

**Fakultas Ilmu Komputer • Universitas Indonesia**

*Slide acknowledgments:*

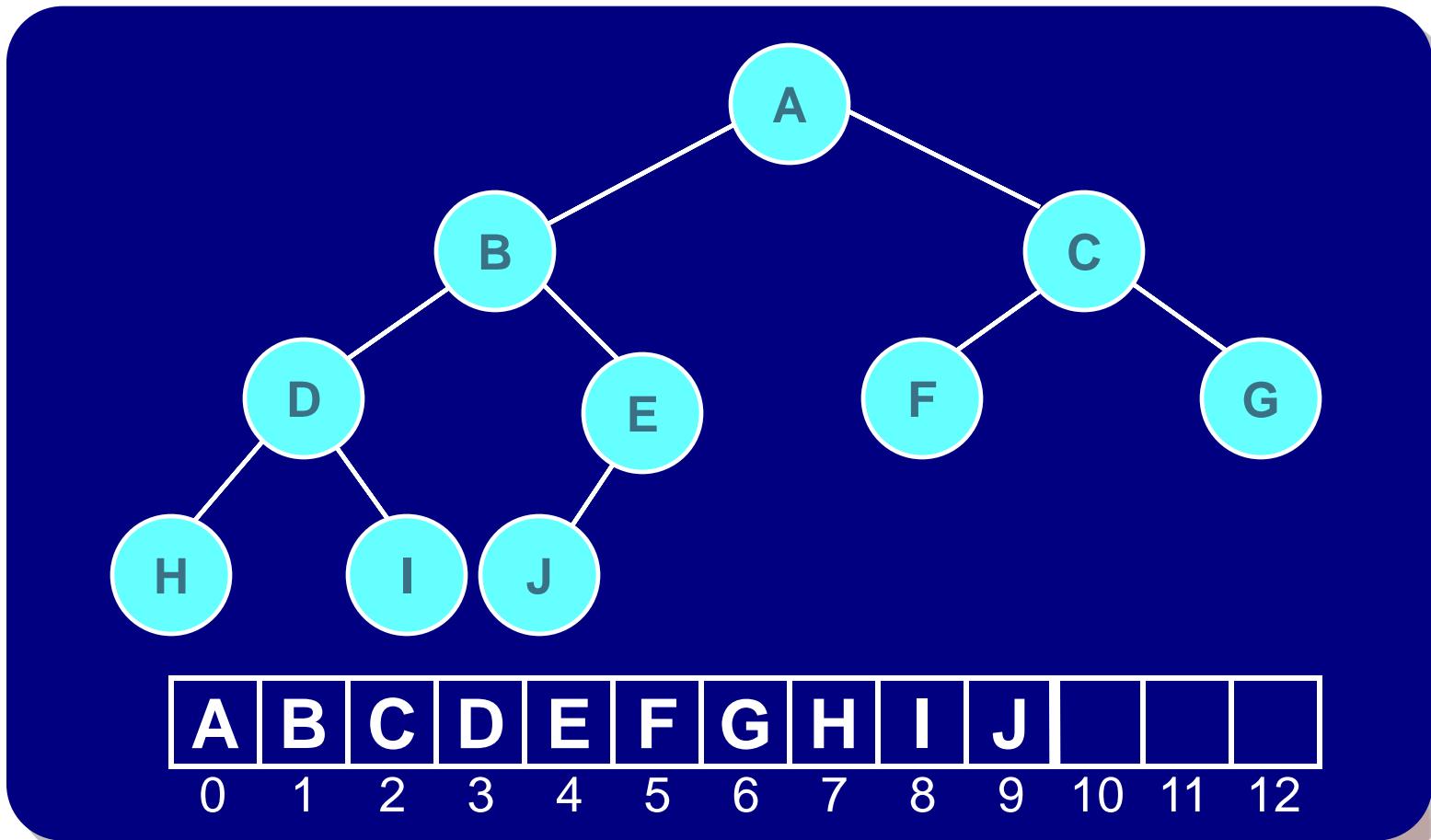
Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung



# BINARY HEAP

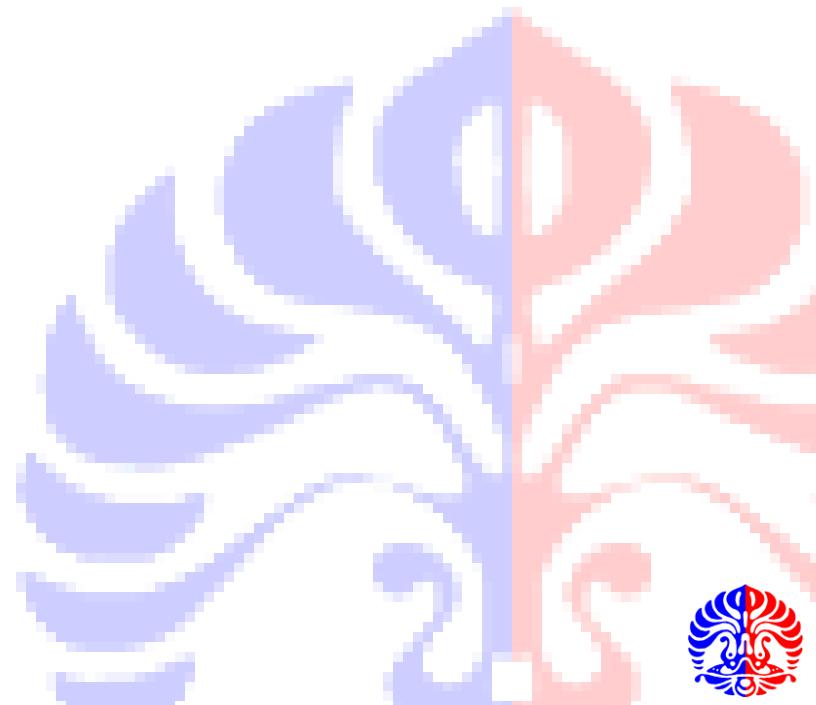
# Review

- Complete binary tree:
  - sebuah tree adalah terisi penuh (complete), kecuali pada level terbawah yang terisi dari kiri ke kanan.



# Priority Queue

- Sebuah queue dengan perbedaan aturan sebagai berikut:
  - operasi *enqueue* tidak selalu menambahkan elemen pada akhir *queue*, namun, meletakkannya sesuai urutan prioritas.



# Binary Heap

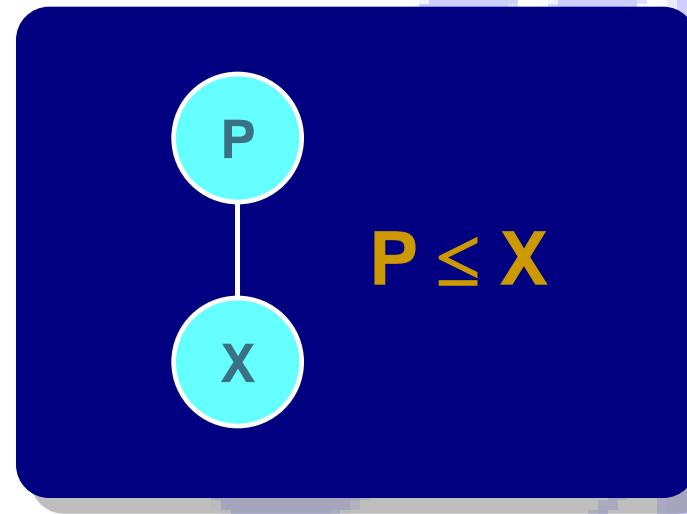
- Hanya diperbolehkan mengakses (membaca) item yang minimum.
- operasi dasar:
  - **menambahkan** item baru dengan kompleksitas waktu worst-case yang logaritmik.
  - **menghapus** item yang minimum dengan kompleksitas waktu worst-case yang logaritmik.
  - **mencari** item yang minimum dengan kompleksitas waktu konstan.



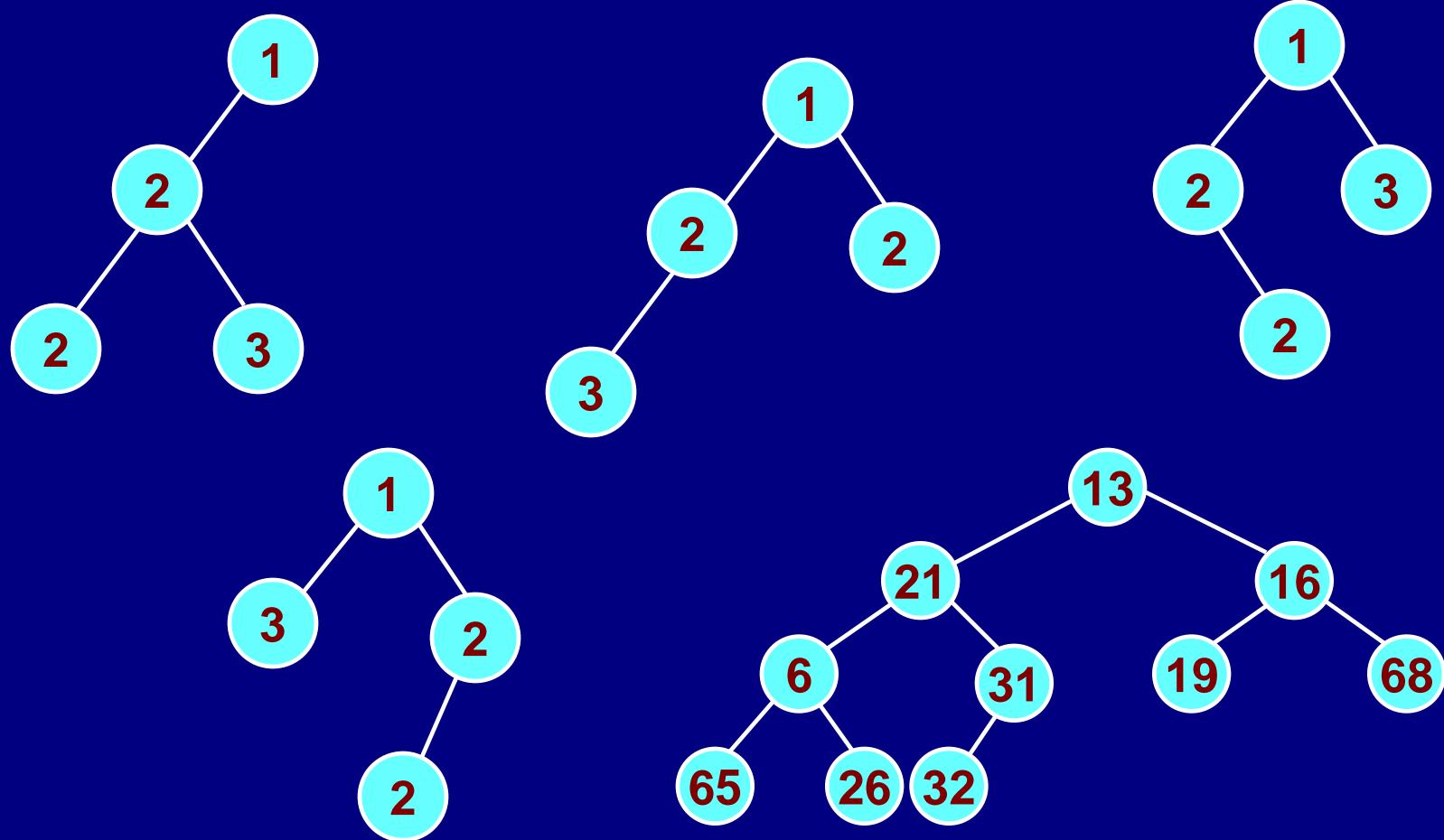
# Properties (Aturan)

- Structure Property
  - Data disimpan pada complete binary tree
    - ⇒ tree selalu balance.
    - ⇒ seluruh operasi dijamin  $O(\log n)$  pada worst case
    - ⇒ data disimpan menggunakan array atau java.util.Vector
- Ordering Property

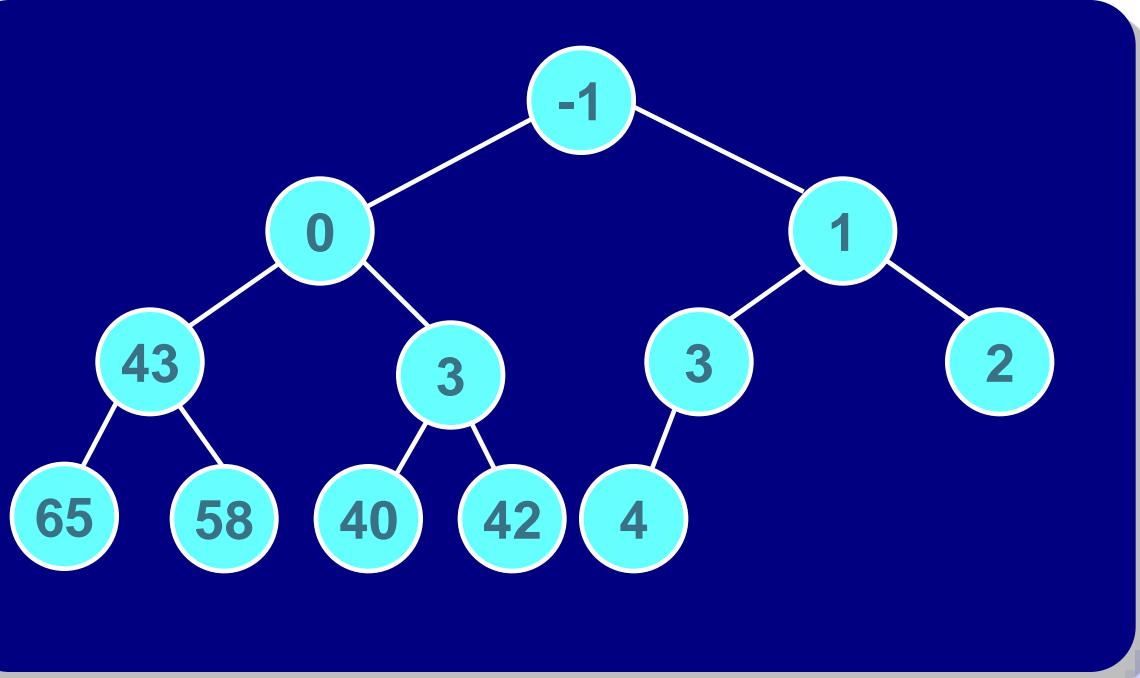
- Heap Order:
  - Parent  $\leq$  Child



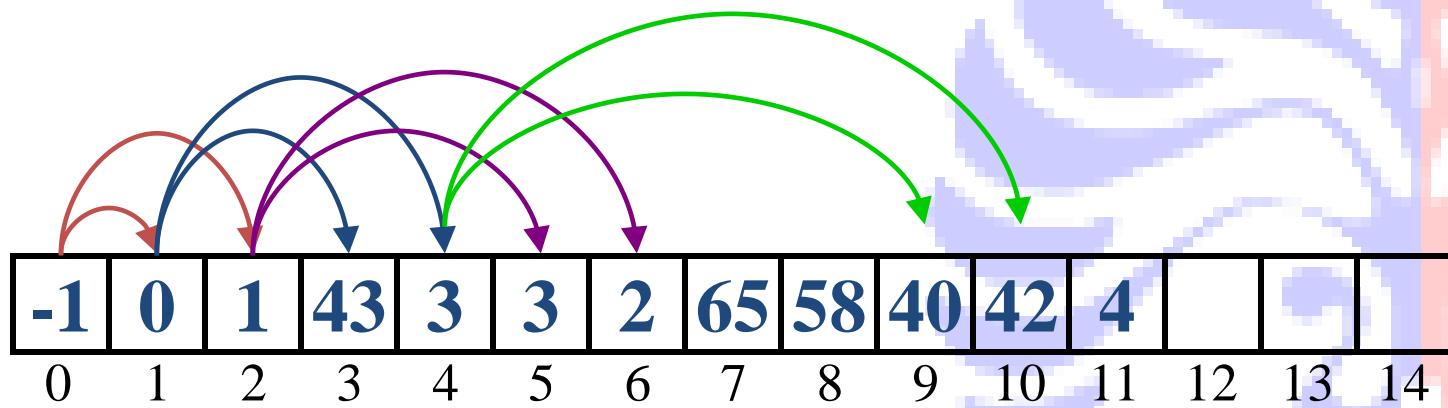
# Mana yang Binary Heap?



# Representasi Heap

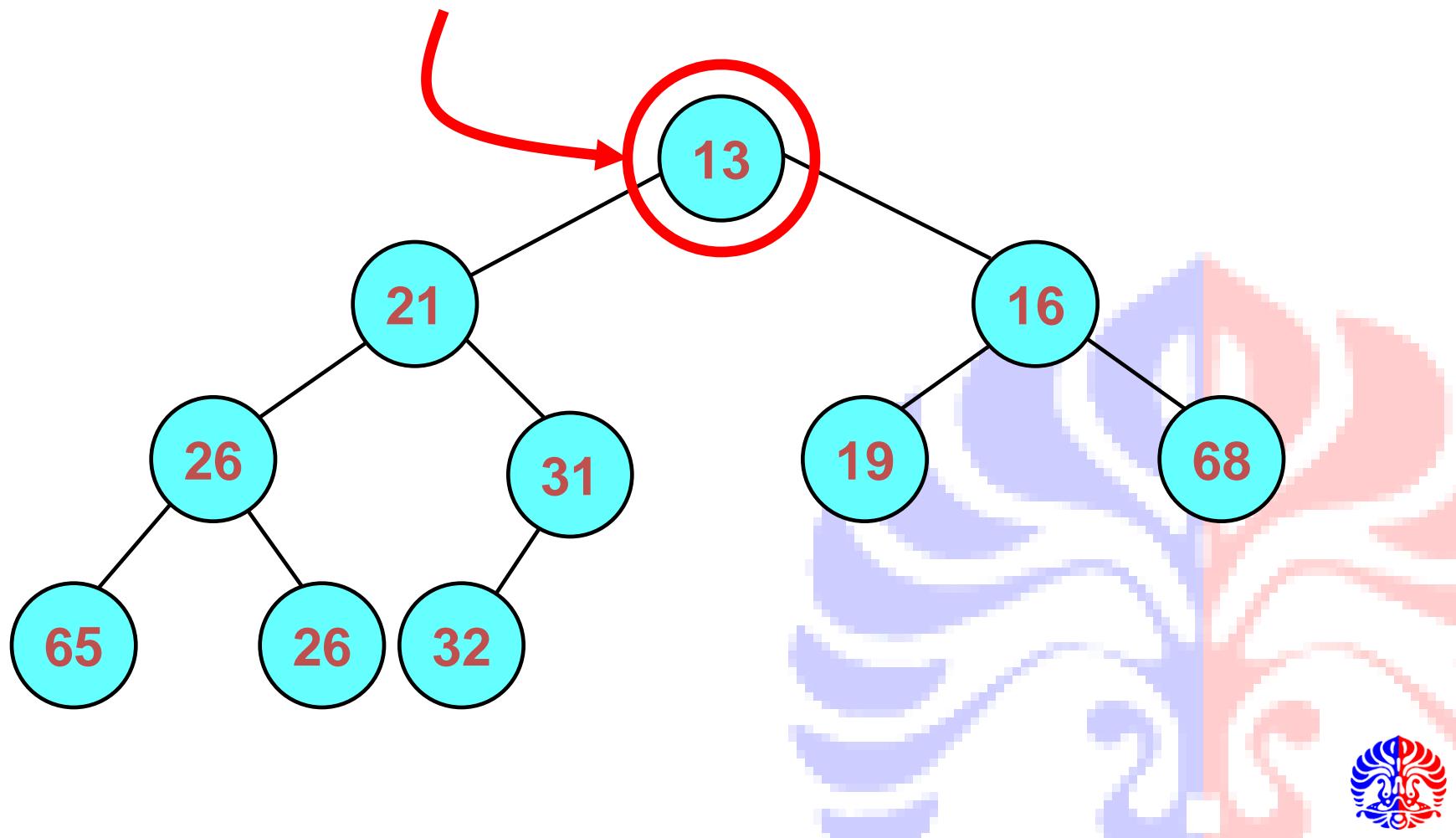


- Root pada indeks 0
- anak kiri dari  $i$  pada indeks  $2i + 1$
- anak kanan dari  $i$  pada indeks  $2i + 2 = 2(i + 1)$
- Parent dari  $i$  pada indeks  $\text{floor}((i - 1) / 2)$



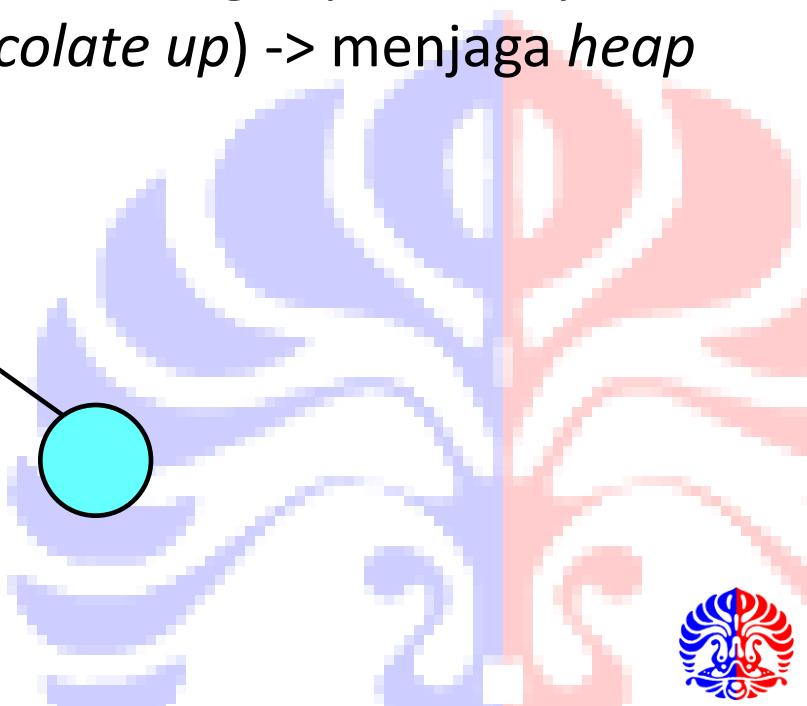
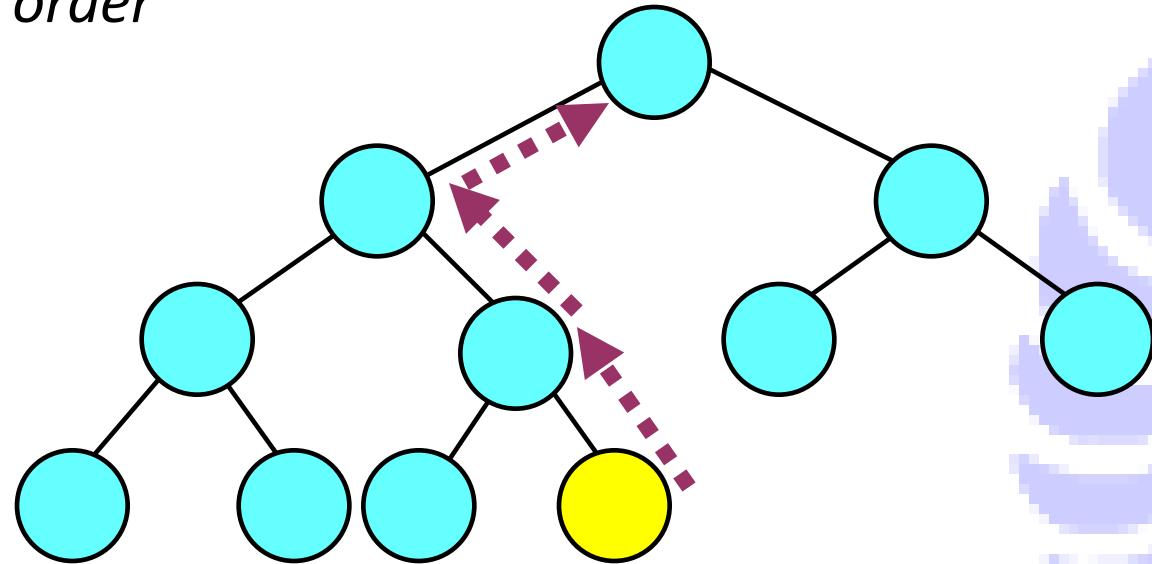
# Find Minimum

- Bagaimana cara mengakses elemen terkecil?
- Return root -> konstan



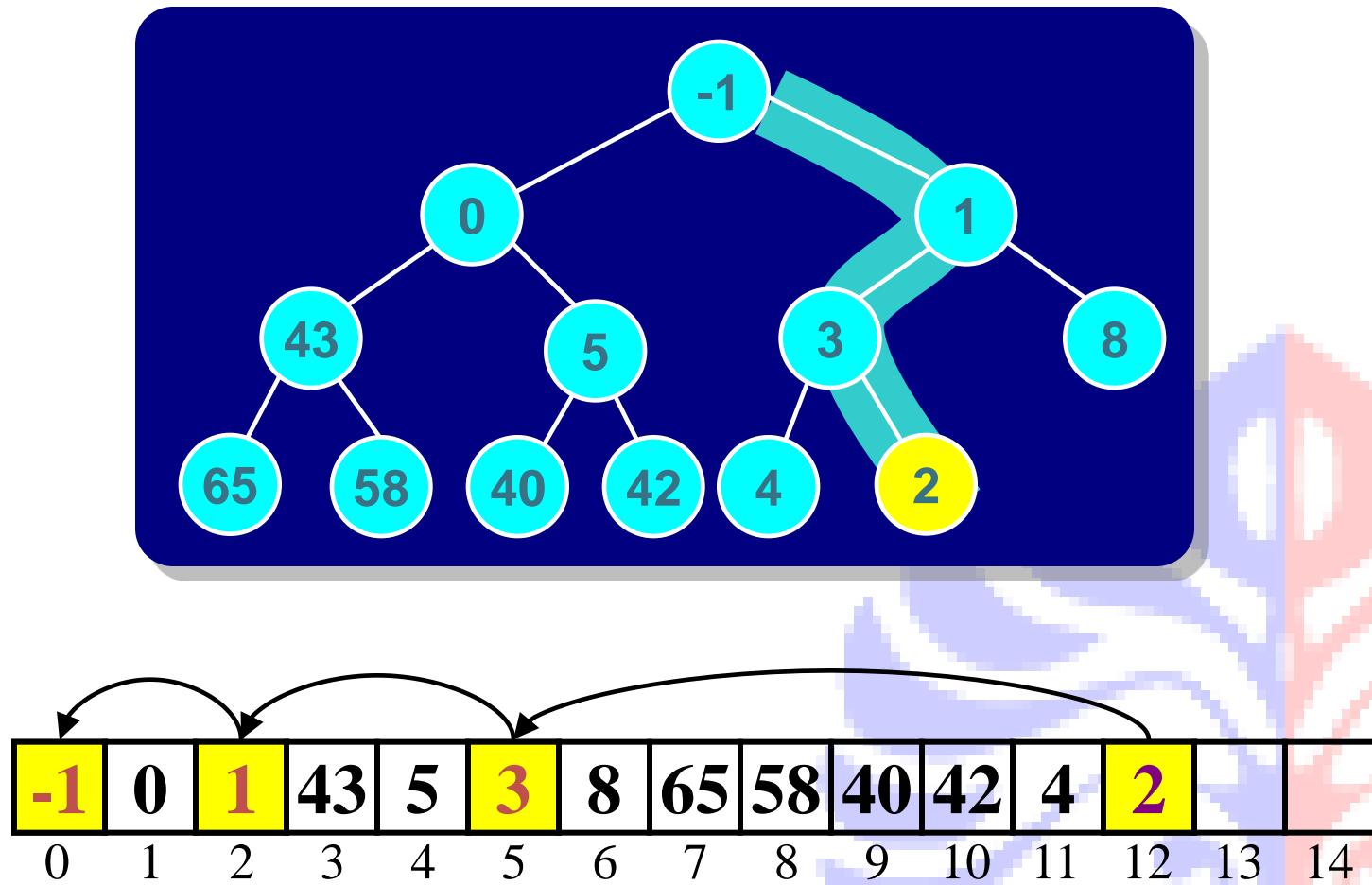
# Insertion

- Masukkan elemen baru ke:
  - Level paling bawah yang masih ada slot kosongnya, bila level sudah penuh, buat level baru lagi.
  - Letakkan elemen baru tersebut di slot kosong paling kiri. Tetap menjaga *complete binary tree*.
- Bila *node parent* lebih besar, tukar elemen dengan *parent*-nya. Lakukan hal tersebut sampai *root* (*percolate up*) -> menjaga *heap order*



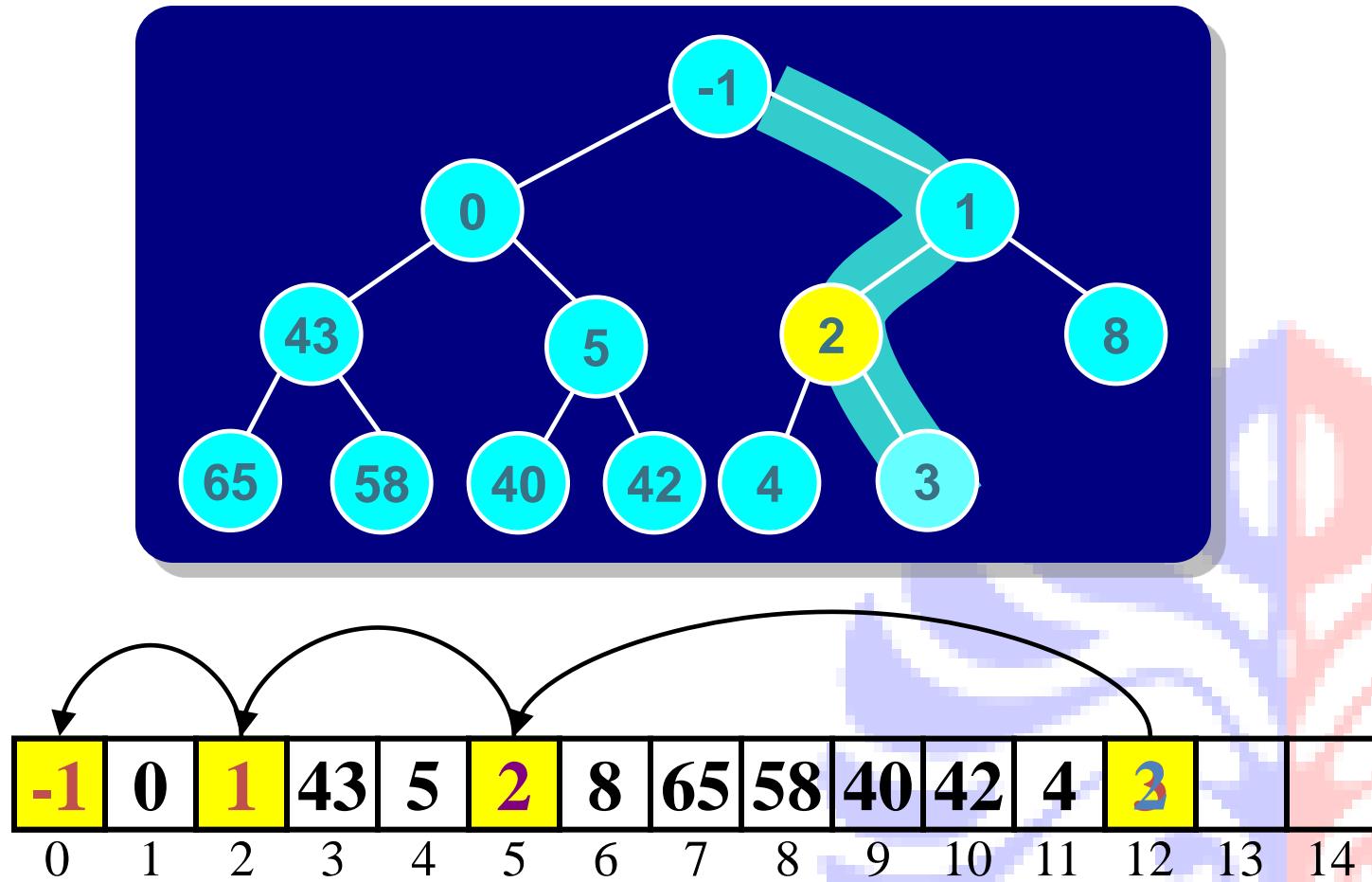
# Insertion

- Insert 2 (Percolate Up)



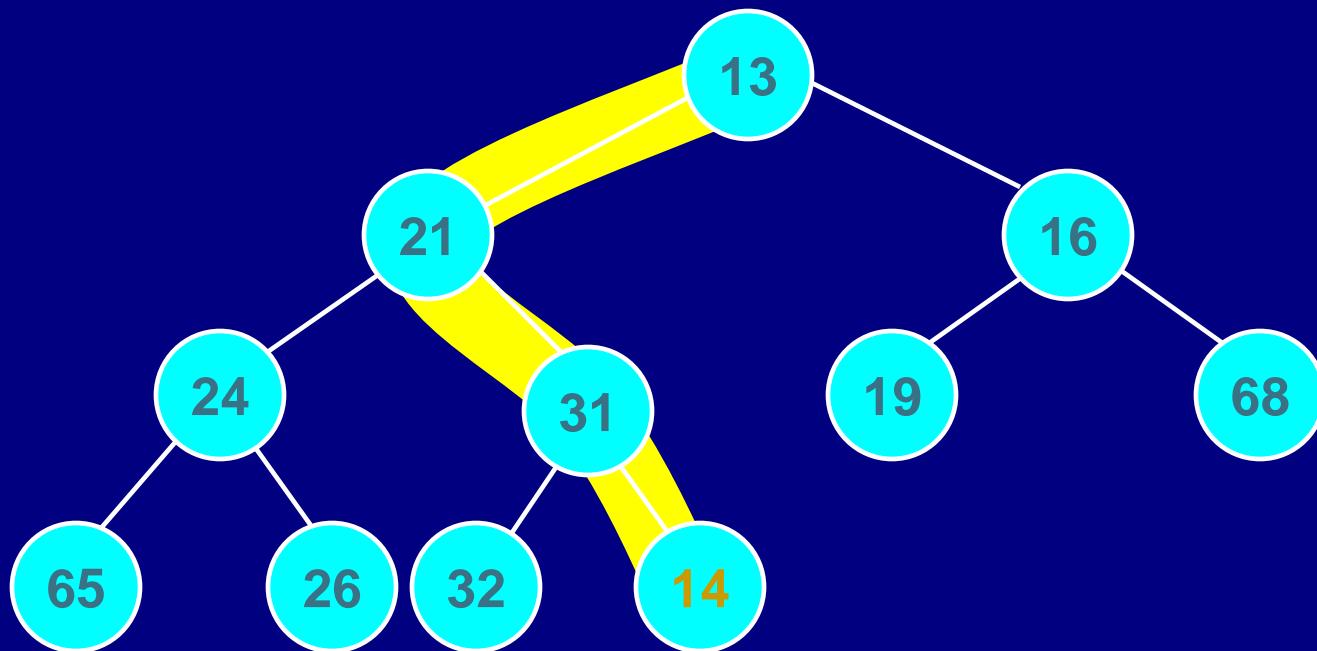
# Insertion

- Insert 2 (Percolate Up)



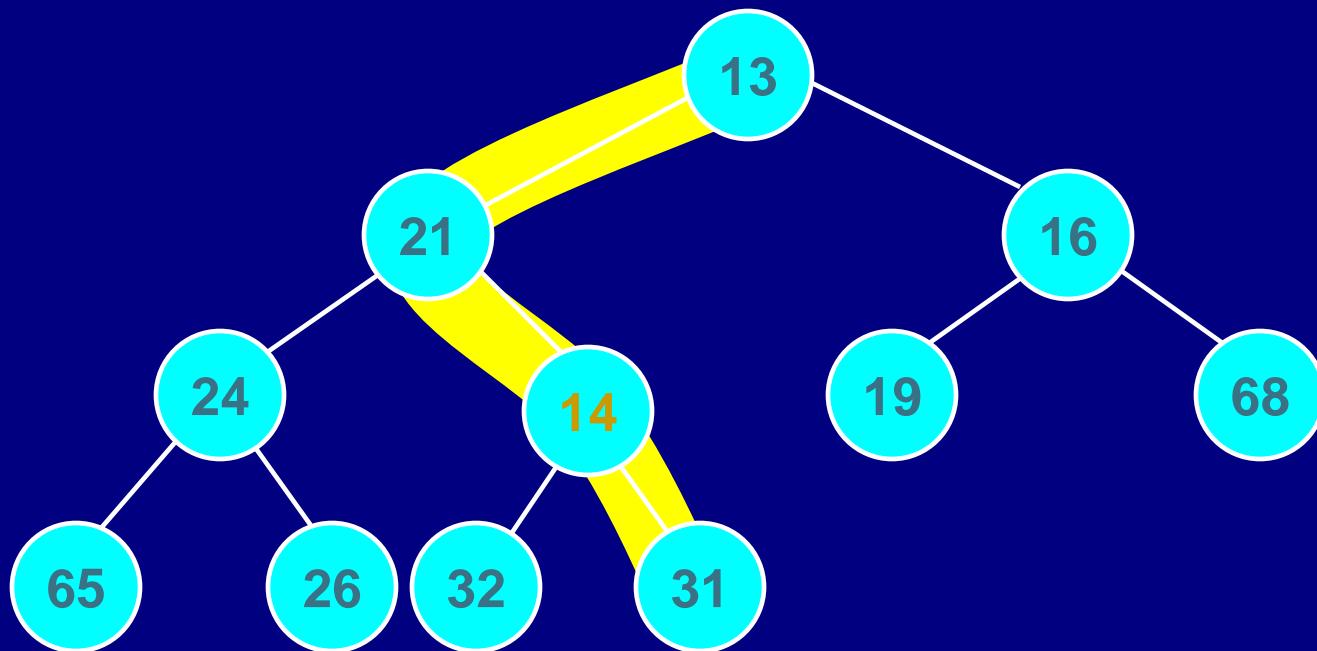
# Insertion

- Insert 14



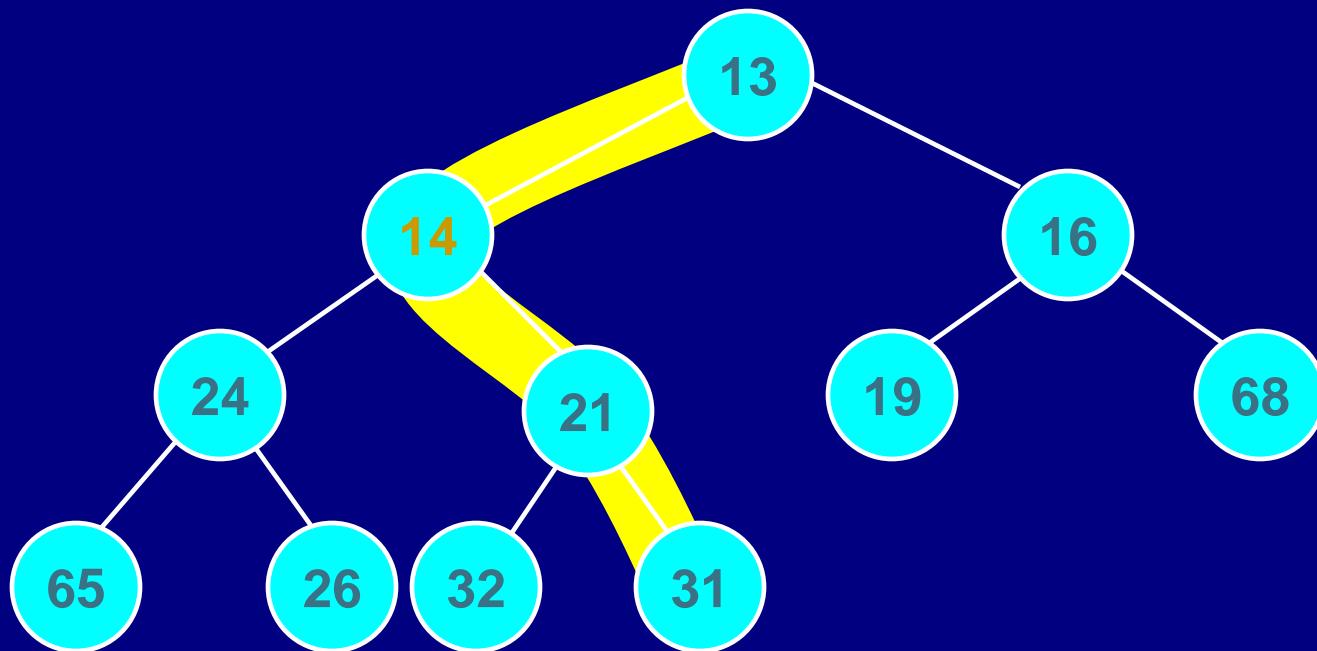
# Insertion

- Insert 14



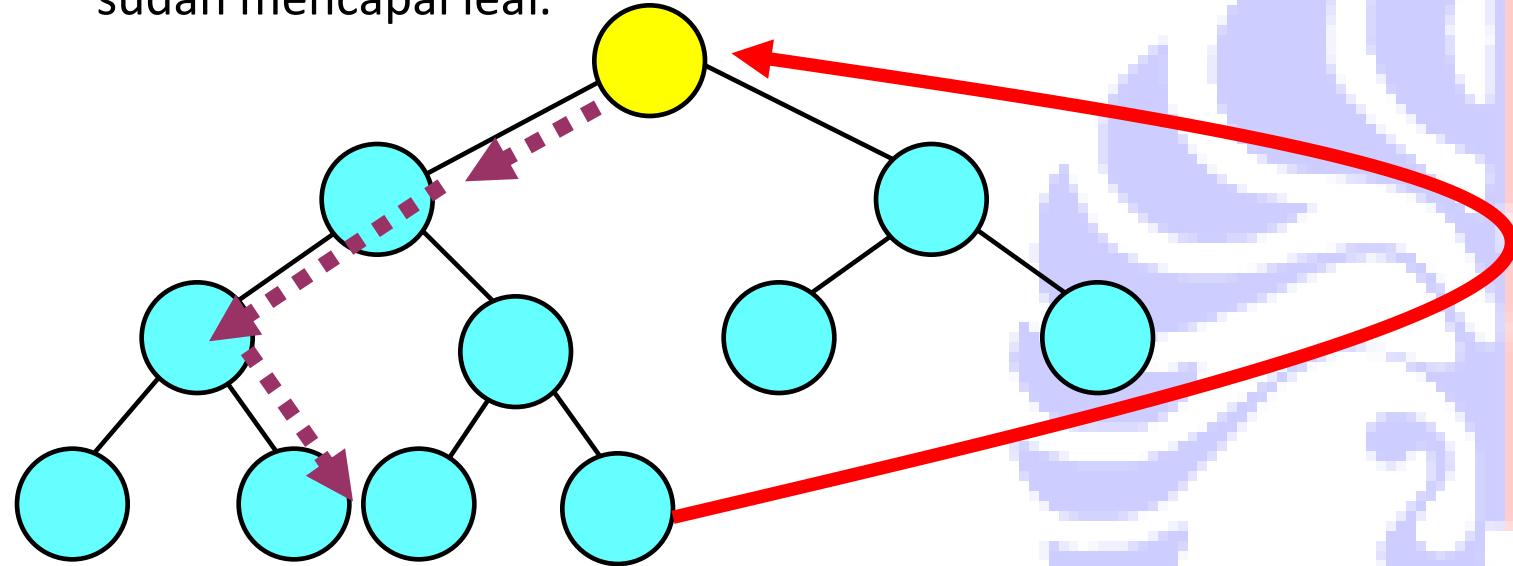
# Insertion

- Insert 14



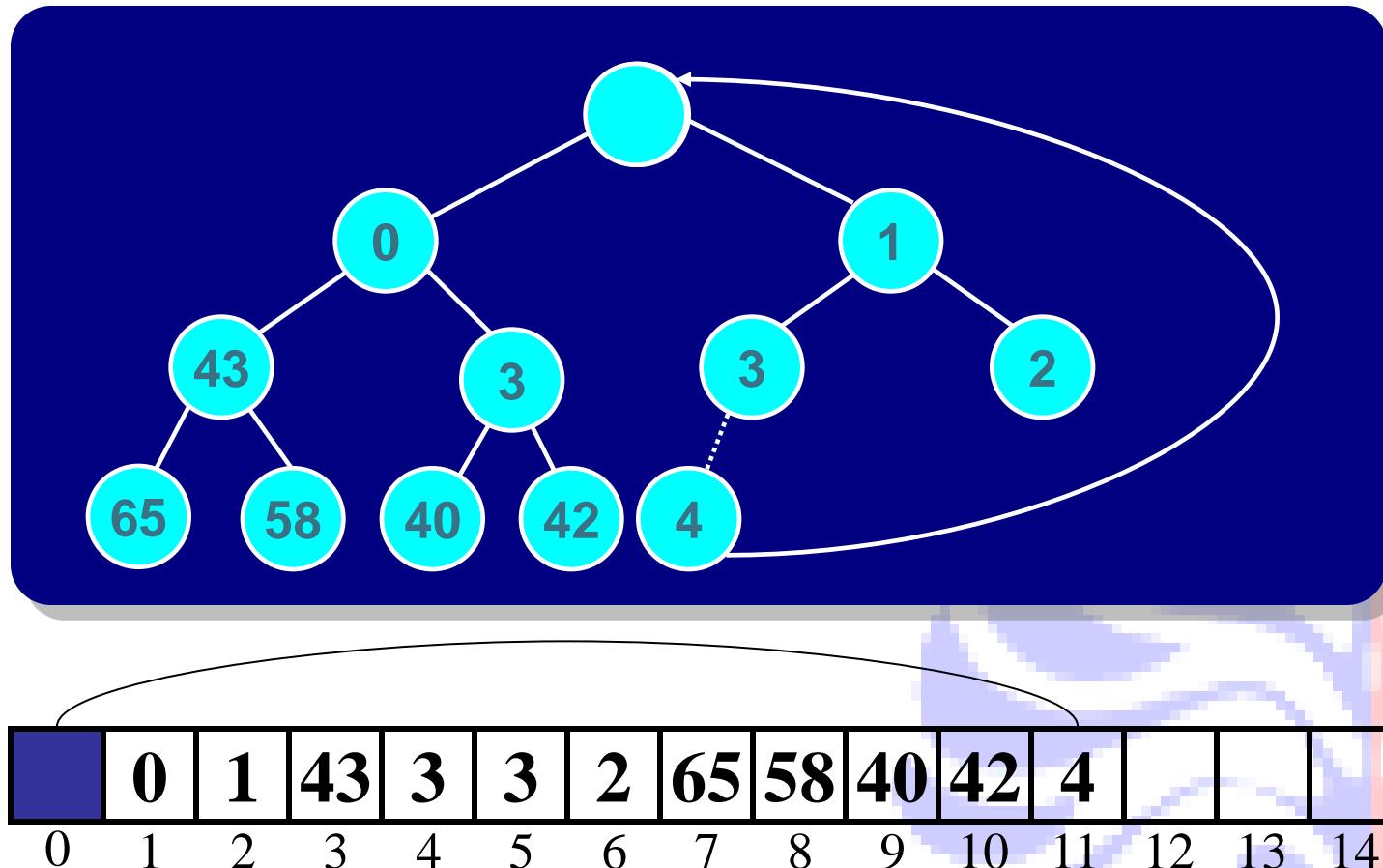
# Delete Minimum

- Hanya *root* yang dapat dihapus. Mengapa?
  - Kita hanya dapat mengakses elemen terkecil saja.
- Tukar *root* dengan elemen paling kanan di level paling bawah.
- Lakukan *percolate down*:
  - Dimulai dari root, cari anak **terkecil** dari node tersebut, bila anak terkecilnya < node yang sedang dikunjungi, tukar node tsb. Lakukan sampai anak terkecil dari node yang dikunjungi tidak < dari node yang dikunjungi atau sudah mencapai leaf.

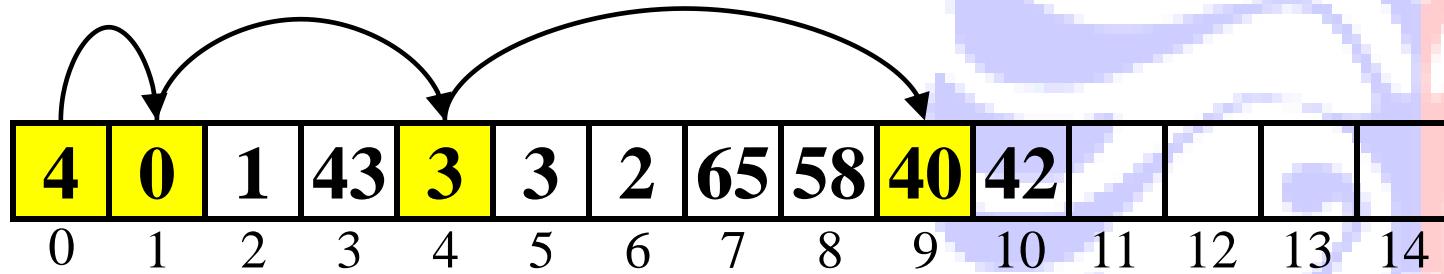
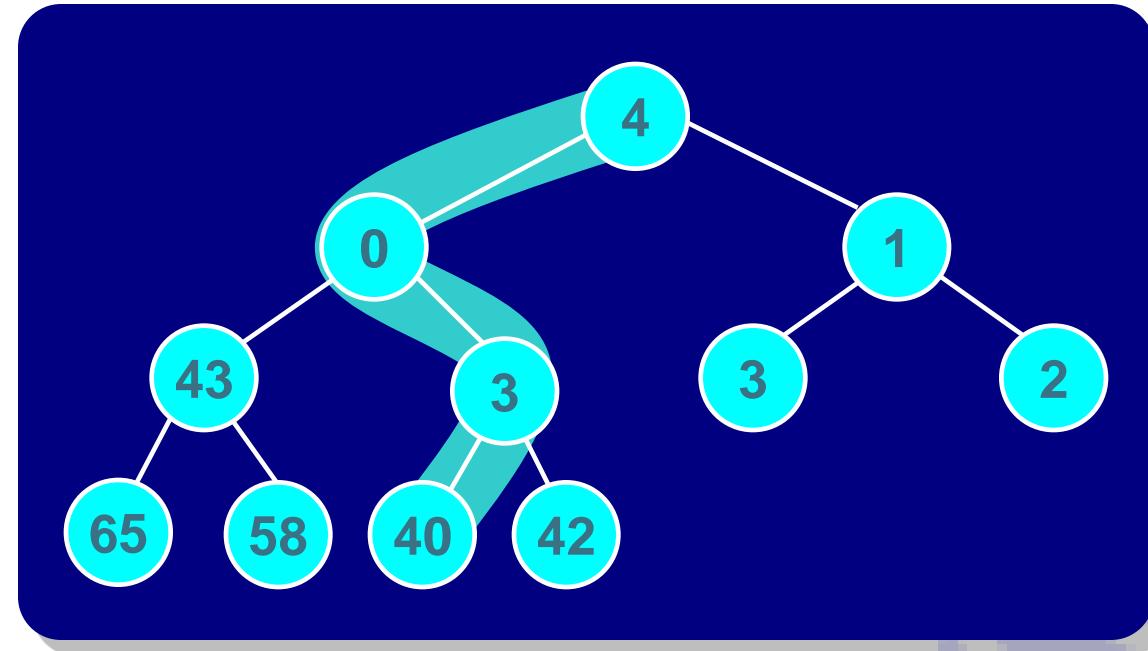


# Delete Minimum

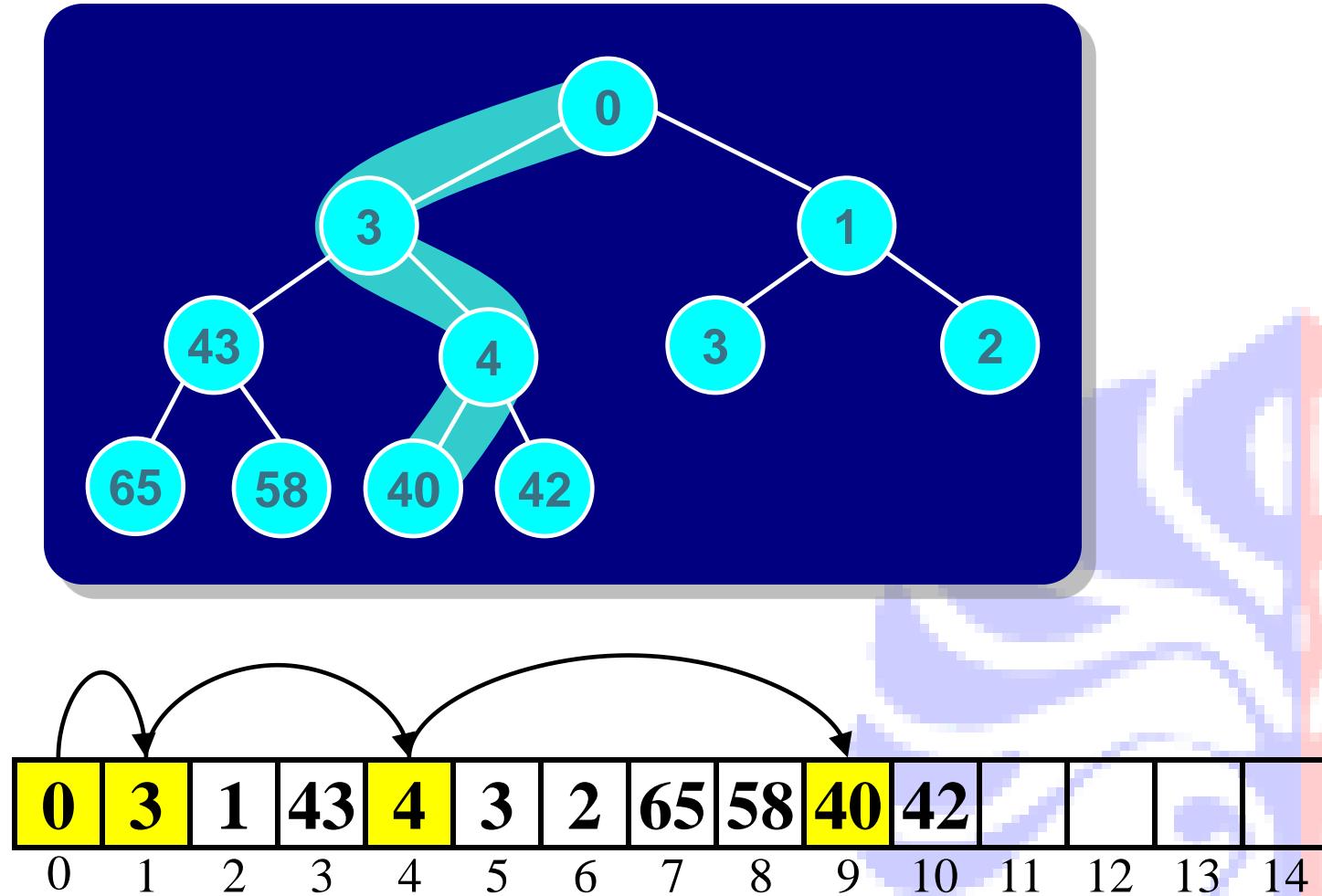
- Percolate Down



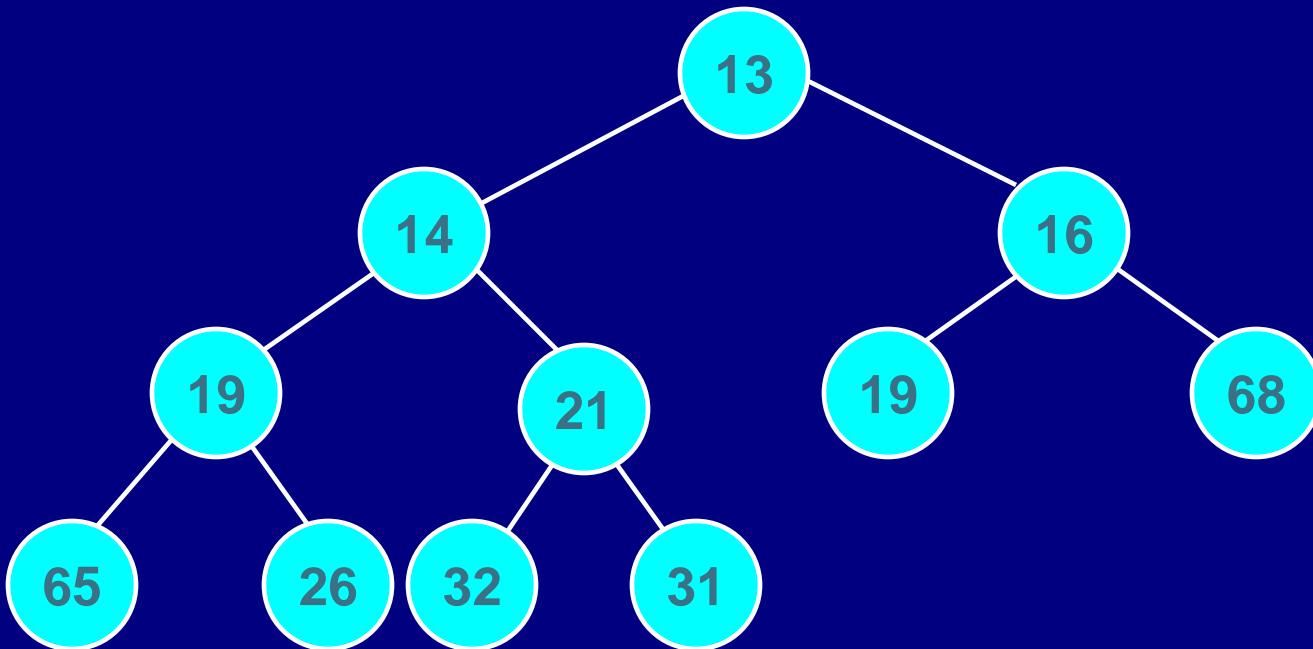
# Delete Minimum



# Delete Minimum: Completed

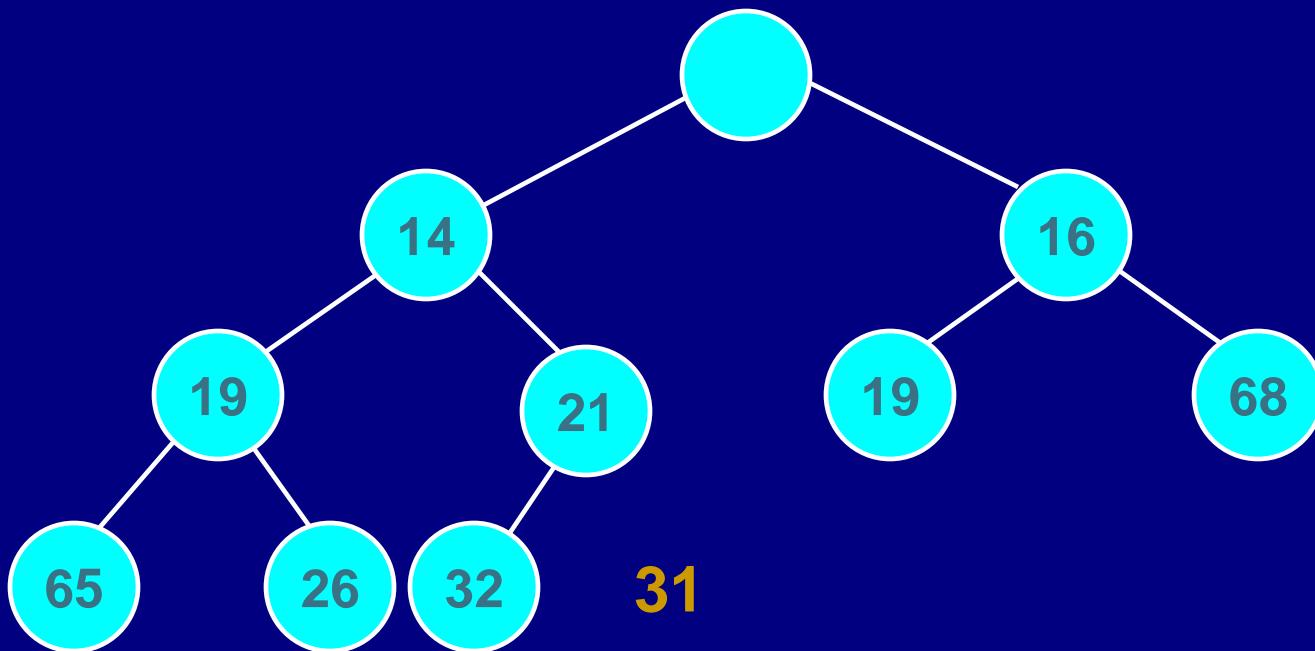


# Delete Min (2)

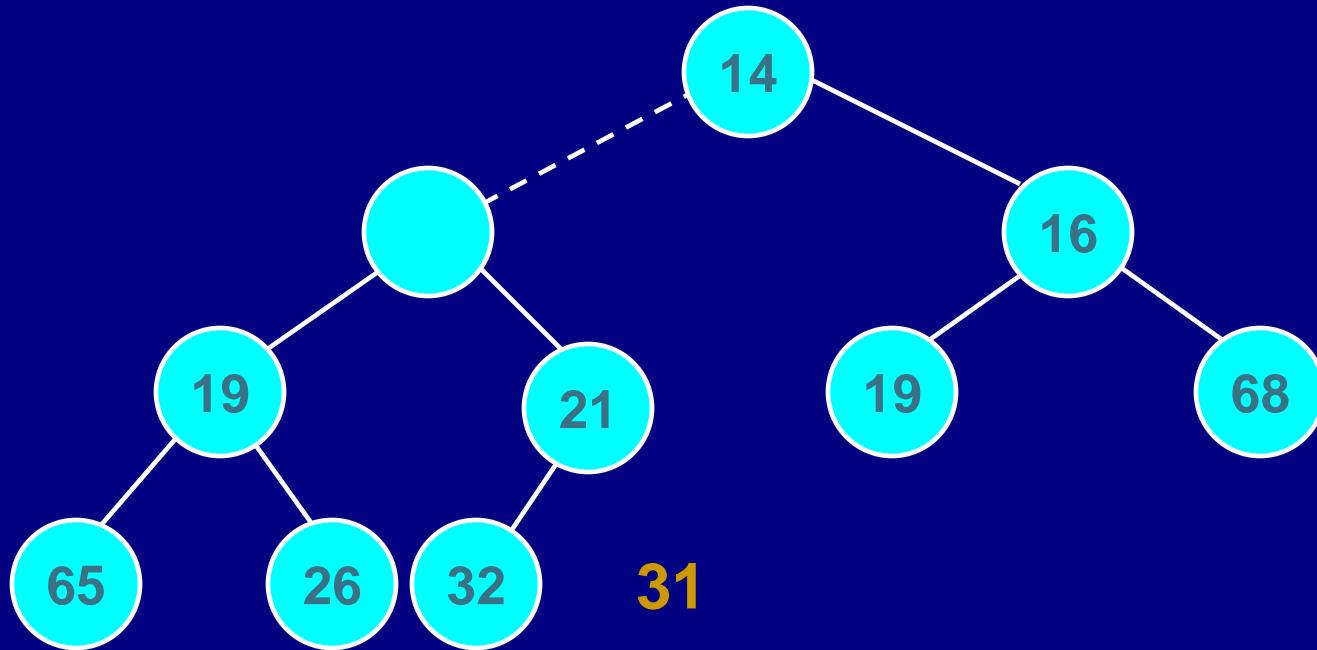


# Delete Min (2)

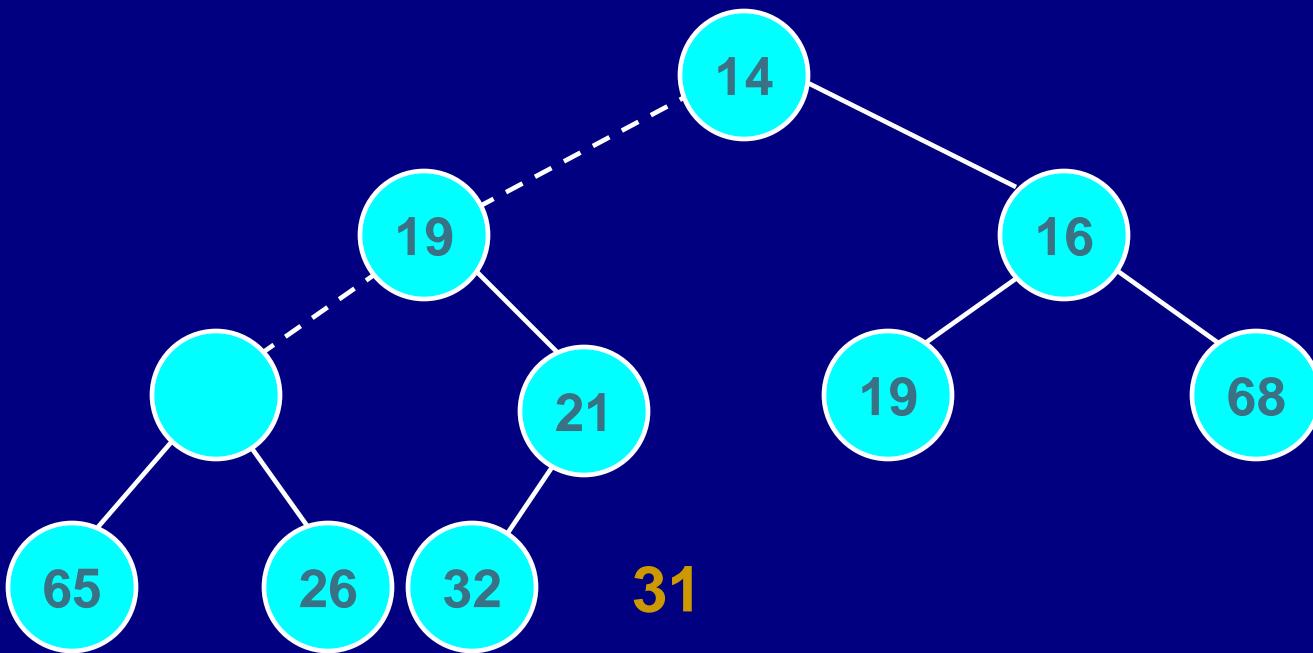
- Percolate Down



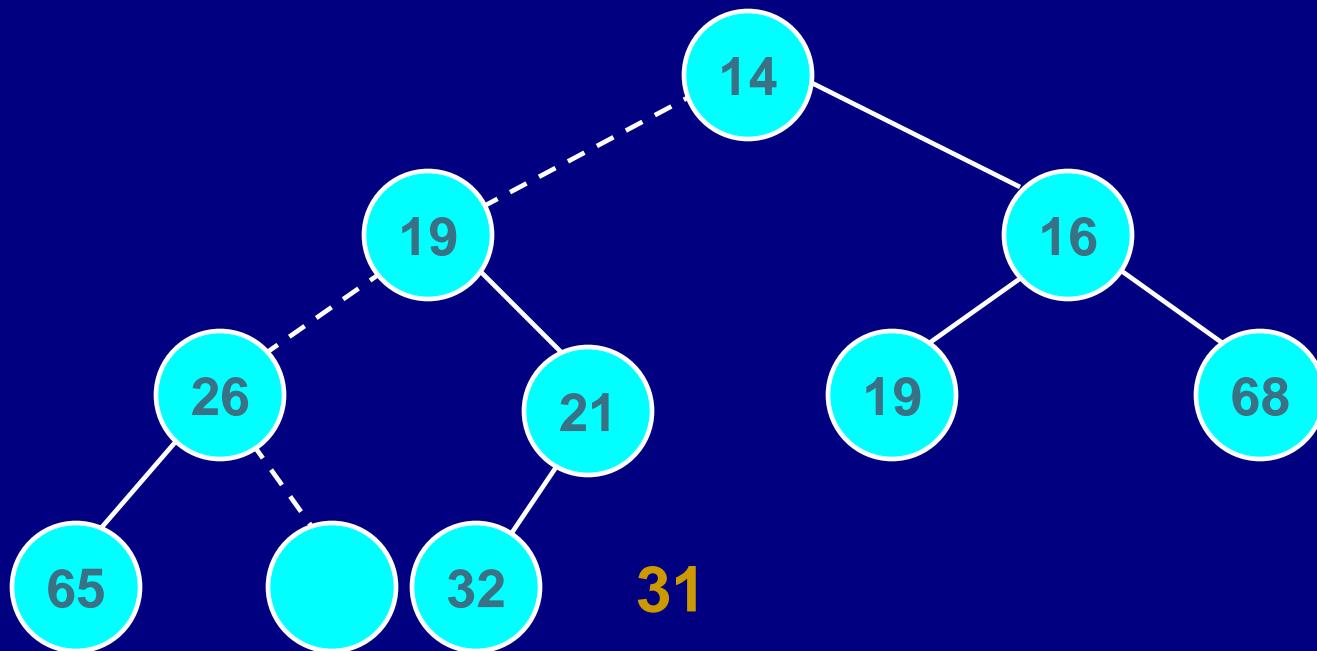
# Delete Min (2)



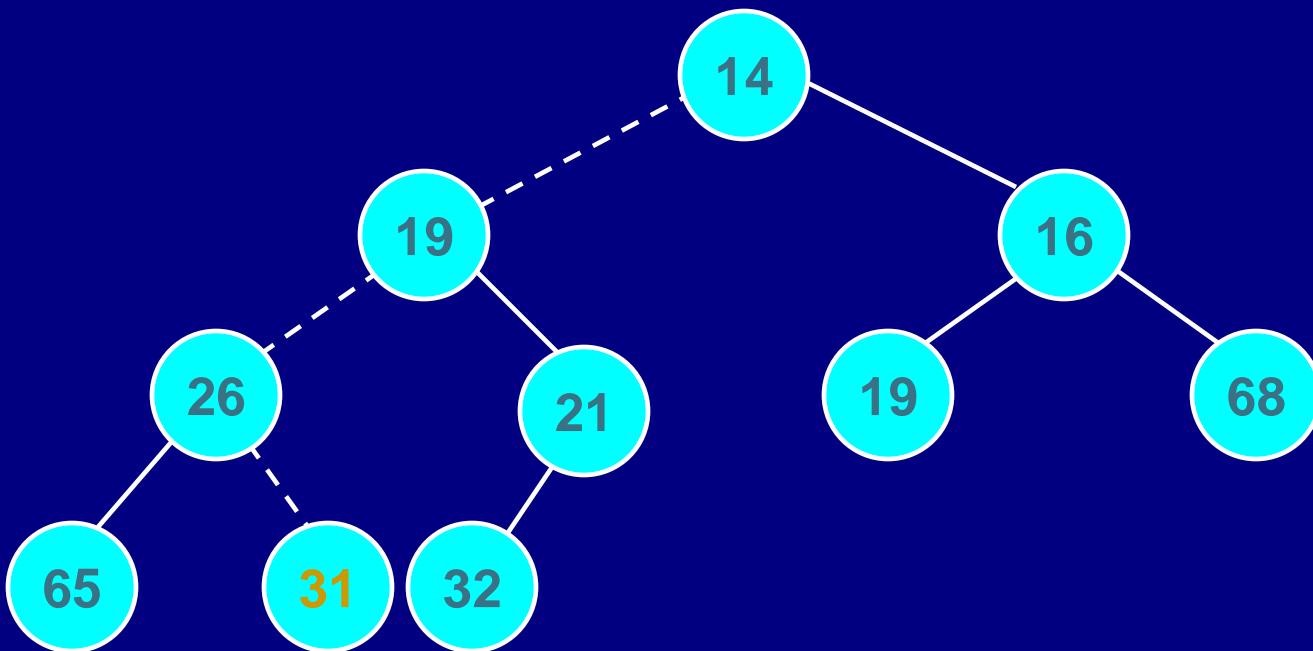
# Delete Min (2)



# Delete Min (2)



# Delete Min (2)



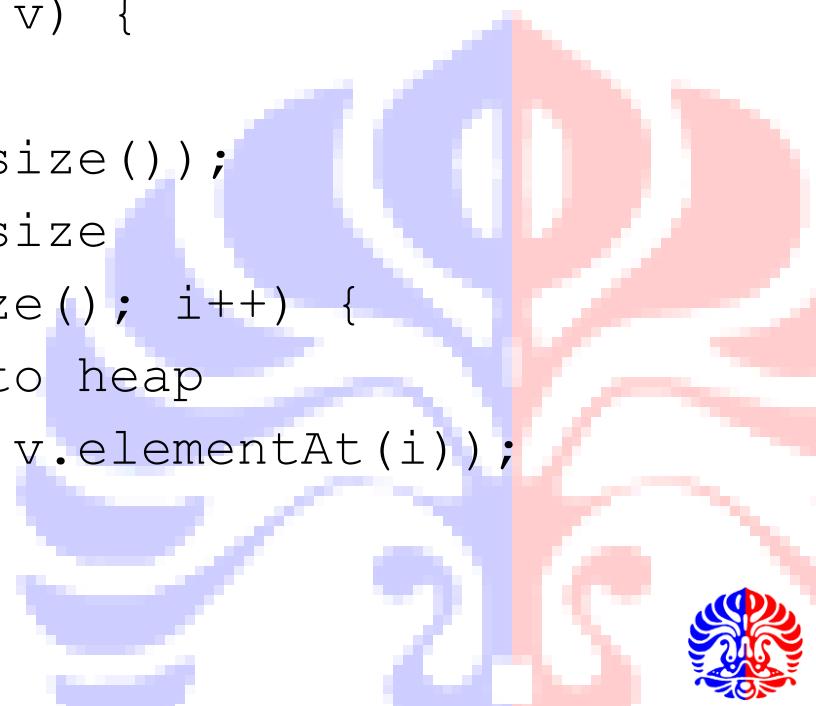
# Latihan

- Simulasi operasi-operasi berikut pada *Min Binary Heap*:
  - Insert: 40, 20, 5, 55, 76, 31, 3
  - Delete Min
  - Delete Min
  - Insert: 10, 22
  - Delete Min
  - Delete Min
- Gambarkan isi Binary Heap, setelah operasi terakhir.



# Heap Constructor

```
public class VectorHeap implements PriorityQueue  
{  
    protected Vector data;  
  
    public VectorHeap() { data = new Vector(); }  
  
    public VectorHeap(Vector v) {  
        int i;  
        data = new Vector(v.size());  
        // we know ultimate size  
        for (i = 0; i < v.size(); i++) {  
            // add elements to heap  
            add((Comparable) v.elementAt(i));  
        }  
    }  
}
```



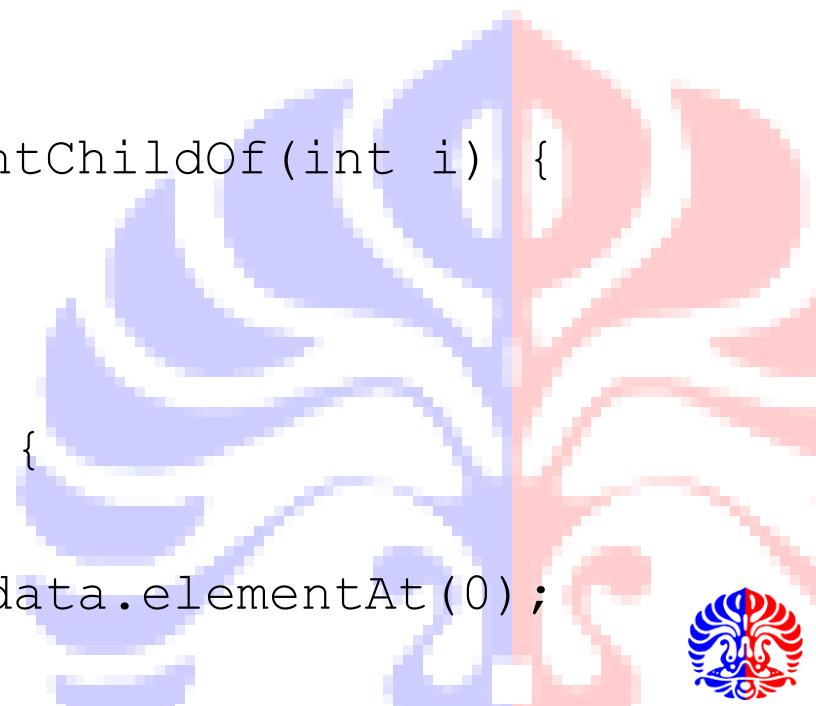
# Heap's Methods

```
protected static int parentOf(int i) {  
    return (i - 1) / 2;  
}
```

```
protected static int leftChildOf(int i) {  
    return 2 * i + 1;  
}
```

```
protected static int rightChildOf(int i) {  
    return 2 * (i + 1);  
}
```

```
public Comparable peek() {  
    // findMin  
    return (Comparable) data.elementAt(0);  
}
```



# Removal & Insertion

```
public Comparable remove()  
{  
    Comparable minValue = peek();  
    data.setElementAt(  
        data.elementAt(data.size() - 1), 0);  
    data.setSize(data.size() - 1);  
    if (data.size() > 1) pushDownRoot(0);  
    return minValue;  
}  
  
public void add(Comparable value) {  
    data.addElement(value);  
    percolateUp(data.size() - 1);  
}
```

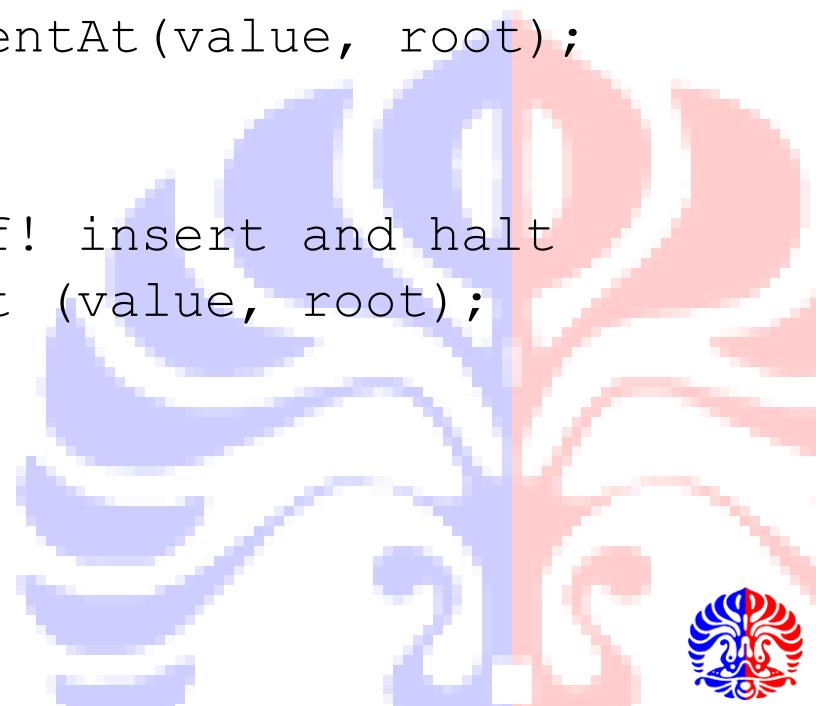


# Perculate Down

```
protected void pushDownRoot (int root)
{
    int heapSize = data.size();
    Comparable value = (Comparable)
        data.elementAt (root);
    while (root < heapSize) {
        int childpos = leftChildOf (root);
        if (childpos < heapSize) {
            // choose the smallest child
            if ((rightChildOf (root) < heapSize) &&
                (((Comparable) (data.elementAt (
                    childpos+1))).compareTo
                ((Comparable) (data.elementAt (
                    childpos))) < 0))
            {
                childpos++;
            }
        }
    }
}
```

# Percolate Down

```
if (((Comparable) (data.elementAt( childpos ) ) .compareTo ( value ) < 0)
{
    data.setElementAt (
        data.elementAt( childpos ), root );
    root = childpos; // keep moving down
} else { // found right location
    data.setElementAt ( value, root );
    return;
}
} else { // at a leaf! insert and halt
    data.setElementAt ( value, root );
    return;
}
}
```



# Perculate Up

```
protected void percolateUp (int leaf)
{
    int parent = parentOf(leaf);
    Comparable value =
        (Comparable) (data.elementAt(leaf));
    while (leaf > 0 && (value.compareTo
        ((Comparable) (data.elementAt(parent))) < 0))
    {
        data.setElementAt (data.elementAt
            (parent), leaf);
        leaf = parent;
        parent = parentOf(leaf);
    }
    data.setElementAt (value, leaf);
}
```



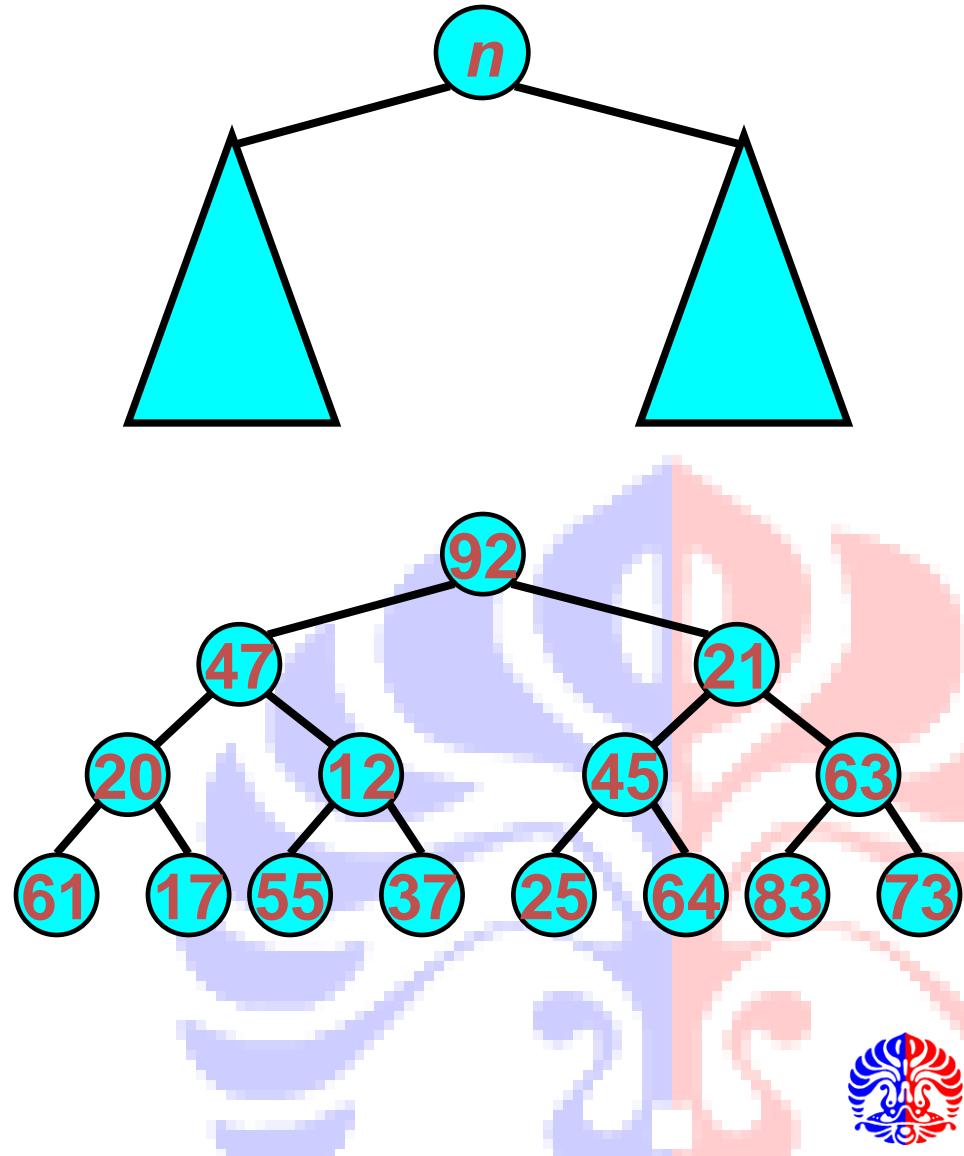
# Heap's Methods

```
public boolean isEmpty()
{
    return data.size() == 0;
}
public int size()
{
    return data.size();
}
public void clear()
{
    data.clear();
}
public String toString()
{
    return "<VectorHeap: " + data + ">";
}
```



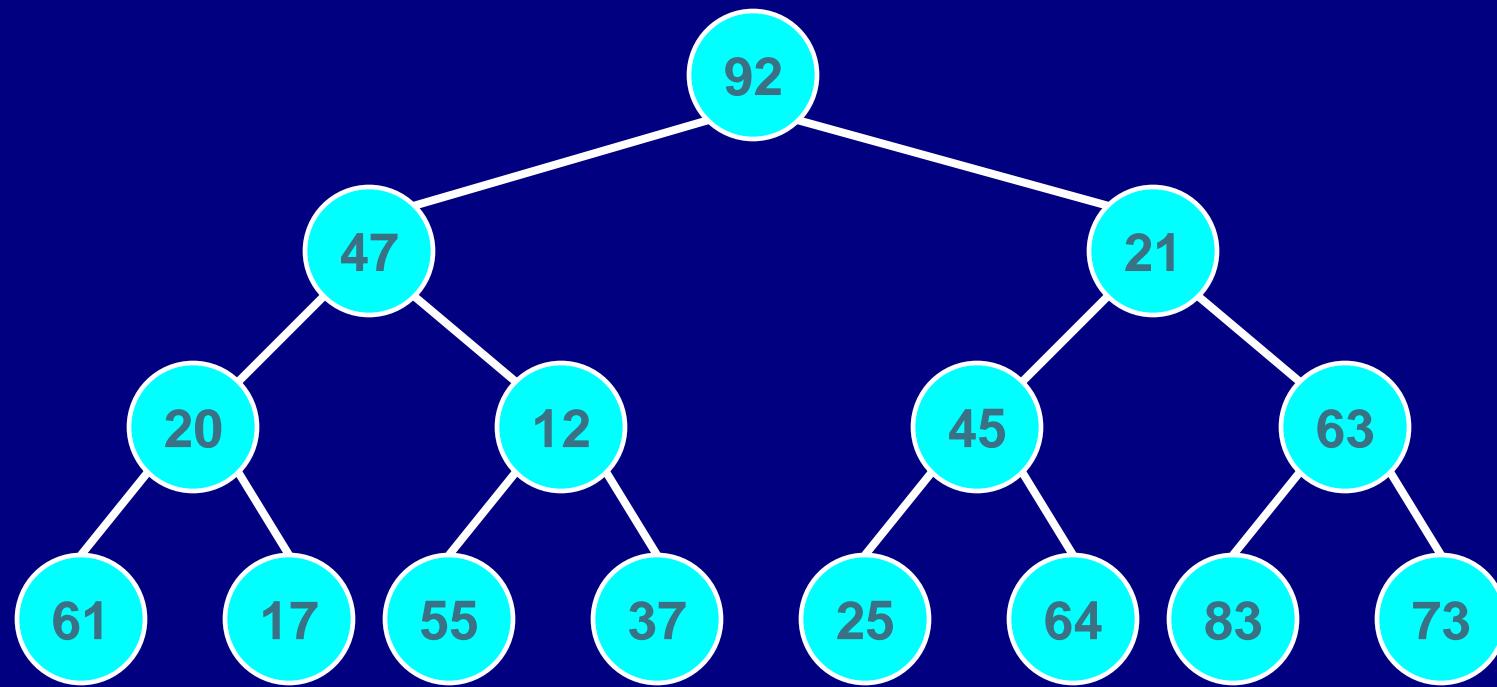
# fixHeap Algorithm

- Versi rekursif:
  - **fixHeap** left subtree
  - **fixHeap** right subtree
  - **percolateDown** root
- Tidak efektif! Ada cara lain yang lebih efisien:
- Panggil **percolateDown** menggunakan *reverse level order* pada *non-leaves node*
- Saat pemanggilan **percolateDown** pada *node n*, sub tree dibawahnya akan dijamin sudah memenuhi *heap order*



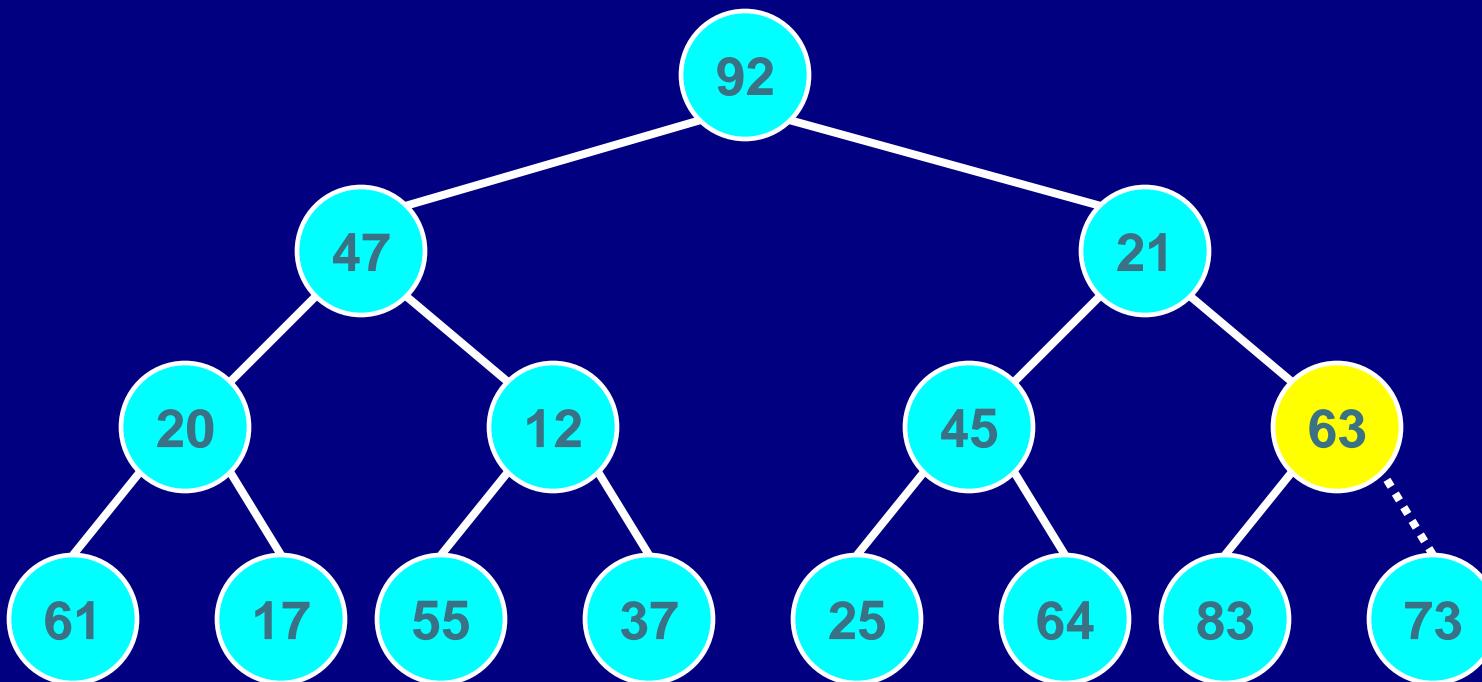
# Fix Heap / Heapify

- Operasi fixHeap menerima *complete tree* yang tidak memenuhi *heap order* dan memperbaikinya.
- penambahan sebanyak N dapat dilakukan dengan  $O(n \log n)$
- Operasi fix heap dapat dilakukan dengan  $O(n)$  !



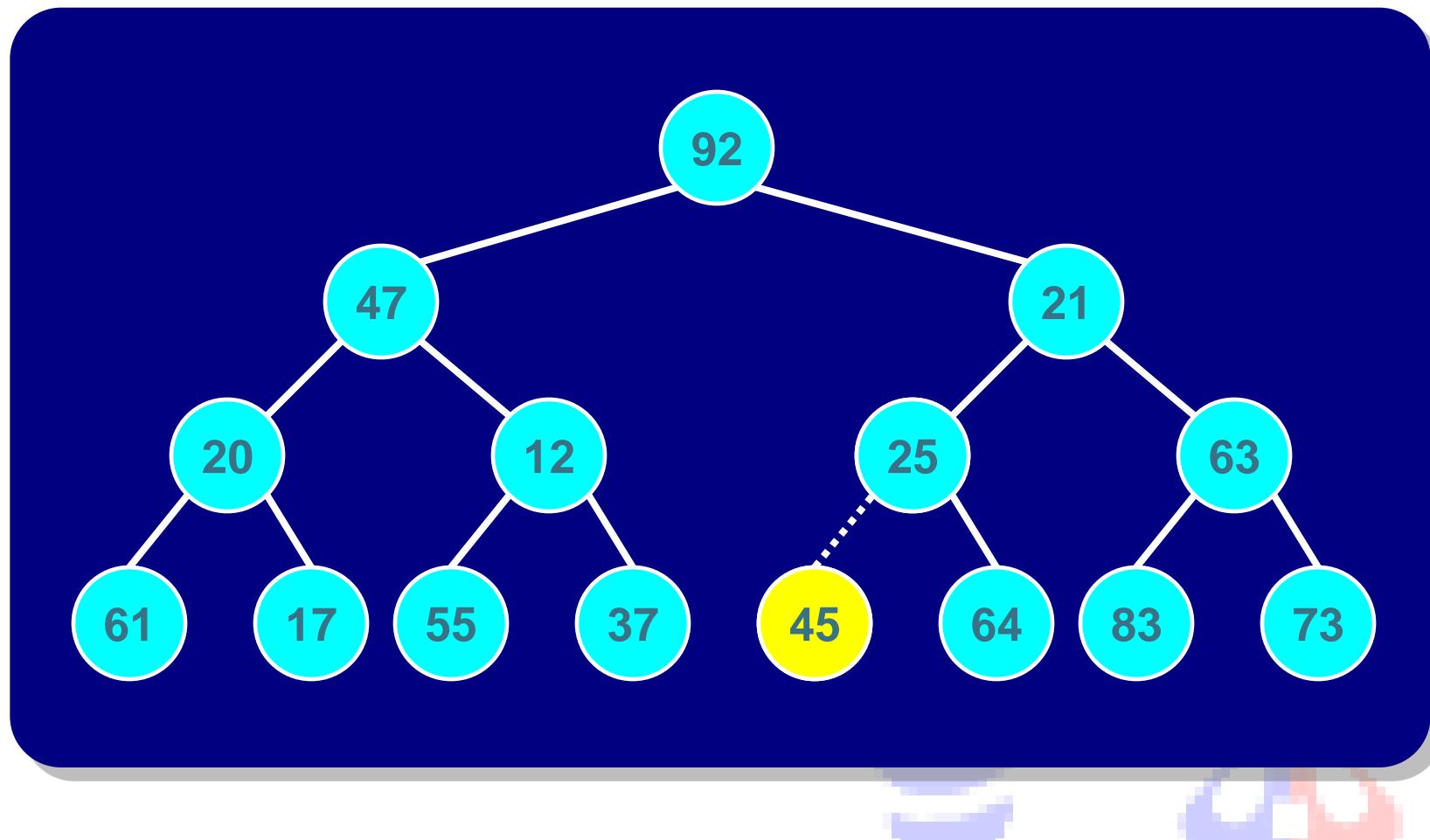
# Fix Heap / Heapify

- Percolate down 6



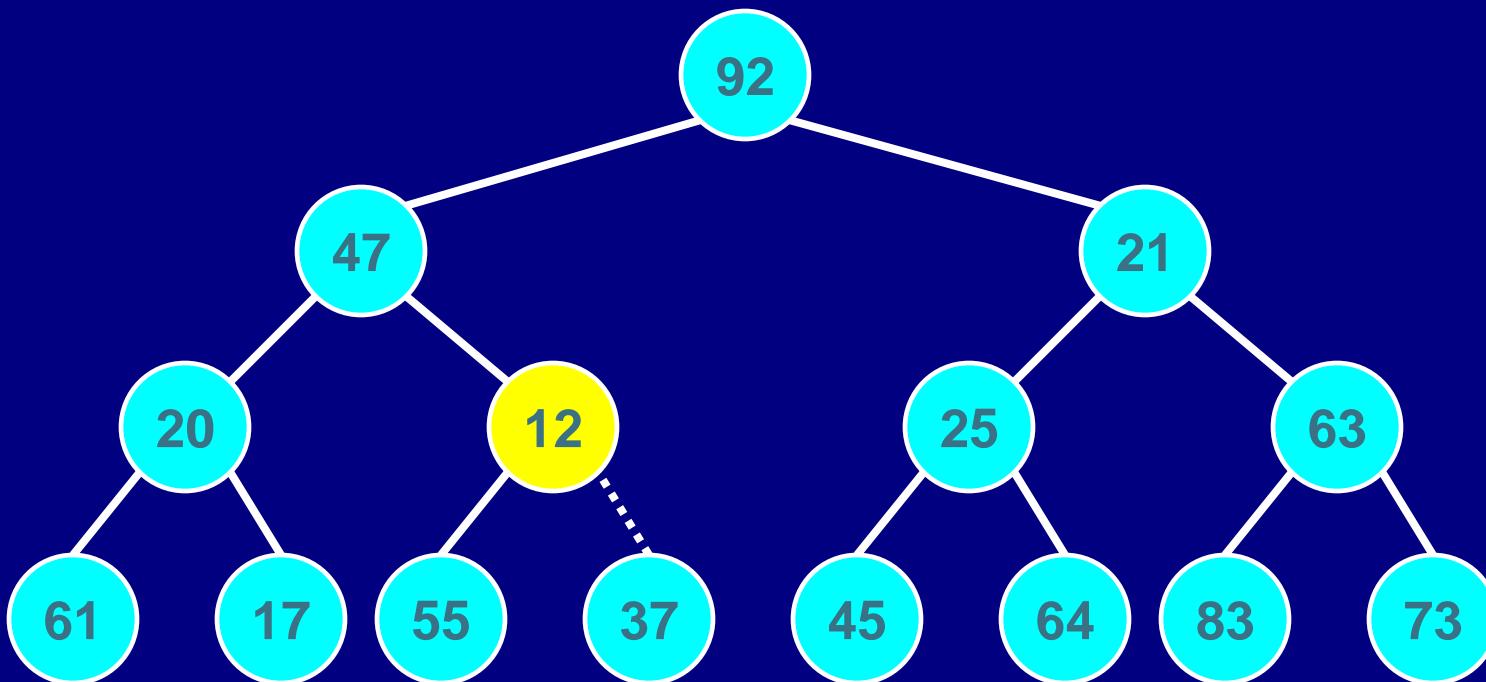
# Fix Heap / Heapify

- Percolate down 5



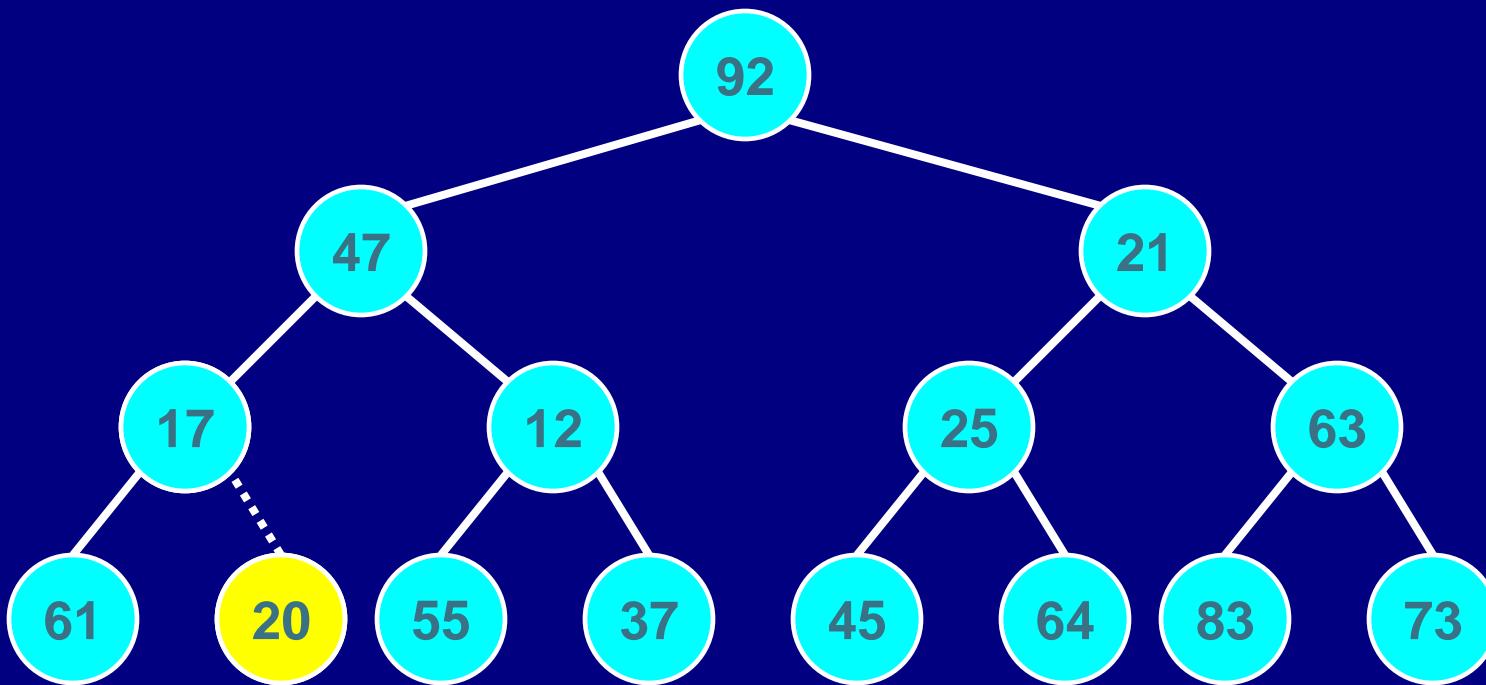
# Fix Heap / Heapify

- Percolate down 4



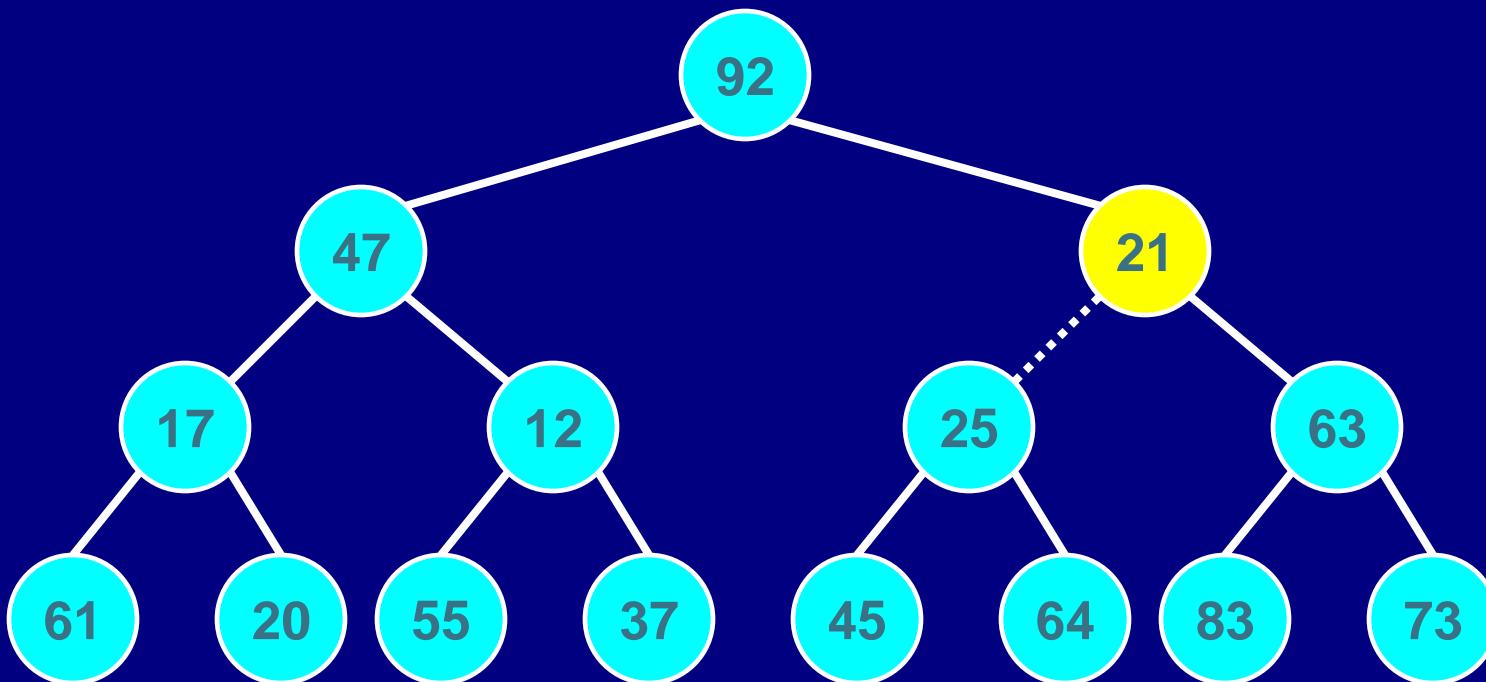
# Fix Heap / Heapify

- Percolate down 3



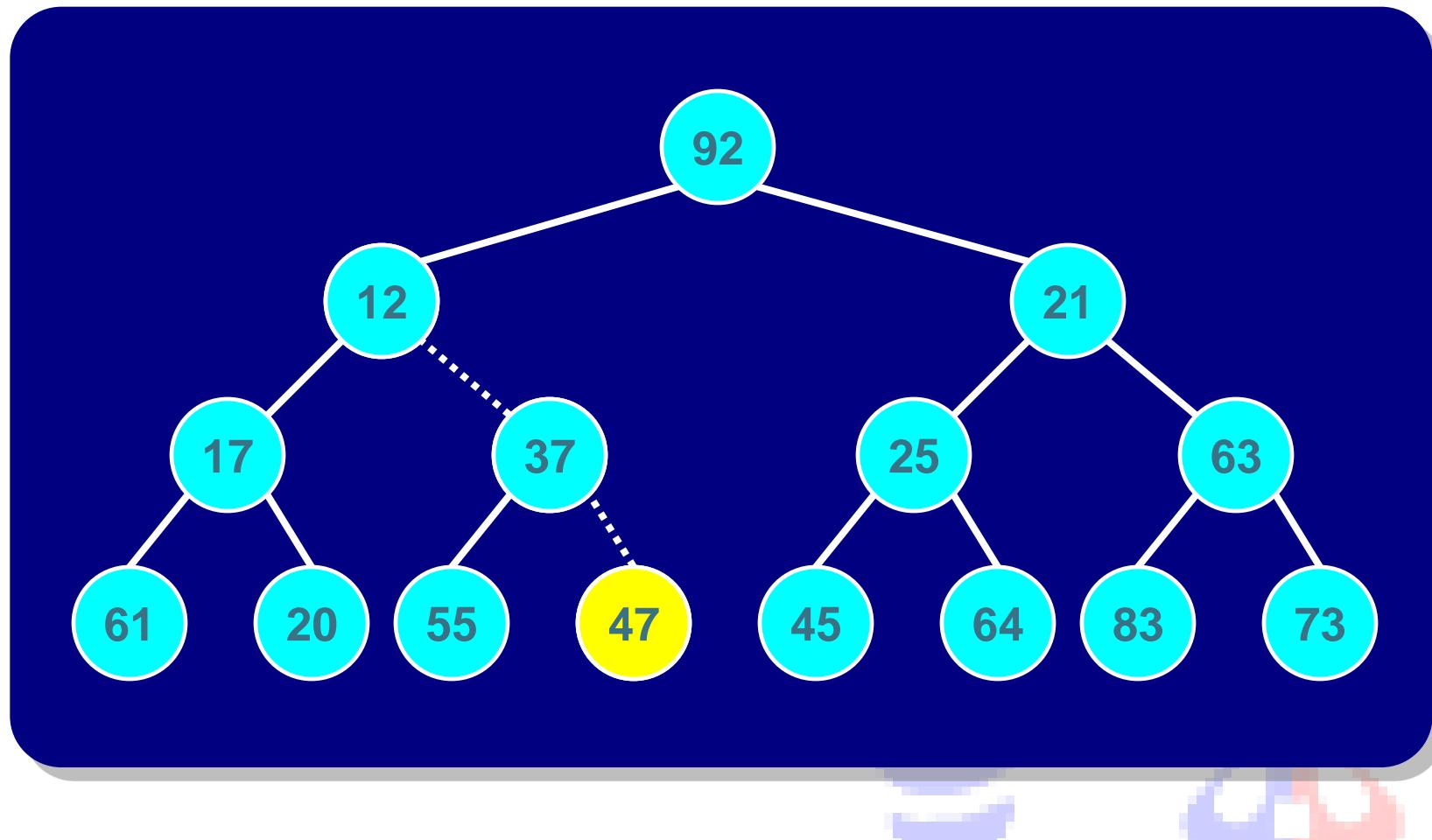
# Fix Heap / Heapify

- Percolate down 2



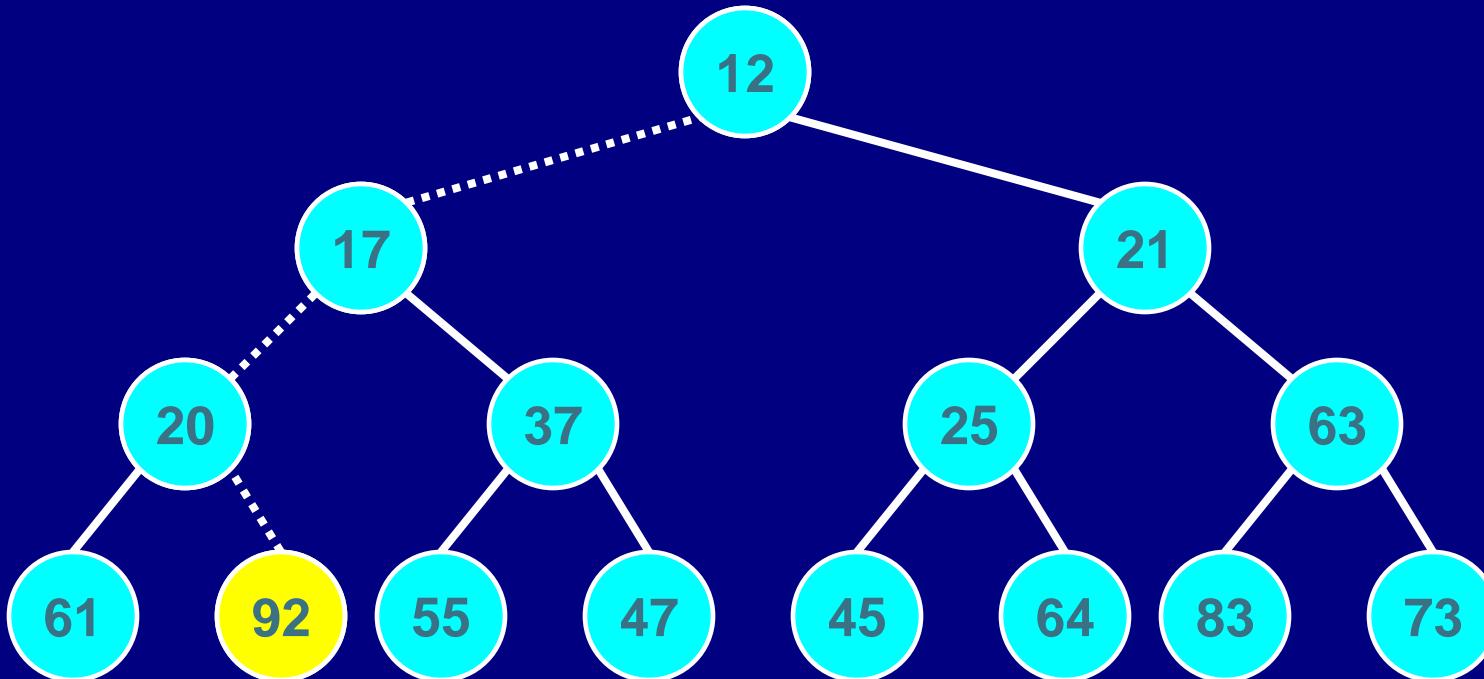
# Fix Heap / Heapify

- Percolate down 1

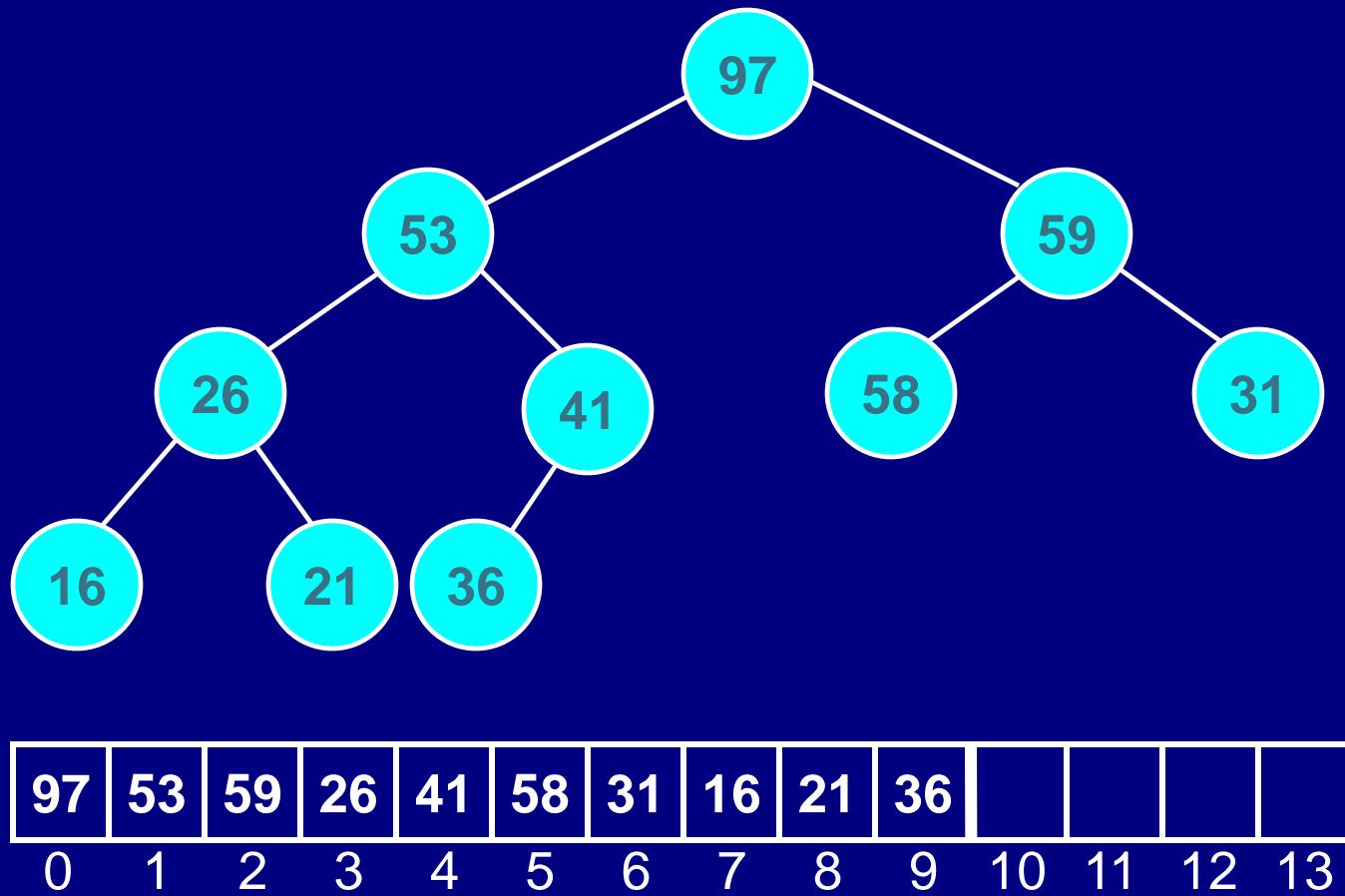


# Fix Heap / Heapify

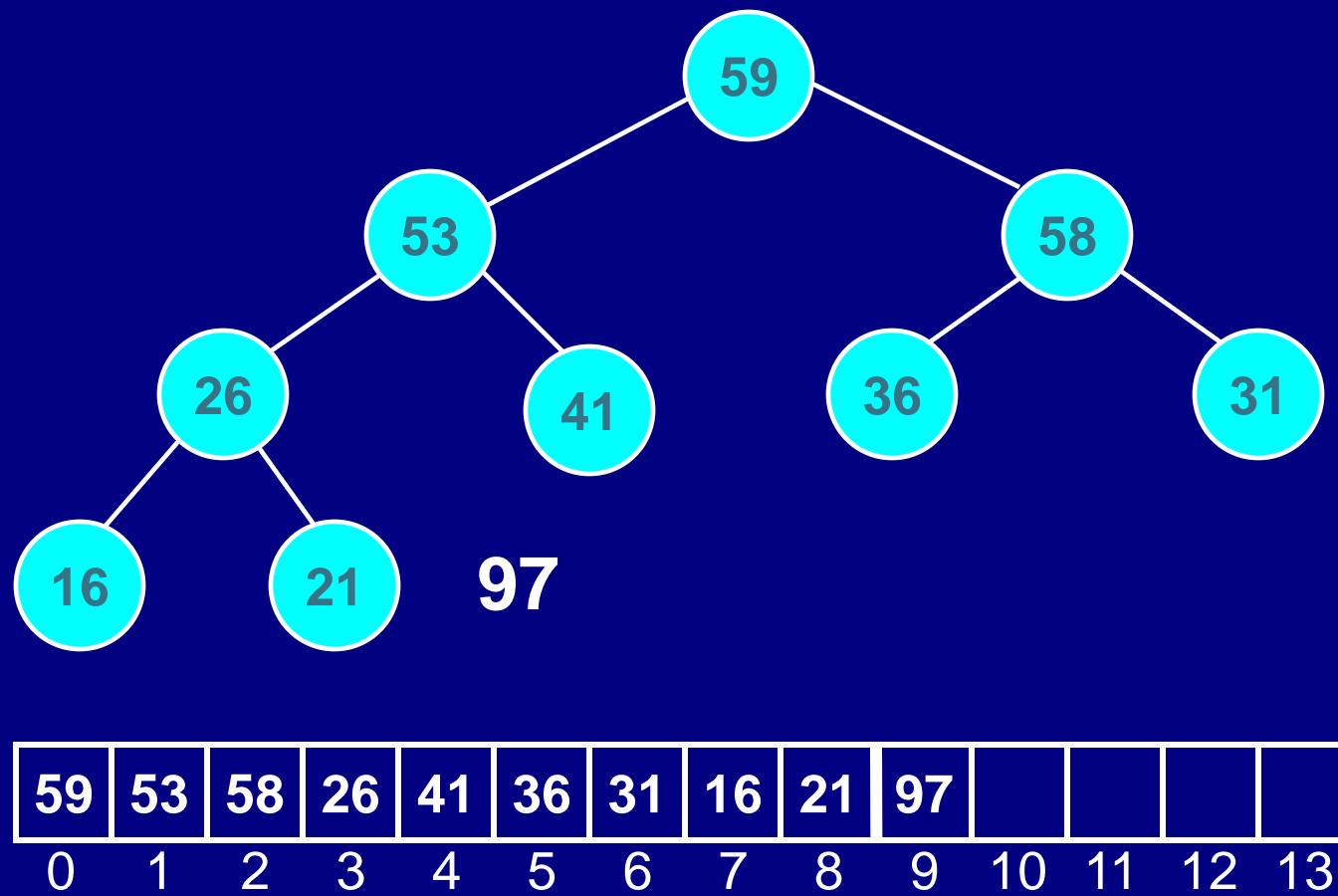
- Percolate down 0



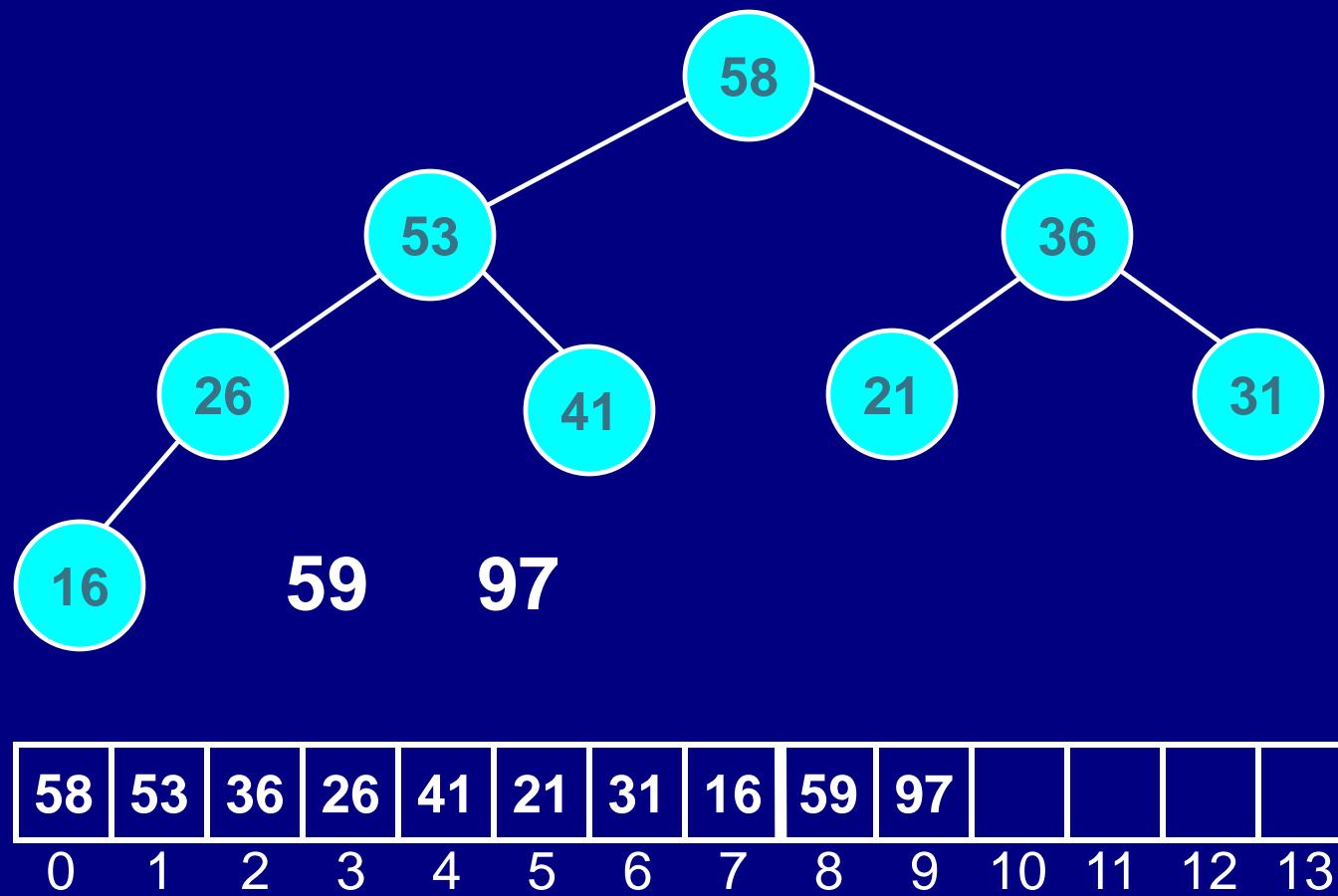
# Max Heap



# Heap setelah *deleteMax* pertama

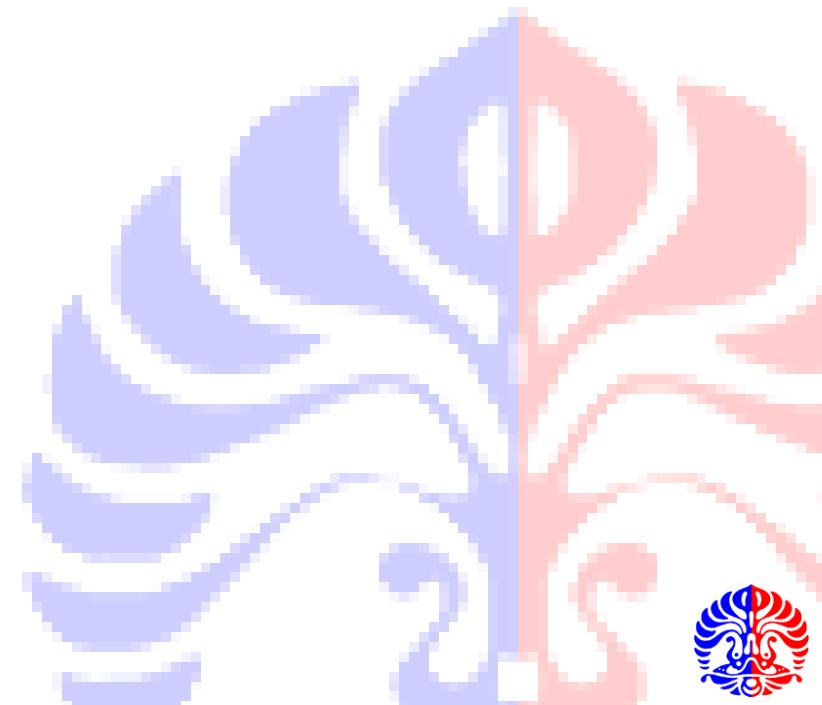


# Heap setelah *deleteMax* kedua



# Heap Sort

1. Buat sebuah heap tree
2. ambil elemen pada posisi root dari heap setiap pengambilan elemen, dan lakukan heapify.



# Rangkuman

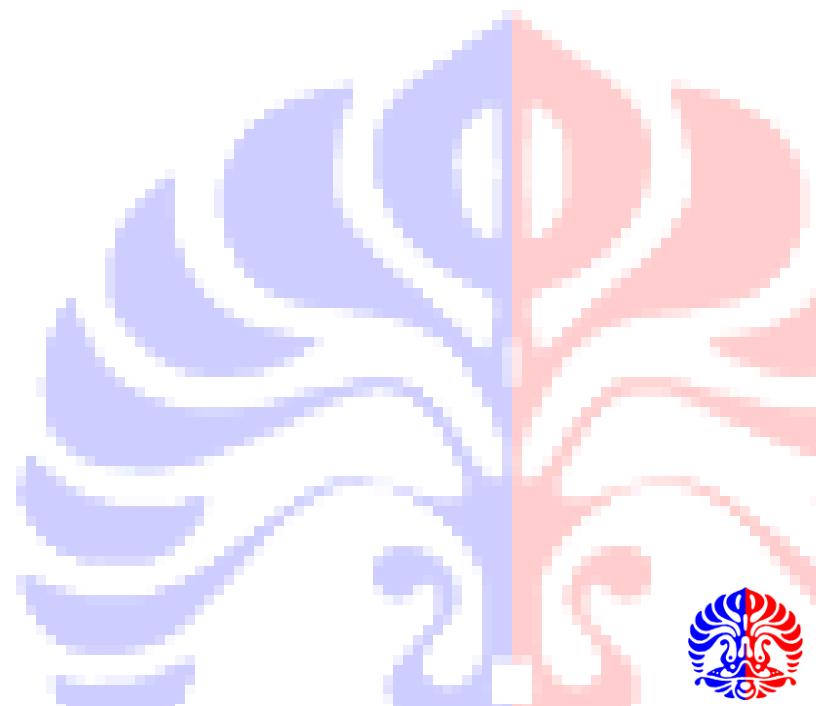
- Priority queue dapat diimplementasi kan menggunakan binary heap
- Aturan-aturan pada binary heap
  - structure property
    - complete binary tree
  - ordering property
    - Heap order: Parent  $\leq$  Child
- Operasi pada binary heap
  - insertion: kompleksitas waktu  $O(\log n)$  pada worst case
  - find min: kompleksitas waktu  $O(1)$
  - delete min: kompleksitas waktu  $O(\log n)$  pada worst case
- Binary heap dapat digunakan untuk *sorting*



# HUFFMAN CODE

# Outline

- Compression
- Huffman compression



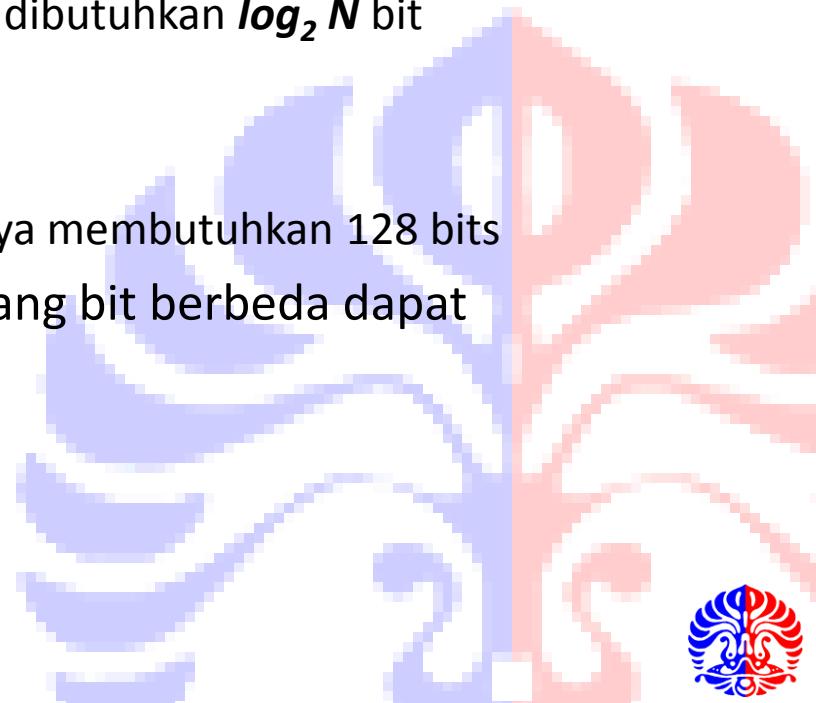
# Kompresi (*Compression*)

- Proses:
  - *Encoding*: raw → compressed
  - *Decoding*: compressed → raw
- Jenis Kompresi
  - *Lossy*: data yang dikompres kemungkinan tidak memiliki kualitas yang sama persis dengan data asli.
    - Contoh: MPEG, JPEG, MP3
  - *Lossless*: data yang dikompresi dapat dikembalikan menjadi data asli tanpa kehilangan kualitas data.
    - Contoh: zip, GIF, wav
- Beberapa algoritma kompresi:
  - RLE: Run Length Encoding
  - Lempel-Zif
  - Huffman Encoding
- Tingkat kompresi bergantung pada jenis file.



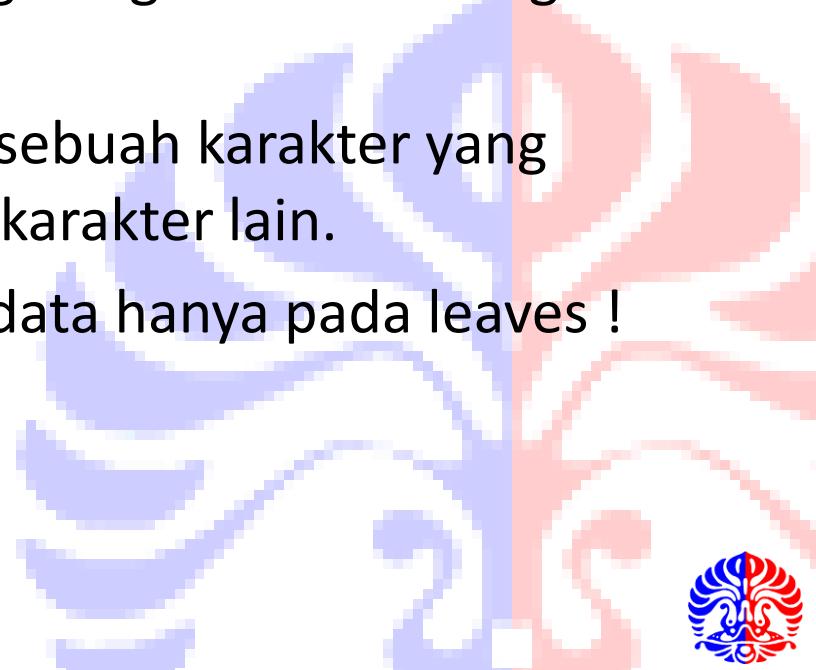
# Huffman Compression: Motivation

- Perhatikan teks berikut:  
If a woodchuck could chuck wood!
- Terdiri dari 32 karakter, masing-masing karakter diencode dengan 8 bit ASCII code. Untuk menyimpan kalimat diatas membutuhkan 256 bit.
  - $32 \text{ char} \times 8 \text{ bit} = 256 \text{ bits}$
- Teori:
  - Untuk meng-encode  $N$  karakter yang berbeda dibutuhkan  $\log_2 N$  bit
- Observasi:
  - Ada 13 karakter berbeda  $\rightarrow 4$  bit
  - Kalimat diatas dapat dikompres sehingga hanya membutuhkan 128 bits
- Apakah menggunakan encoding dengan panjang bit berbeda dapat memperbaiki tingkat kompresi ?



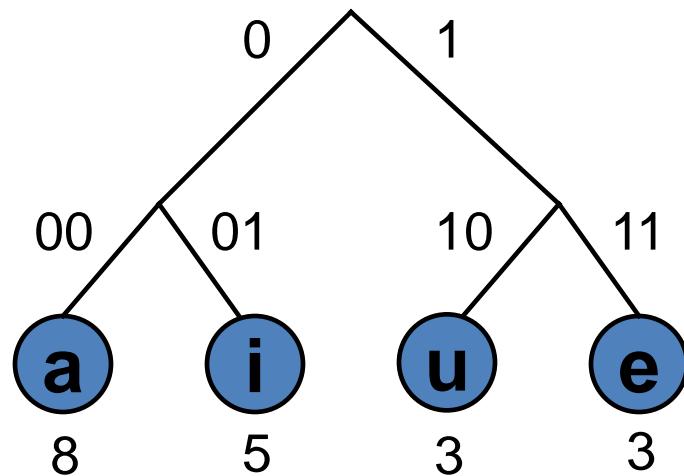
# Huffman Compression: Ide

- Untuk karakter yang sering muncul, gunakan representasi encoding yang pendek.
- Untuk karakter yang jarang muncul, gunakan representasi encoding yang panjang.
- Bila panjang bit encoding tiap karakter beda-beda, bagaimana memisahkan sekumpulan bit encoding dengan bit encoding karakter lain?
- Prefix code: Tidak ada code encoding sebuah karakter yang merupakan prefix dari code encoding karakter lain.
- Prefix code dimodelkan dengan tree, data hanya pada leaves !



# Huffman Encoding: perbandingan

- Menggunakan encoding dengan panjang bits sama.
- Ada 4 karakter, masing-masing di encode dengan 2 bits.
- Karakter **a** muncul 8 kali, **i** 5 kali, **u** dan **e** masing-masing 3 kali.
- Panjang bit yang dibutuhkan: **42 bit**



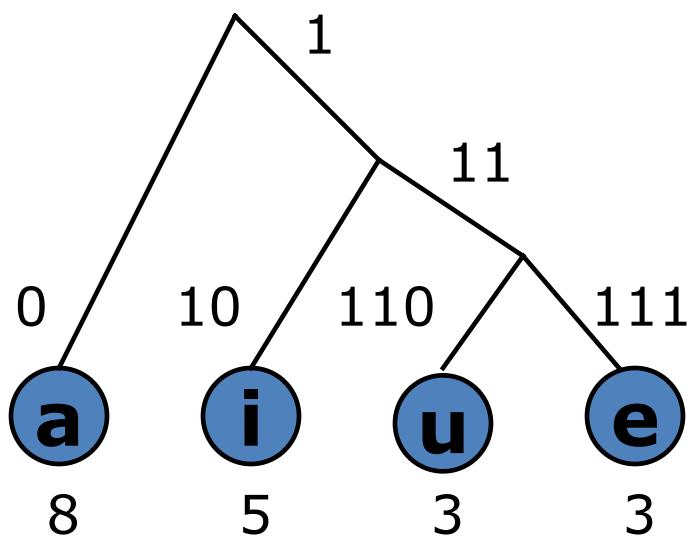
$a = 00$	$\rightarrow 16 \text{ bits}$
$i = 01$	$\rightarrow 10 \text{ bits}$
$u = 10$	$\rightarrow 6 \text{ bits}$
$e = 11$	$\rightarrow 6 \text{ bits}$

**Total : 38 bits**



# Huffman Encoding: perbandingan

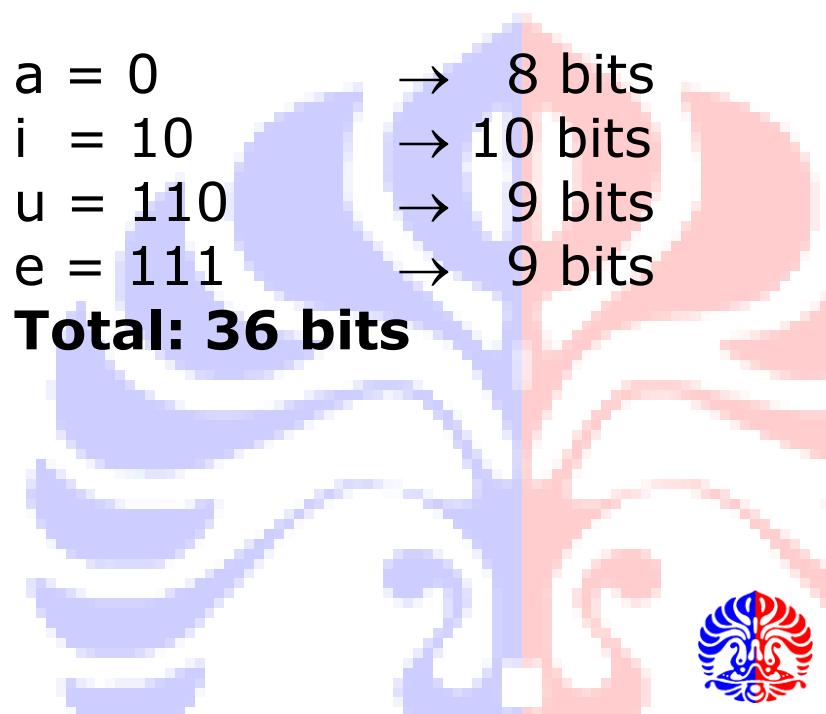
- Menggunakan encoding dengan panjang bits berbeda-beda.
- Karakter **a** di encode dengan binary code: **0**,  
**i** dengan **10**, **u** dengan **110**, dan **e** dengan **111**.
- Panjang bit yang dibutuhkan: **36 bit**
- **Kesimpulan: lebih baik dari pada encoding yang fixed length.**



$a = 0$   
 $i = 10$   
 $u = 110$   
 $e = 111$

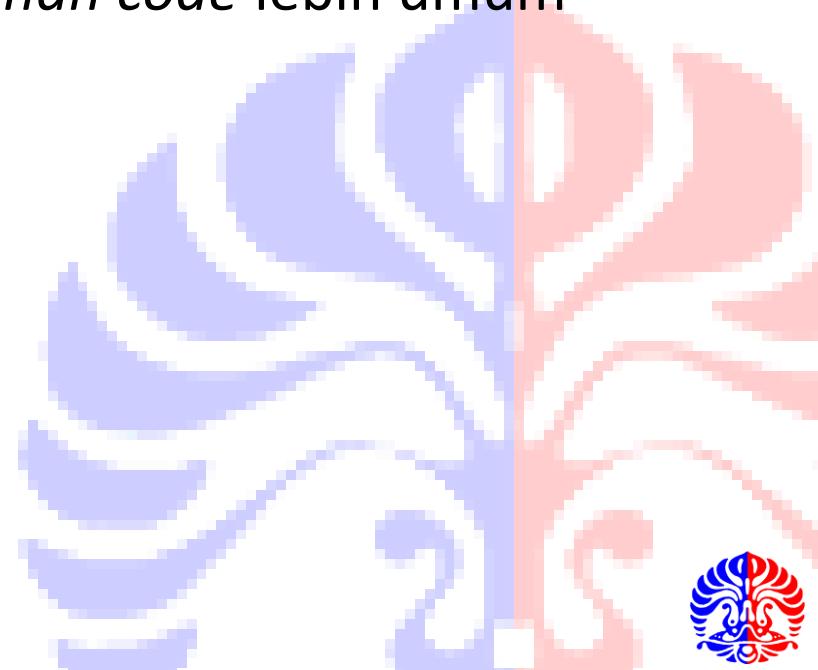
→ 8 bits  
→ 10 bits  
→ 9 bits  
→ 9 bits

**Total: 36 bits**

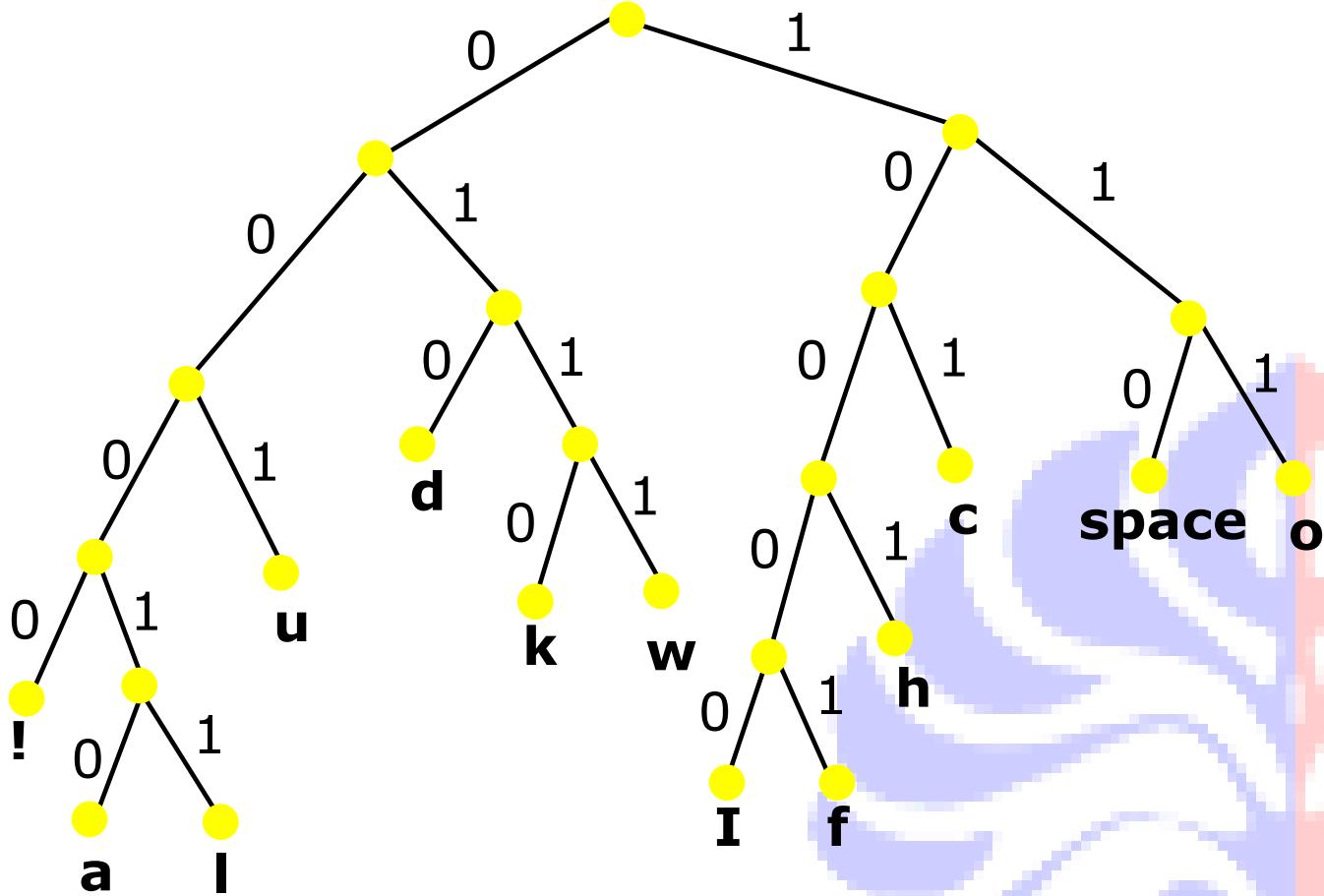


# Huffman Code

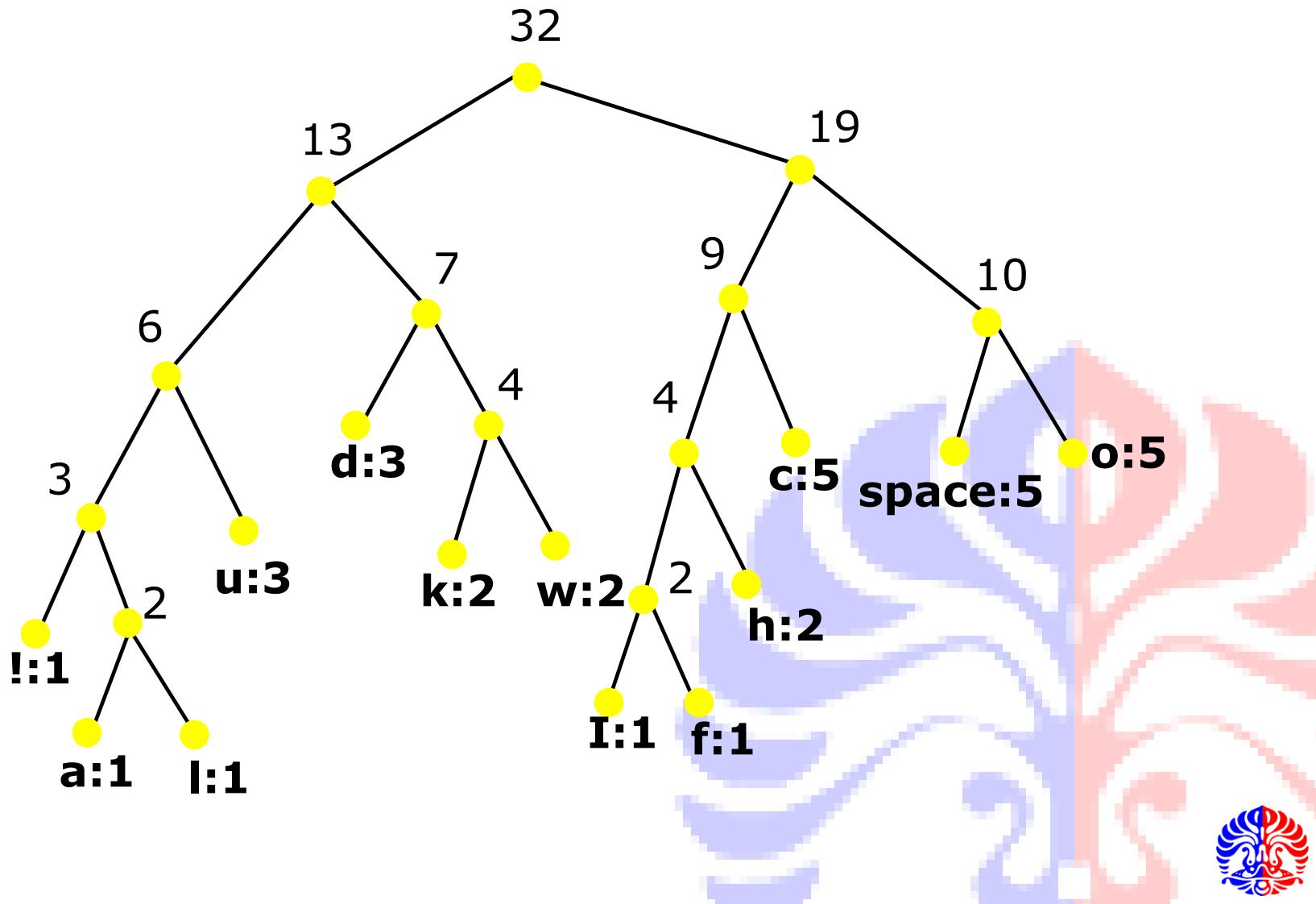
- Algoritma Huffman digunakan untuk menggenerate prefix code.
- Prefix code: Tidak ada code encoding sebuah karakter yang merupakan prefix dari code encoding karakter lain.
- Prefix code dimodelkan dengan tree, data hanya pada leaves !
- Walau algoritma Huffman hanya salah satu algoritma untuk menghasilkan prefix code, istilah *huffman code* lebih umum dikenal dari pada prefix code.



# Huffman Encoding



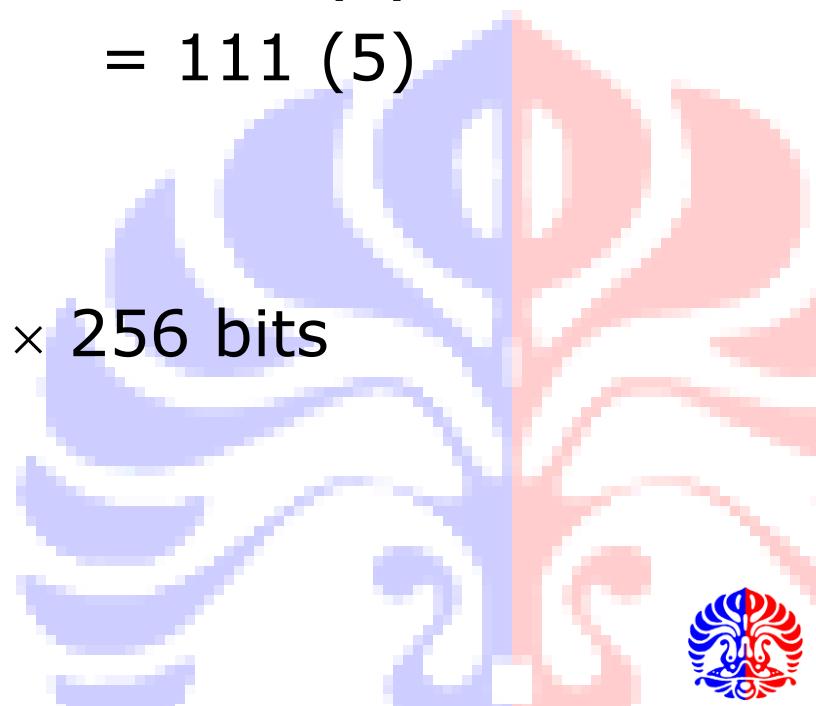
# Huffman Encoding



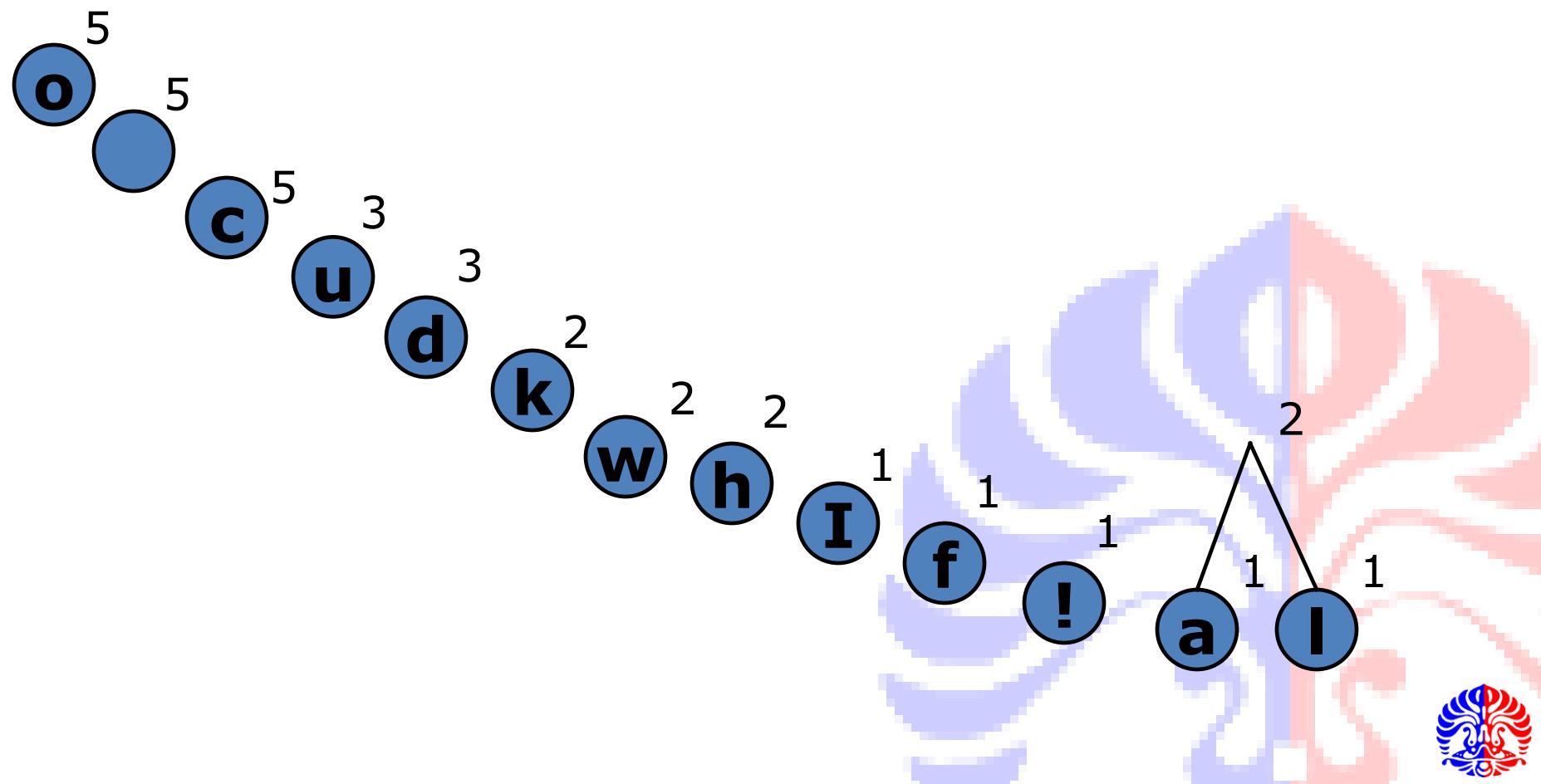
# Huffman Encoding (freq)

! = 0000 (1)	I	= 10000 (1)
a = 00010 (1)	f	= 10001 (1)
l = 00011 (1)	h	= 1001 (2)
u = 001 (3)	c	= 101 (5)
d = 010 (3)	space	= 110 (5)
k = 0110 (2)	o	= 111 (5)
w = 0111 (2)		

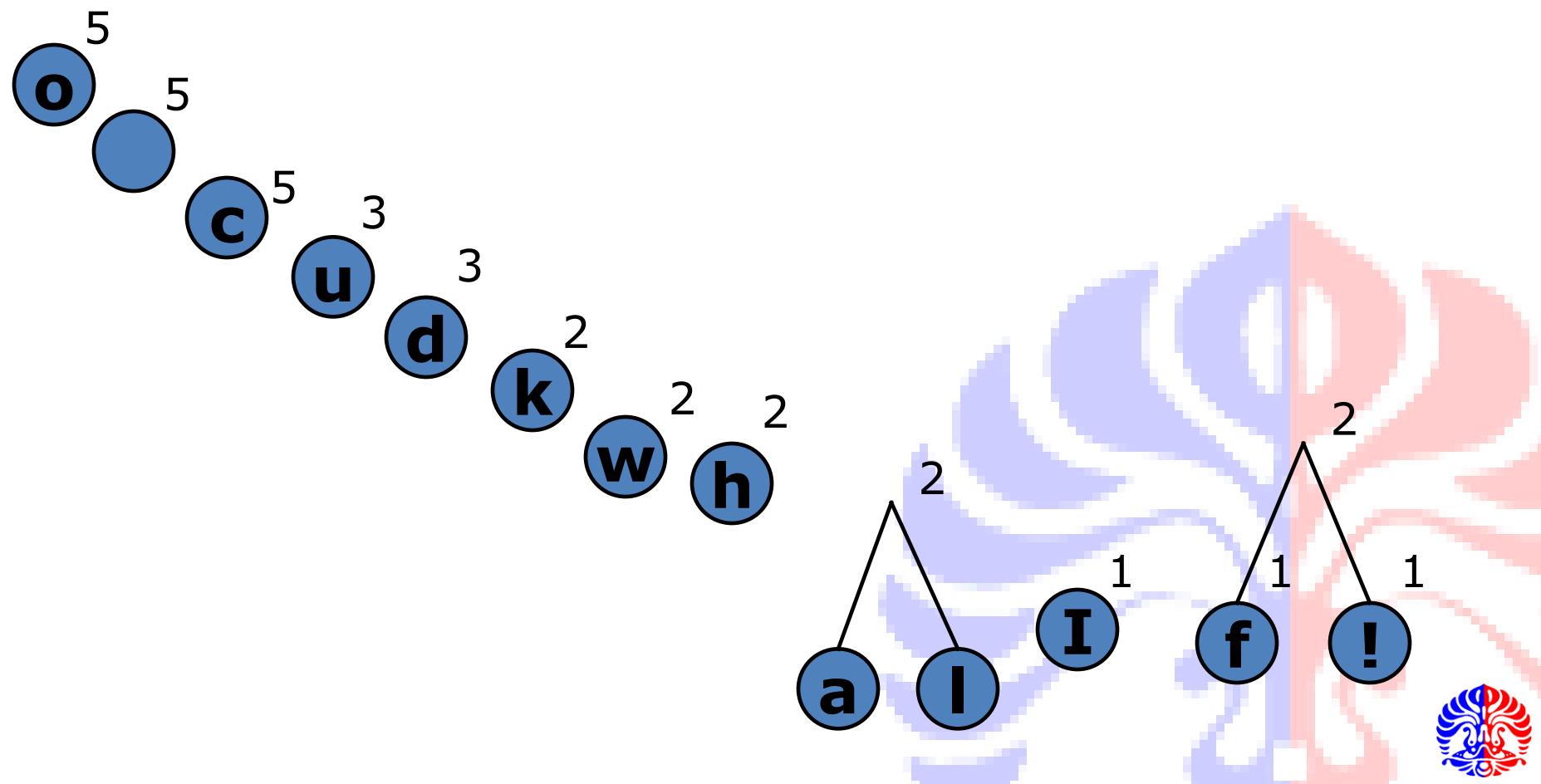
Cost:  $\sum d_i * f_i = 111 \text{ bits} = 44\% \times 256 \text{ bits}$



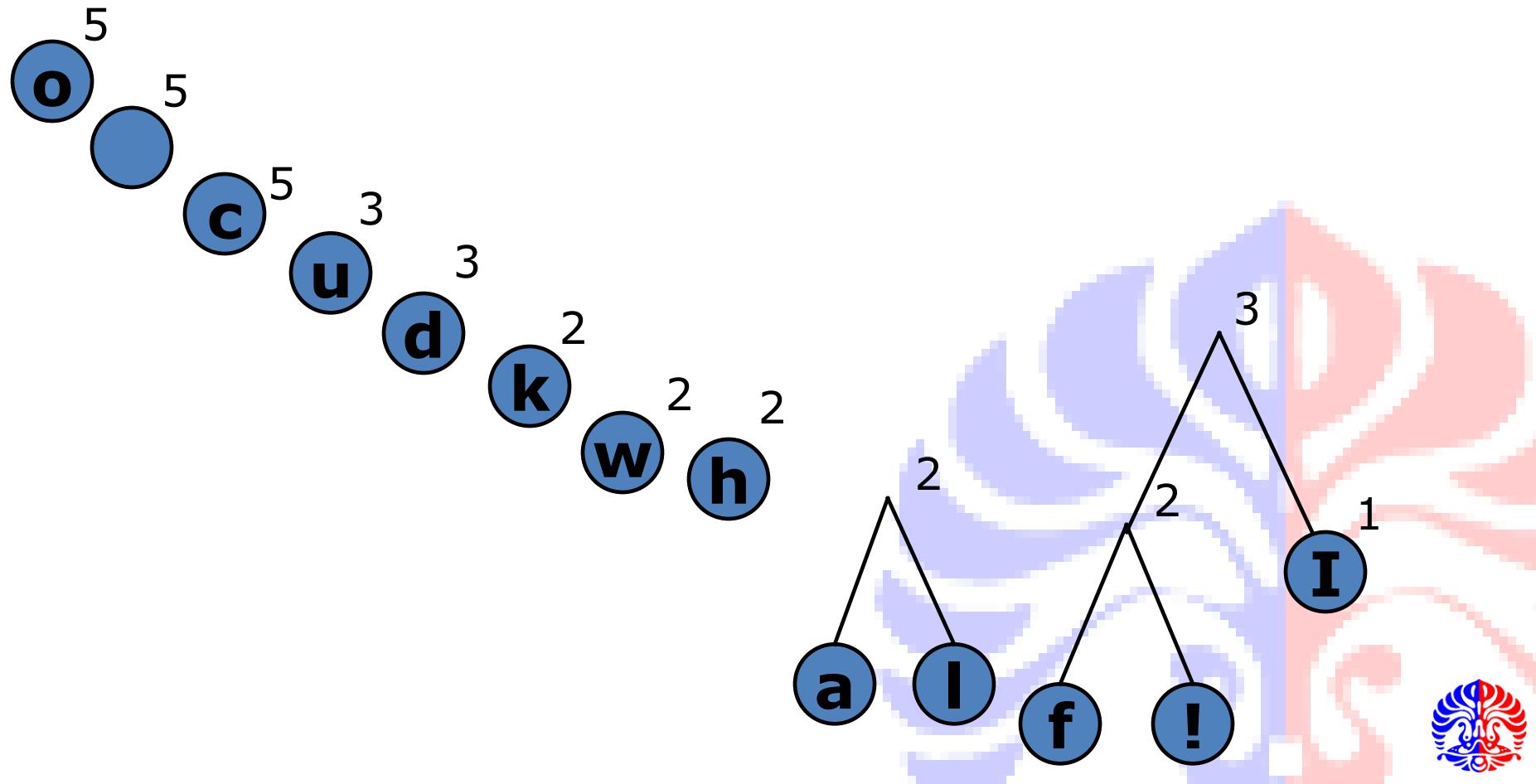
# Huffman Encoding: langkah-langkah



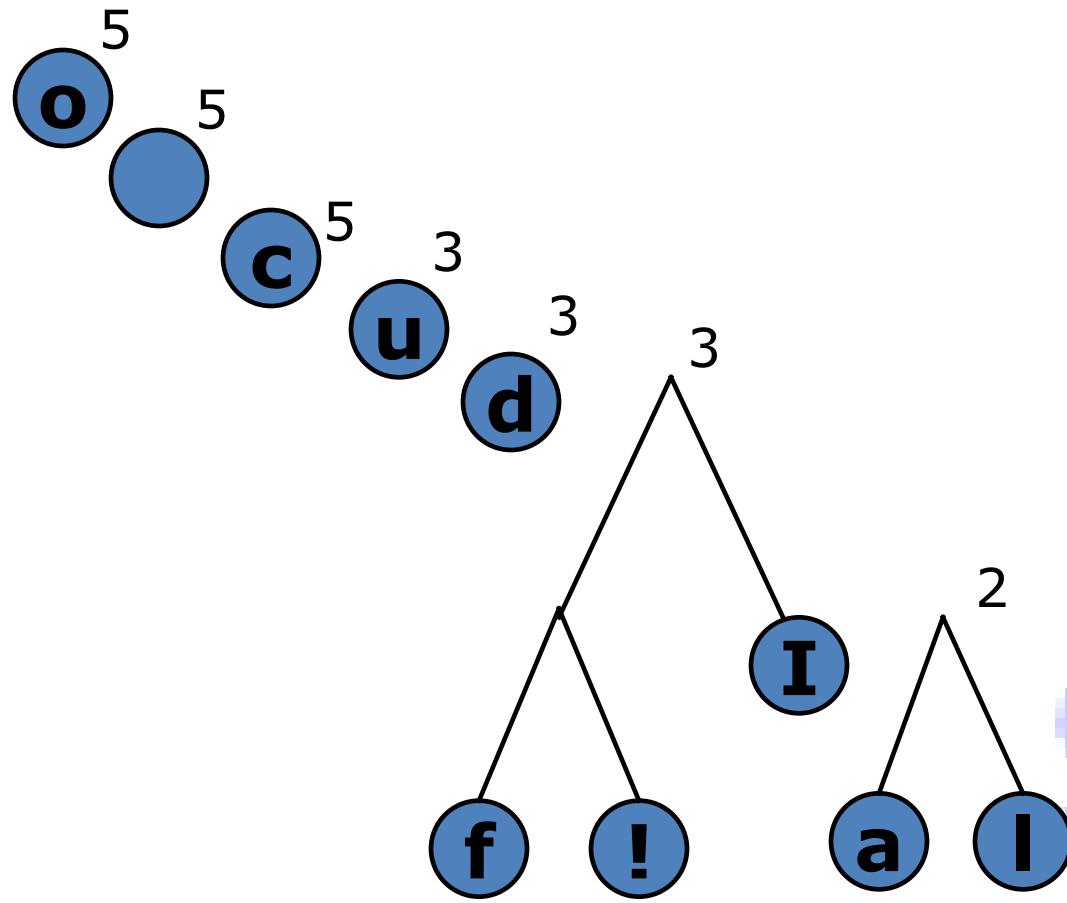
# Huffman Encoding: langkah-langkah



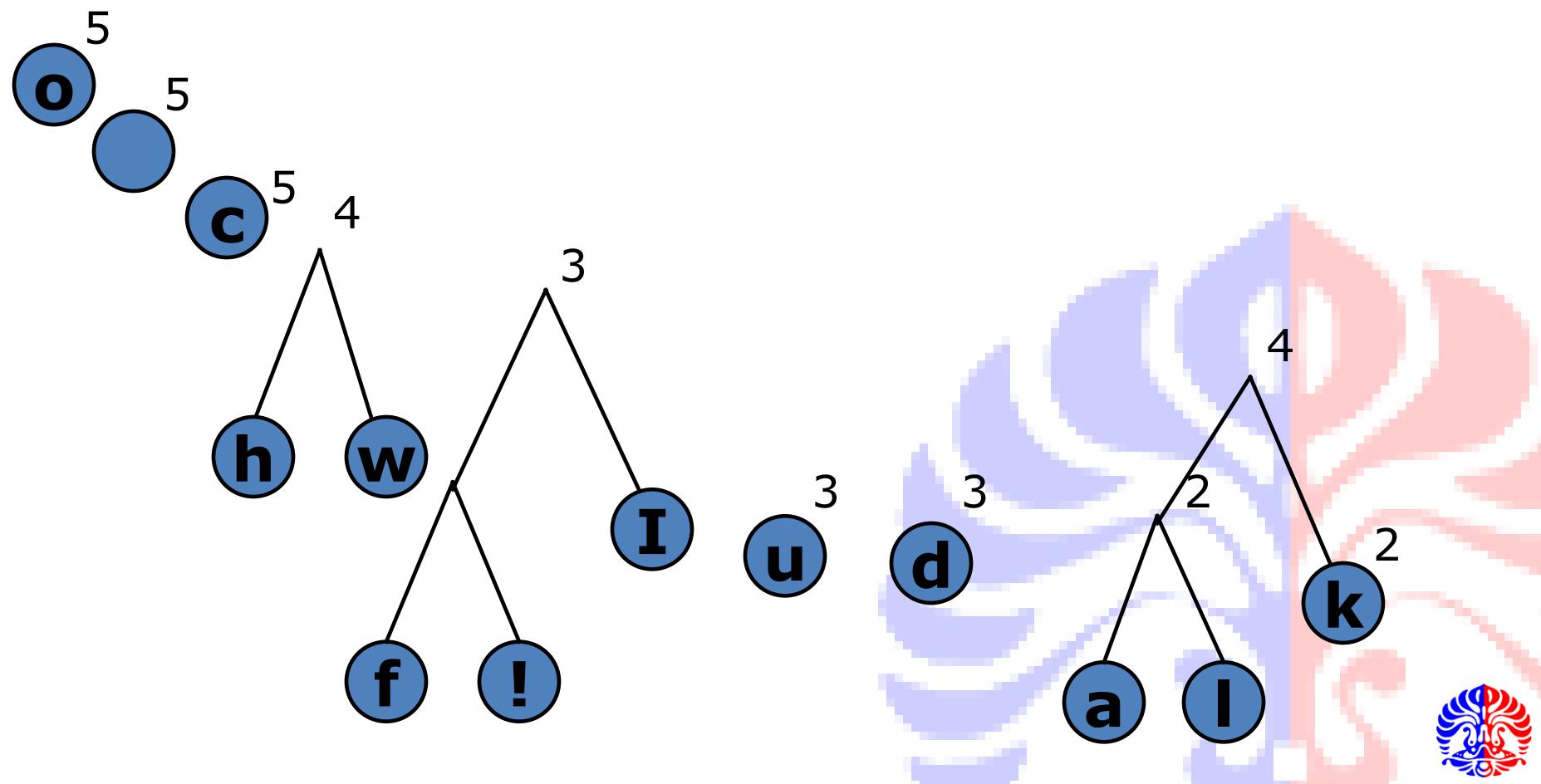
# Huffman Encoding: langkah-langkah



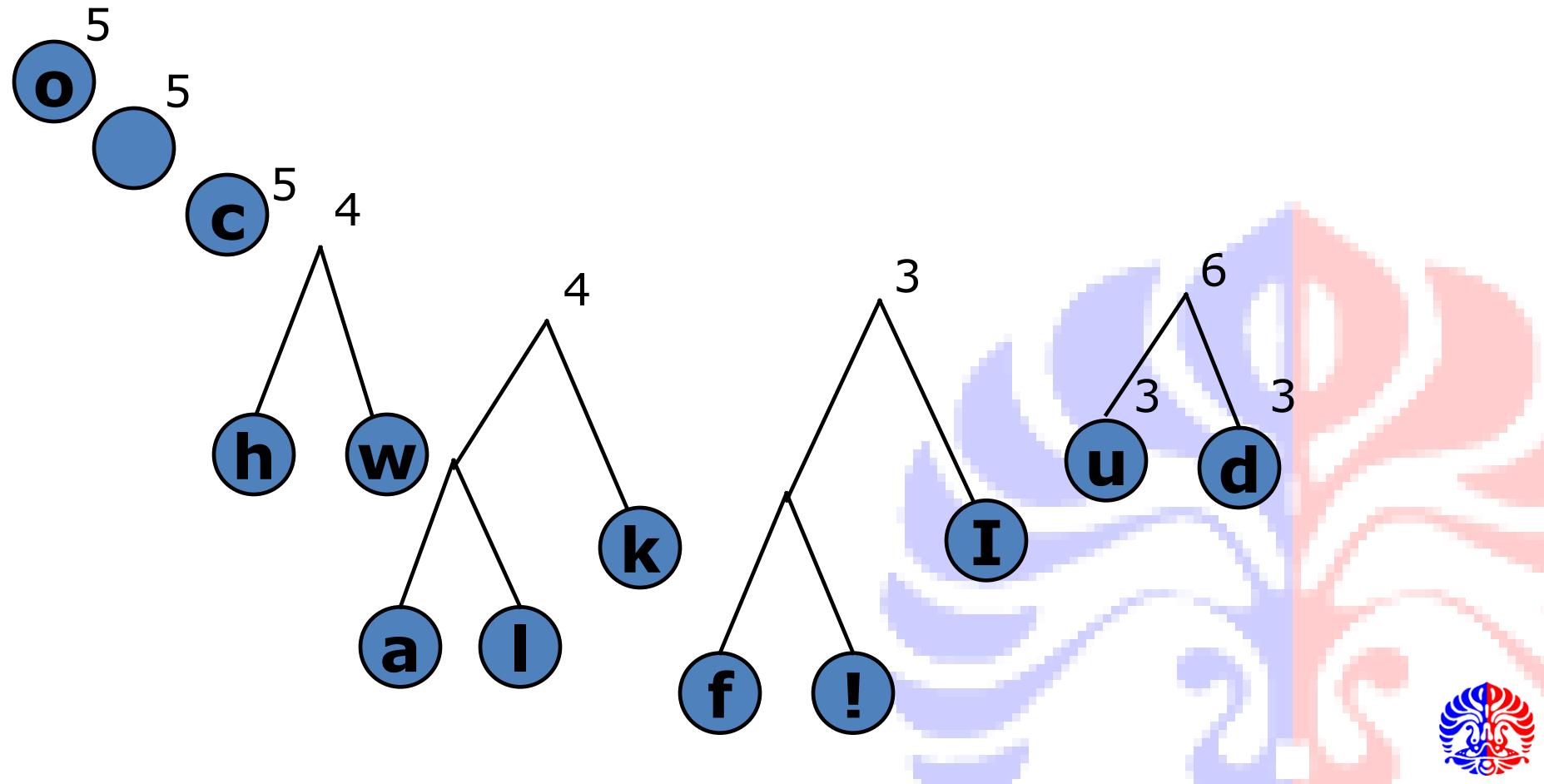
# Huffman Encoding: langkah-langkah



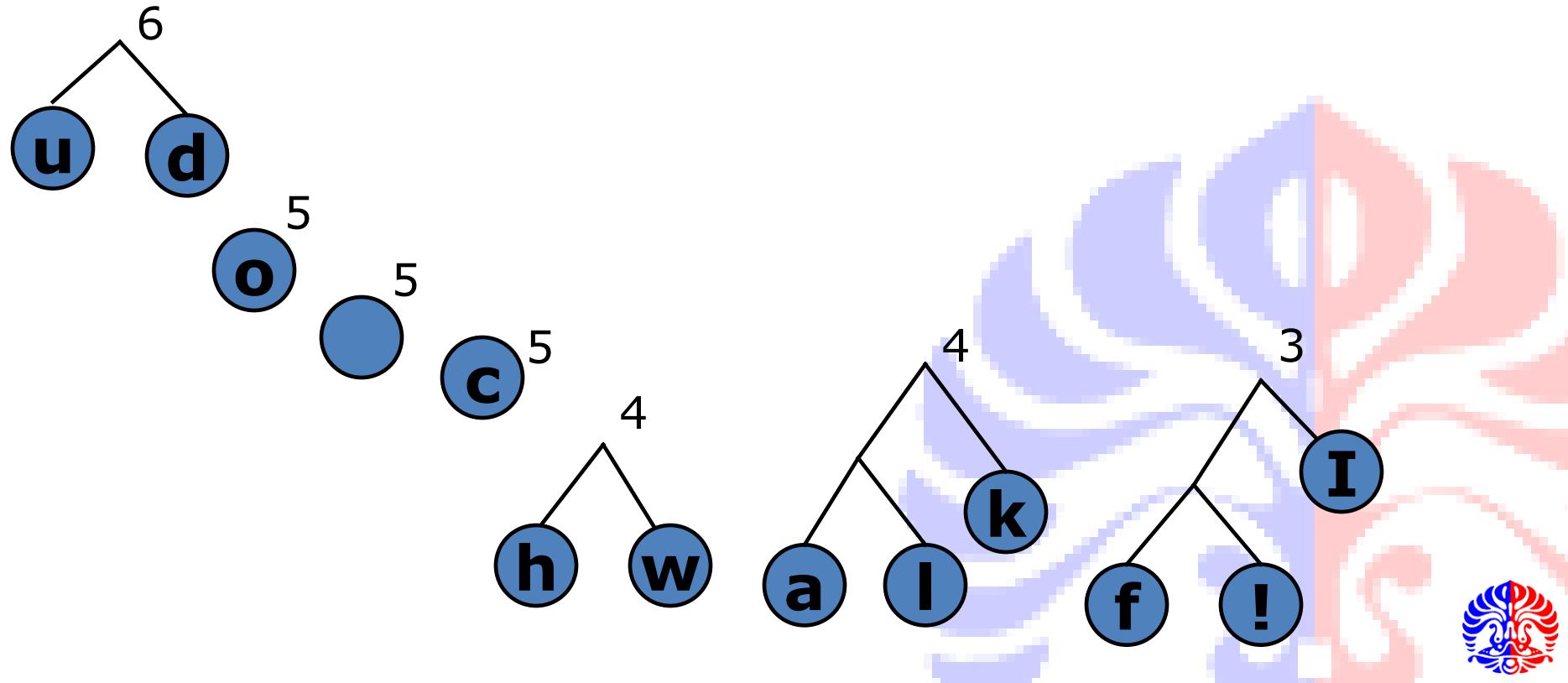
# Huffman Encoding: langkah-langkah



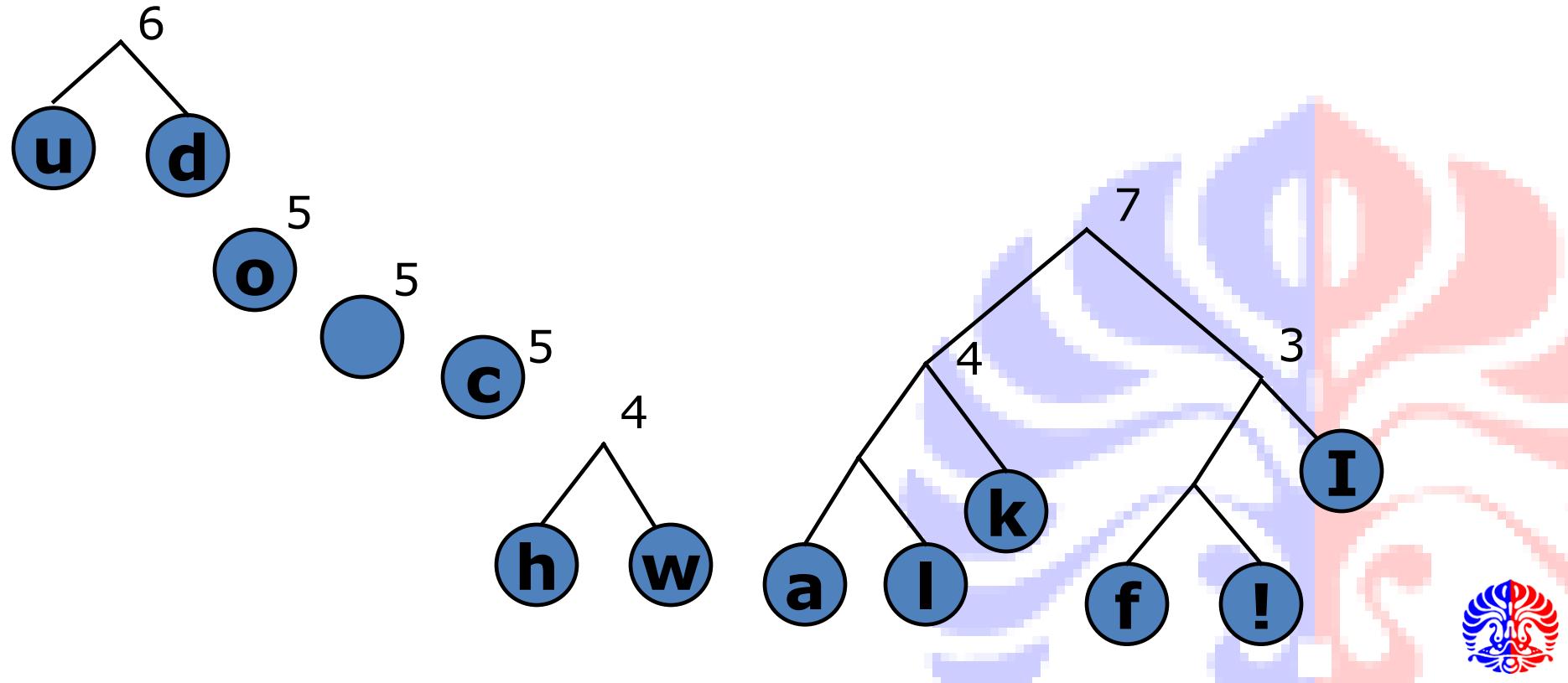
# Huffman Encoding: langkah-langkah



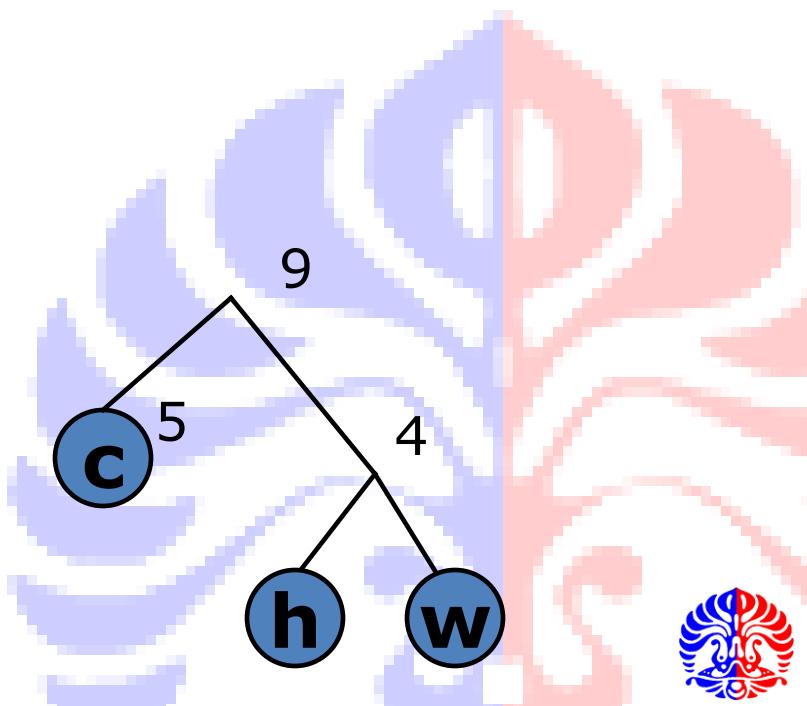
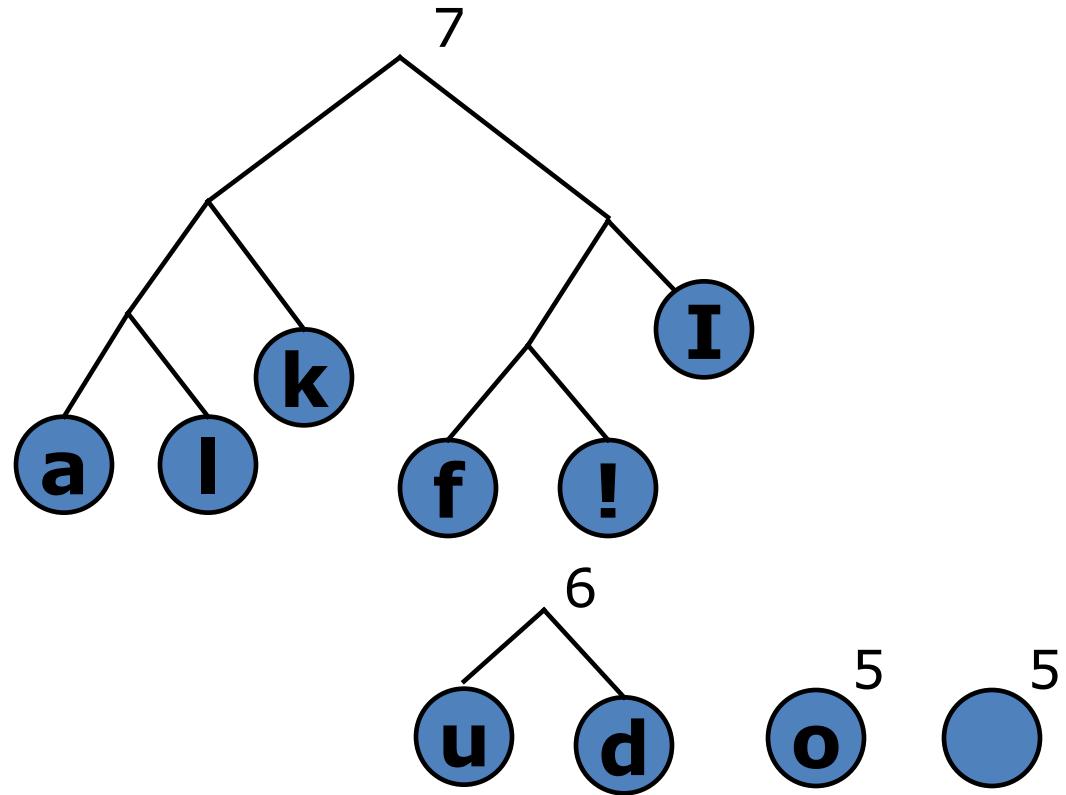
# Huffman Encoding: langkah-langkah



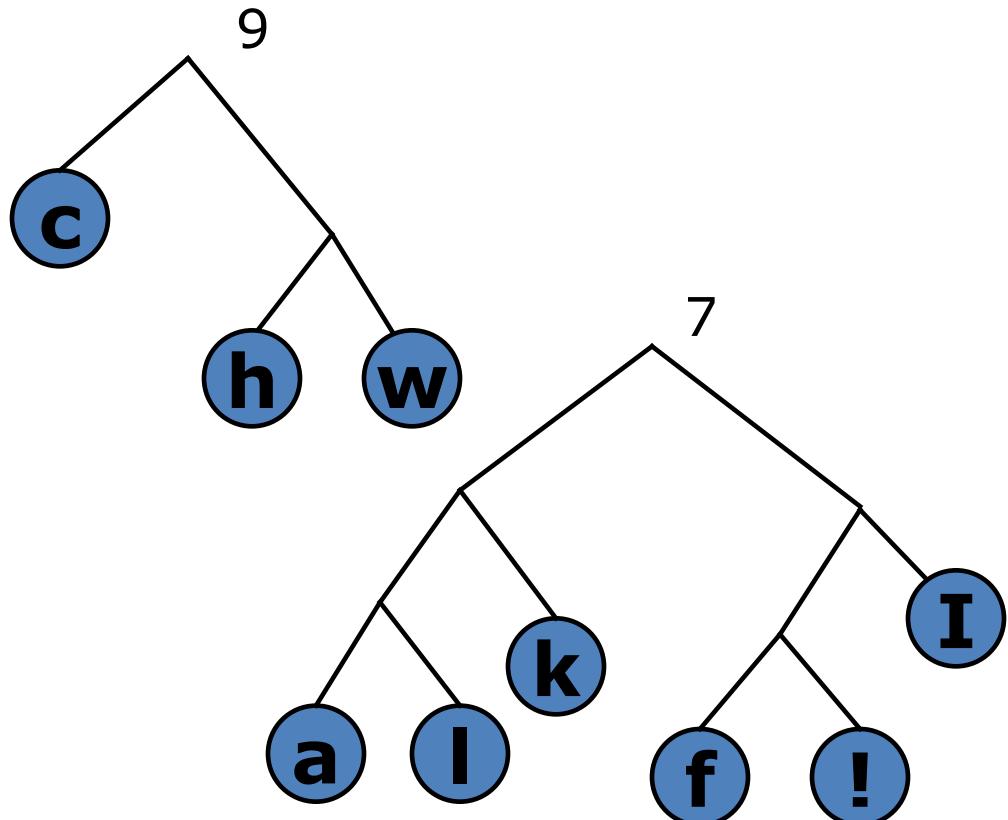
# Huffman Encoding: steps



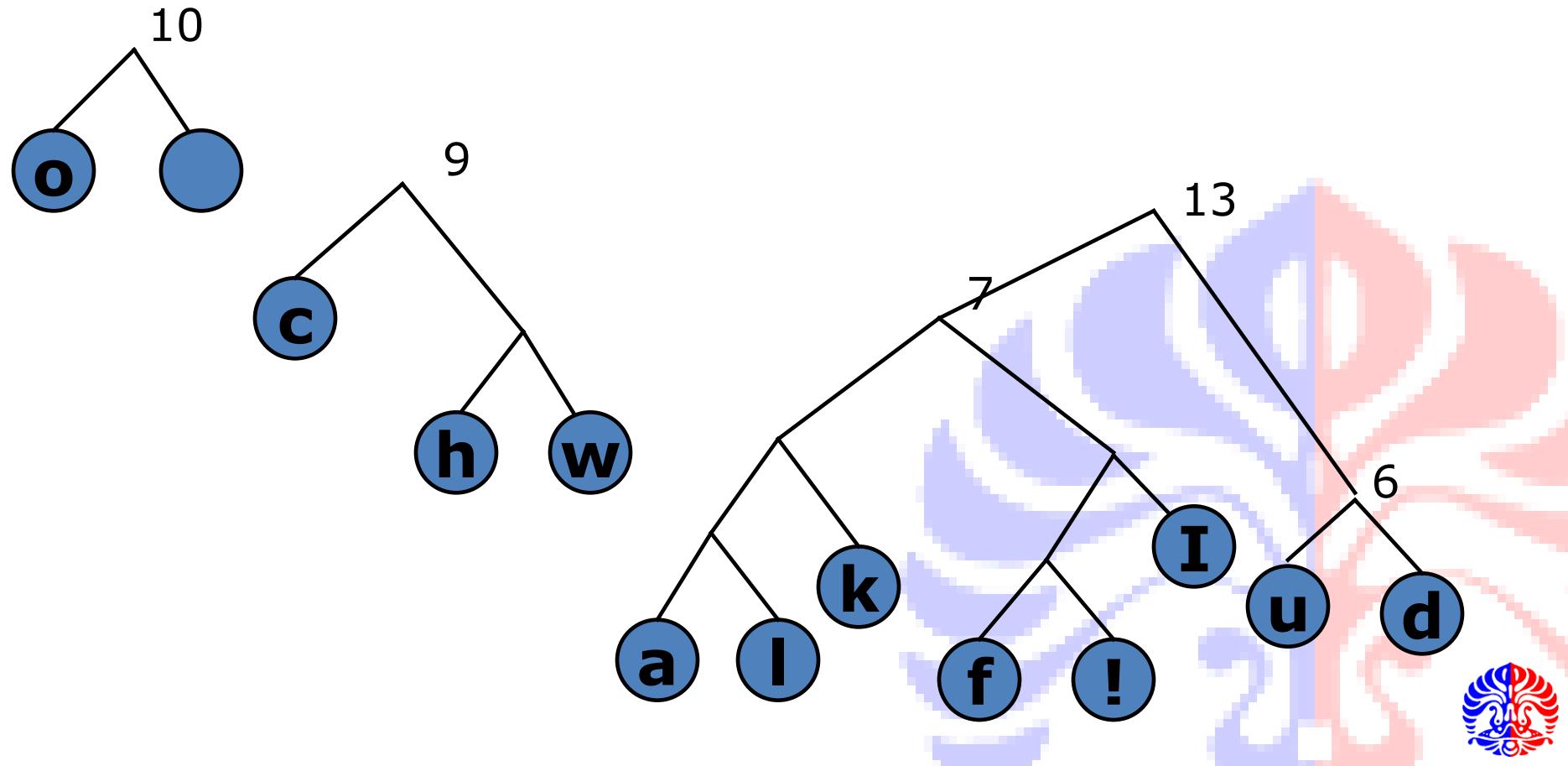
# Huffman Encoding: langkah-langkah



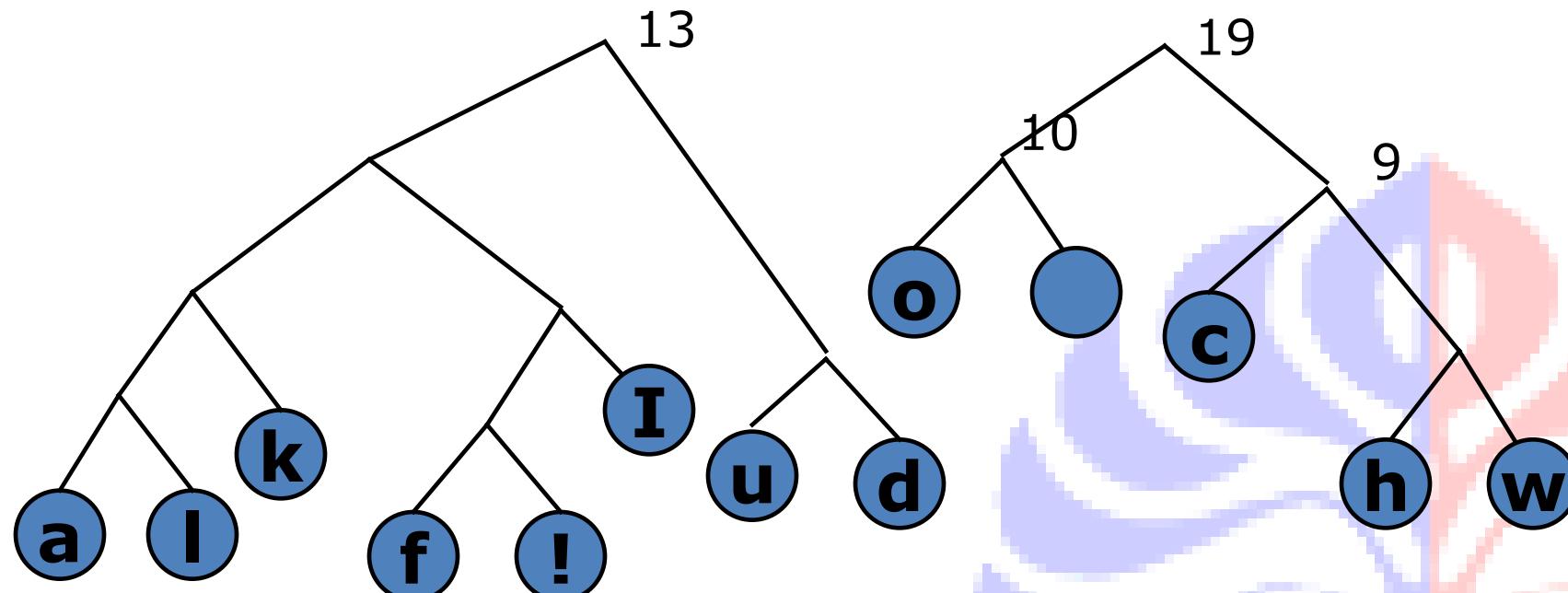
# Huffman Encoding: langkah-langkah



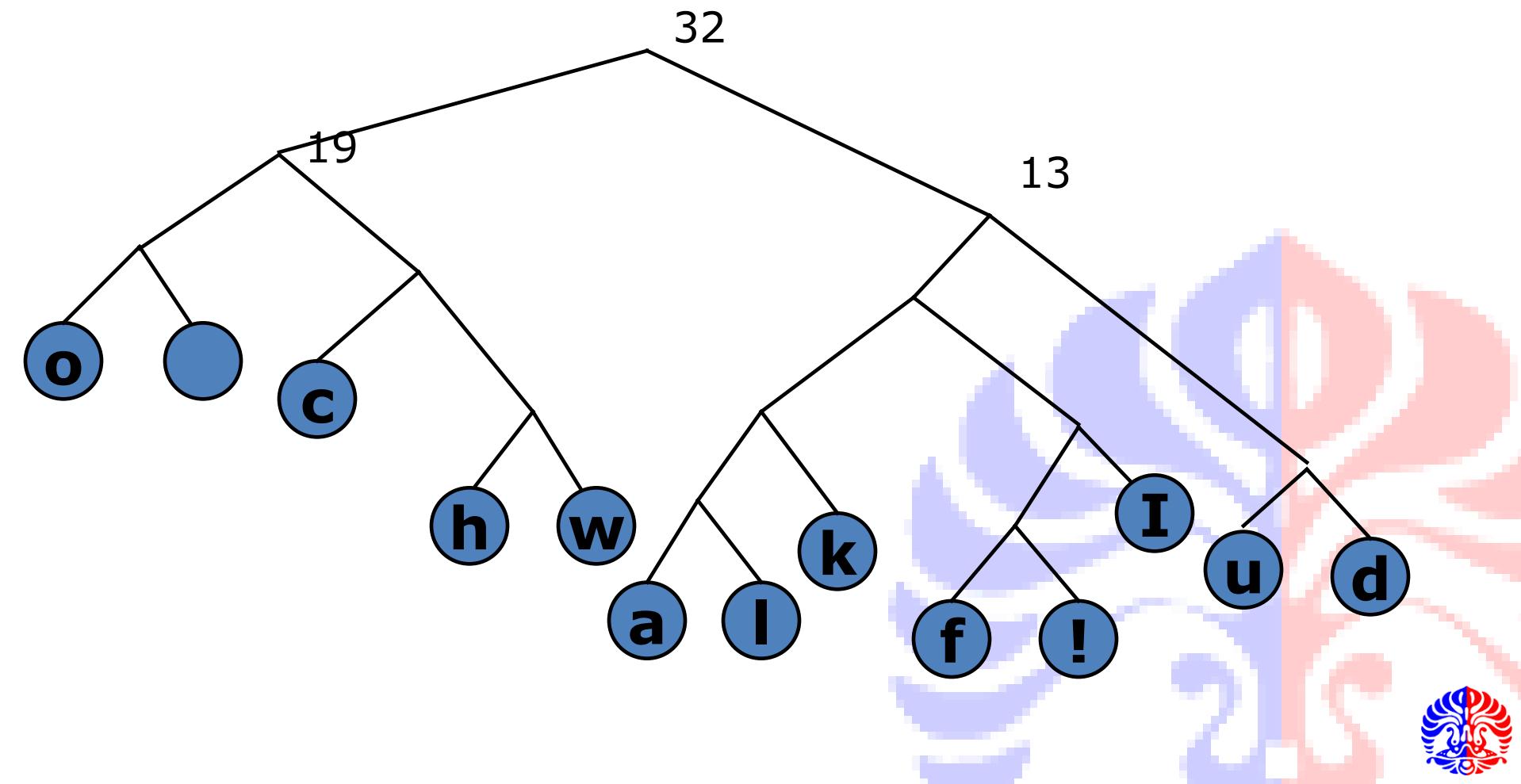
# Huffman Encoding: langkah-langkah



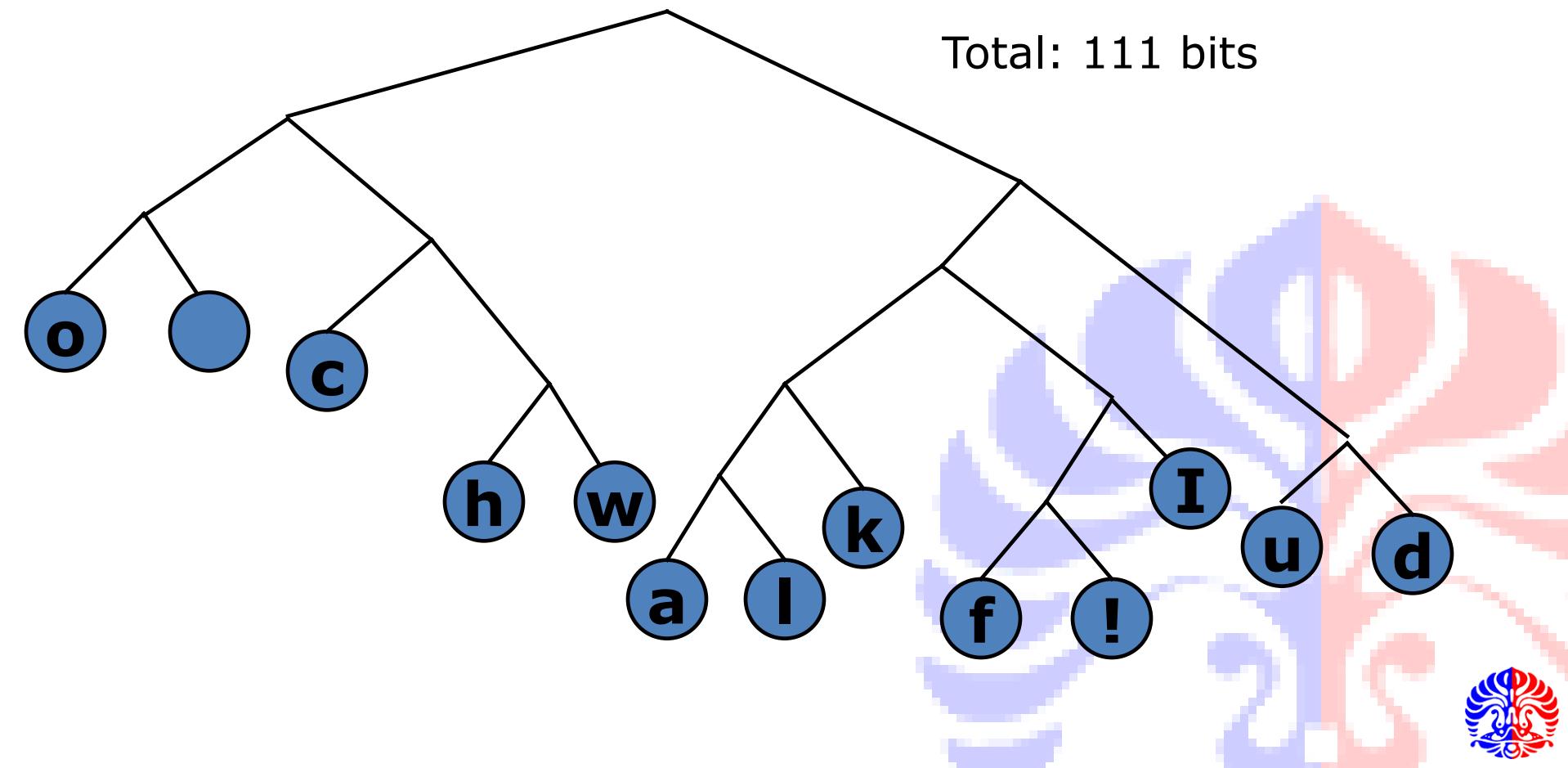
# Huffman Encoding: langkah-langkah



# Huffman Encoding: langkah-langkah

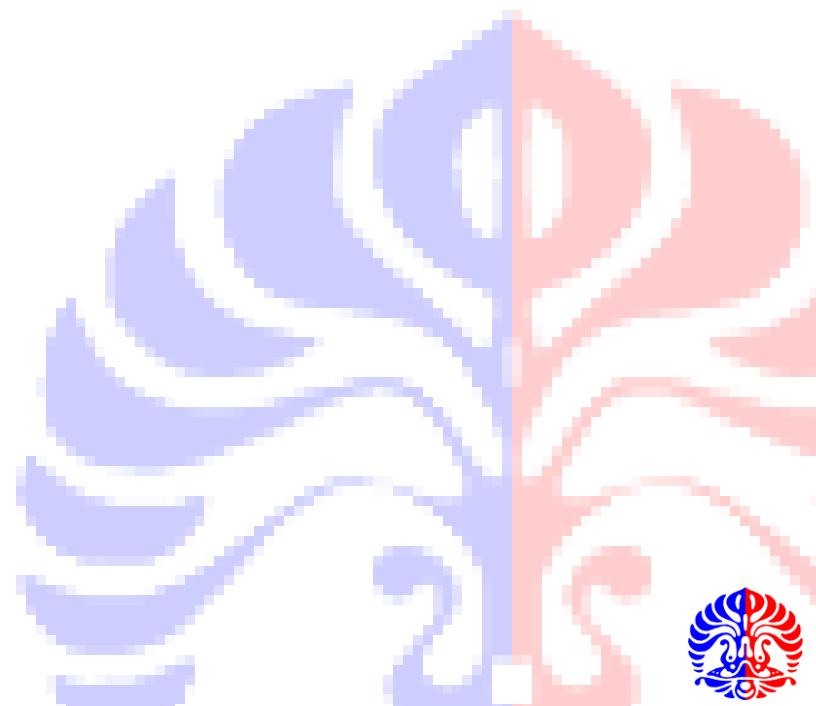


# Huffman Encoding: langkah-langkah



# Rangkuman

- Huffman encoding menggunakan informasi frekuensi kemunculan untuk meng-kompres file.
- Karakter dengan kemunculan paling tinggi di berikan encoding yang paling pendek dan sebaliknya.
- Penentuan code encoding menggunakan representasi binary tree.



# IKI10400 • Struktur Data & Algoritma: *Graph*

**Fakultas Ilmu Komputer • Universitas Indonesia**

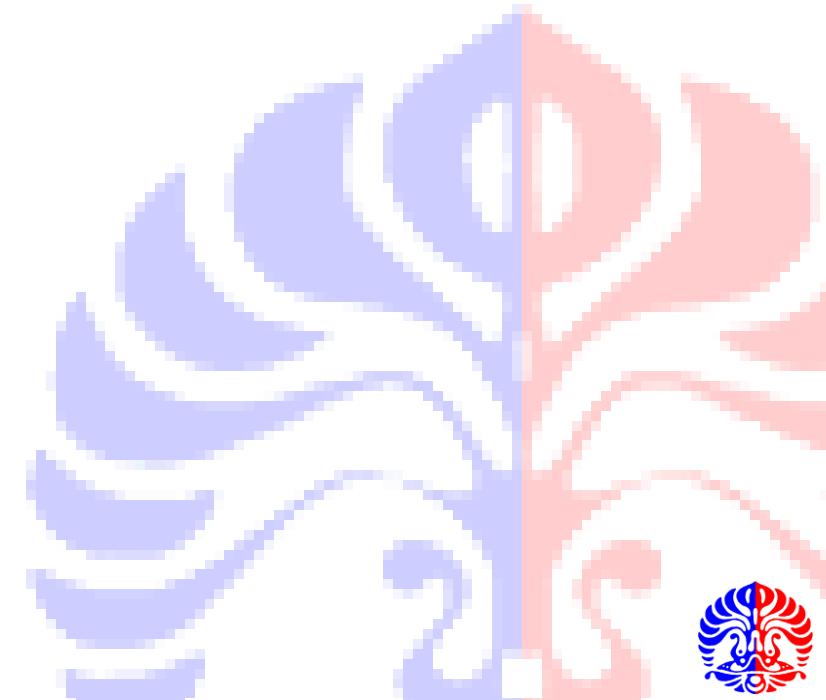
*Slide acknowledgments:*

Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung, Tisha Melia,  
Clara Vania



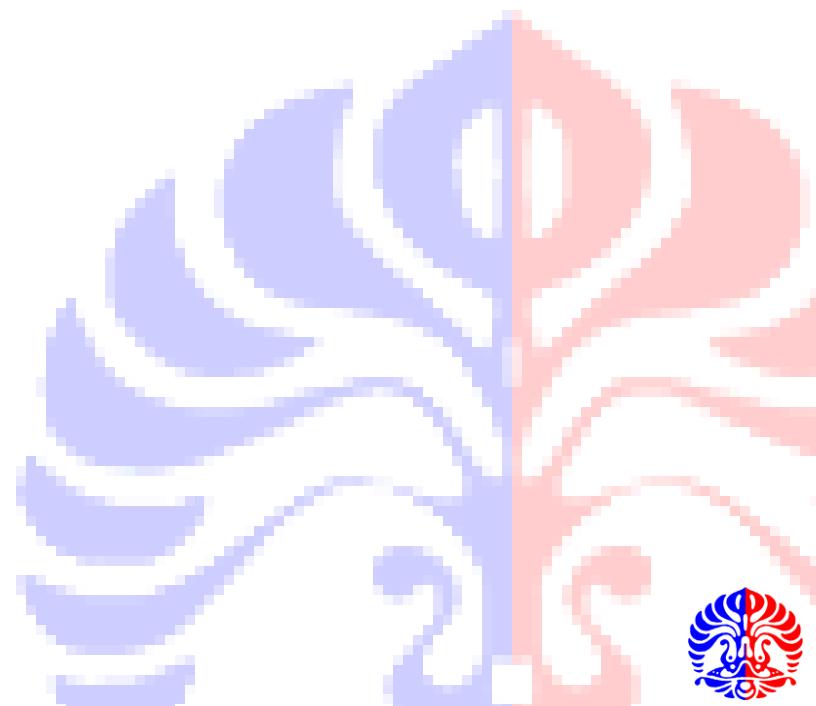
# Materi

- Motivasi
- Definisi dan Istilah
- Representasi Graph
- Algoritma mencari shortest path
- Topological Sort
- Minimum spanning tree
  - Prim's Algoritma
  - Kruskal's Algoritma



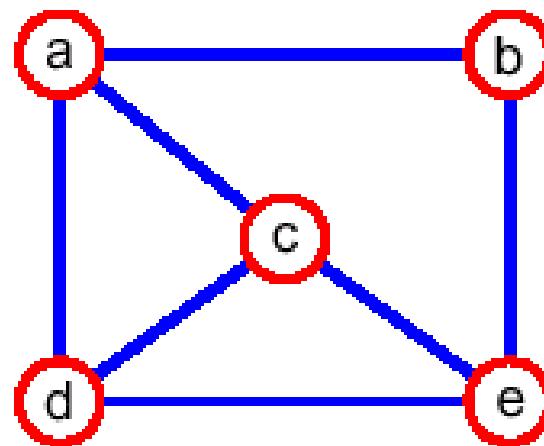
# Penggunaan Graph

- Jaringan
- Peta
  - Mencari jalur terpendek
- Penjadwalan (Perencanaan Proyek)



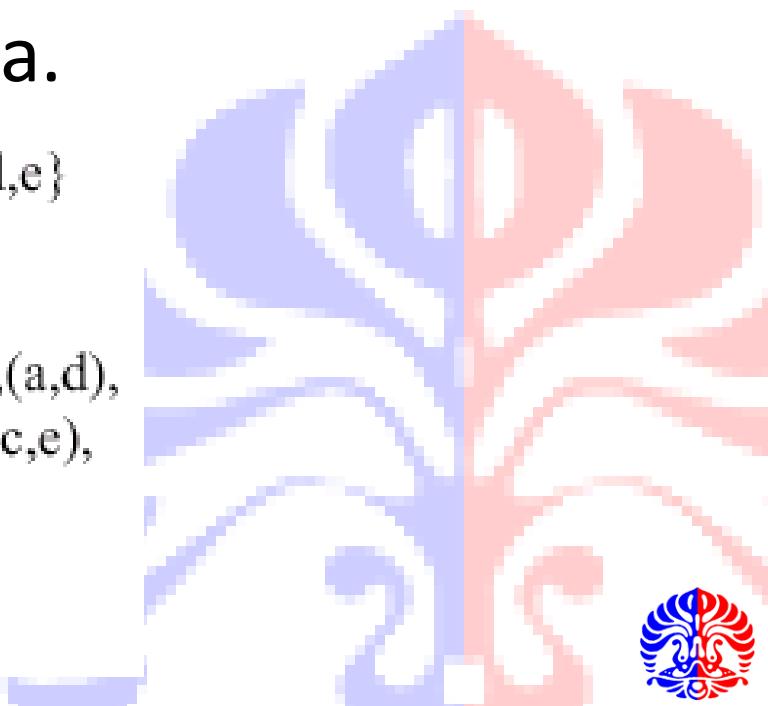
# Definisi

- Sebuah graph  $G = (V, E)$  terdiri dari:
  - $V$ : kumpulan simpul (*vertices/nodes*)
  - $E$ : kumpulan *sisi/busur (edge)* yang menghubungkan simpul-simpul.
- Sebuah sisi  $e = (a, b)$  memiliki informasi dua simpul yang dihubungkannya.



$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$$



# Istilah

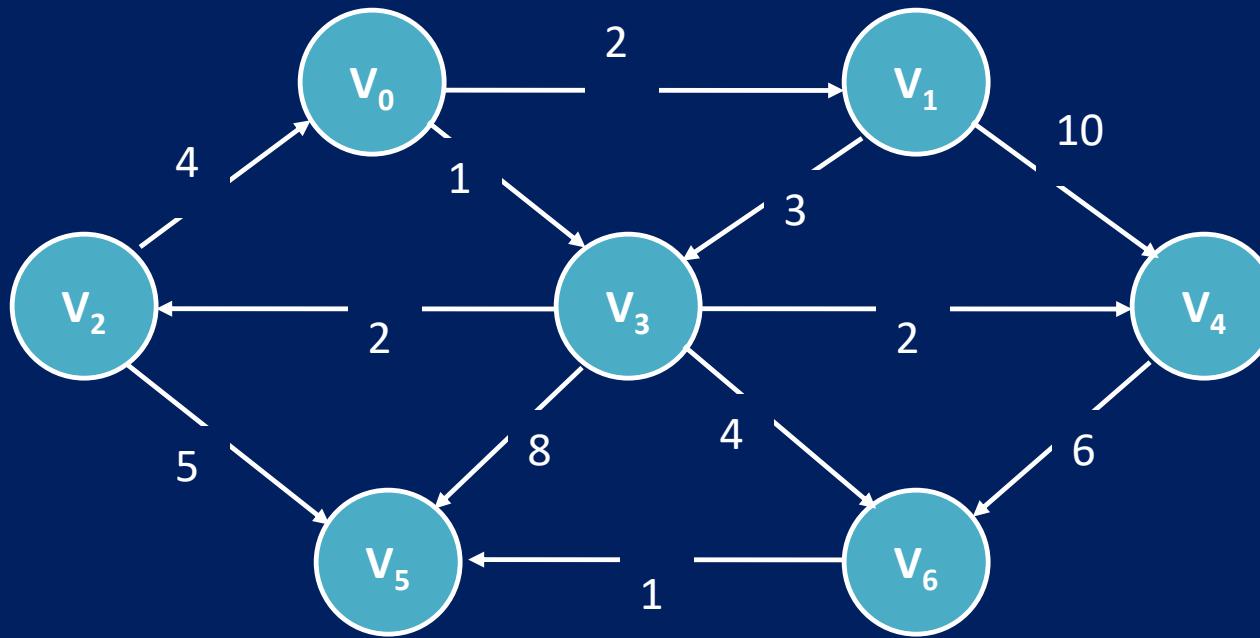
- **undirected graph**
- **directed graph**
- **adjacent vertices**: adalah simpul-simpul yang dihubungkan oleh sebuah sisi (*edge*)
- **degree** (of a vertex): adalah jumlah simpul lain yang terhubung langsung melalui sebuah sisi.
  - Untuk kategori directed graph
    - in-degree
    - out-degree



# Weighted Graph

- **weighted graph:** setiap sisi memiliki **bobot/nilai**.

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$$



$(V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), (V_1, V_4, 10)$

$(V_3, V_4, 2), (V_3, V_6, 4), (V_3, V_5, 8), (V_3, V_2, 2)$

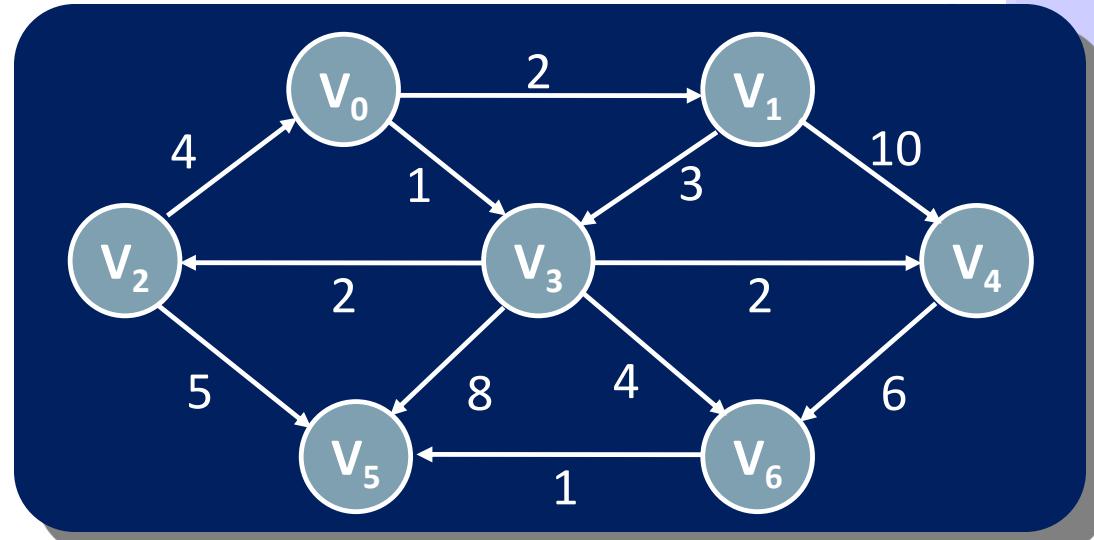
$(V_2, V_0, 4), (V_2, V_5, 5), (V_4, V_6, 6), (V_6, V_5, 1)$

■  $|V| = 7; |E| = 12$



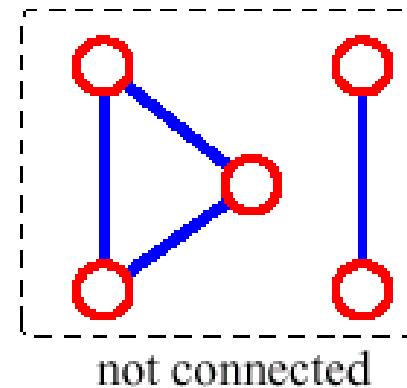
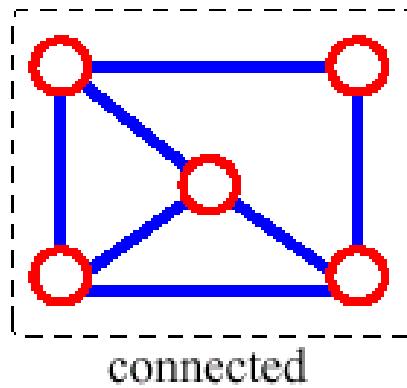
# Istilah

- **Jalur/path**: urutan simpul (vertices)  $v_1, v_2, \dots, v_k$  sedemikian sehingga simpul yang berurutan  $v_i$  dan  $v_{i+1}$  adalah simpul yang terhubung.
- **simple path**: tidak ada simpul yang diulang.
- **cycle**: simple path, dengan catatan simpul awal sama dengan simpul akhir
- **DAG (Directed Acyclic Graph)**: Graph dengan busur/sisi yang memiliki arah dan tidak memiliki cycles.



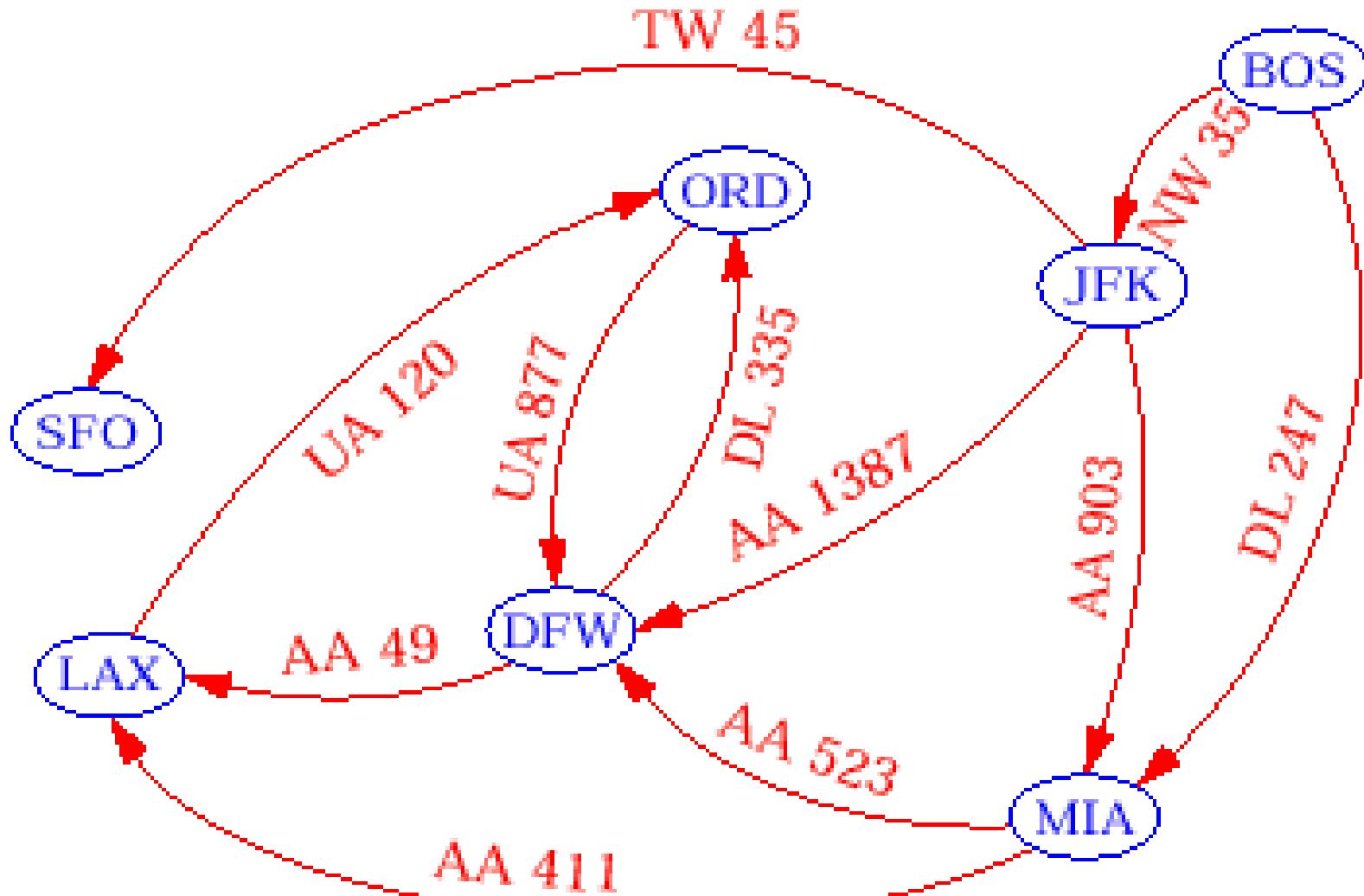
# Istilah

- *connected graph*: tiap simpul terhubung dengan simpul lain



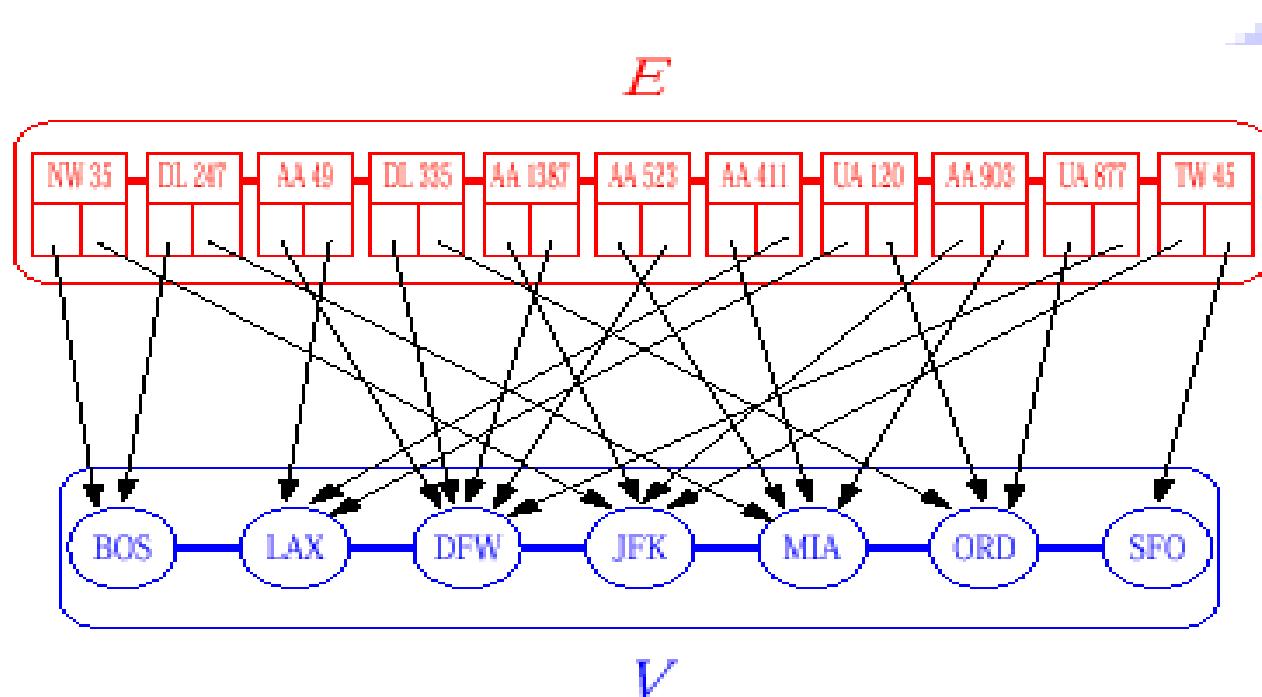
- *subgraph*: bagian simpul dan sisi yang dapat membentuk graph

# Representasi



# Representasi: *Edge List*

- Struktur ***edge list*** hanya menyimpan simpul dan sisi dalam sebuah list yang tidak terurut.
- Pada tiap sisi disimpan informasi simpul yang terhubung oleh sisi tersebut.
- mudah diimplementasikan.
- Tidak efisien dalam keperluan mencari sisi bila diketahui simpulnya.



# Edge List: Representation

```
class Node  
{  
    String label;  
}
```

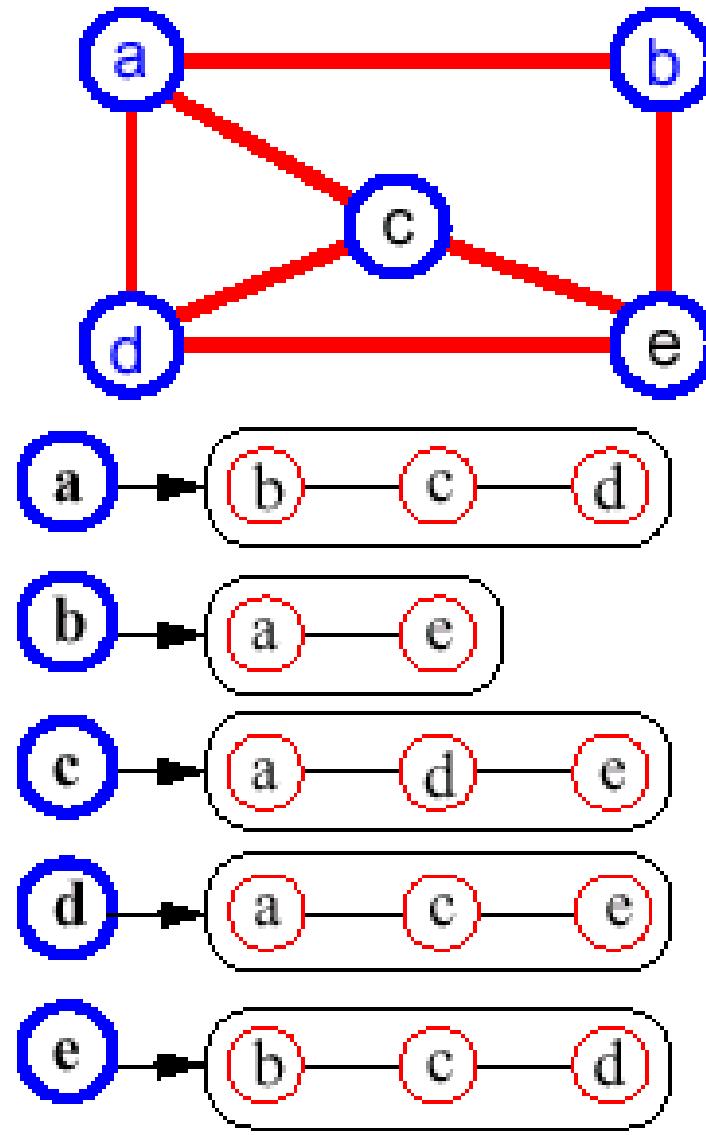
```
class Edge  
{  
    Node from;  
    Node to;  
    String label;  
    int weight;  
}
```

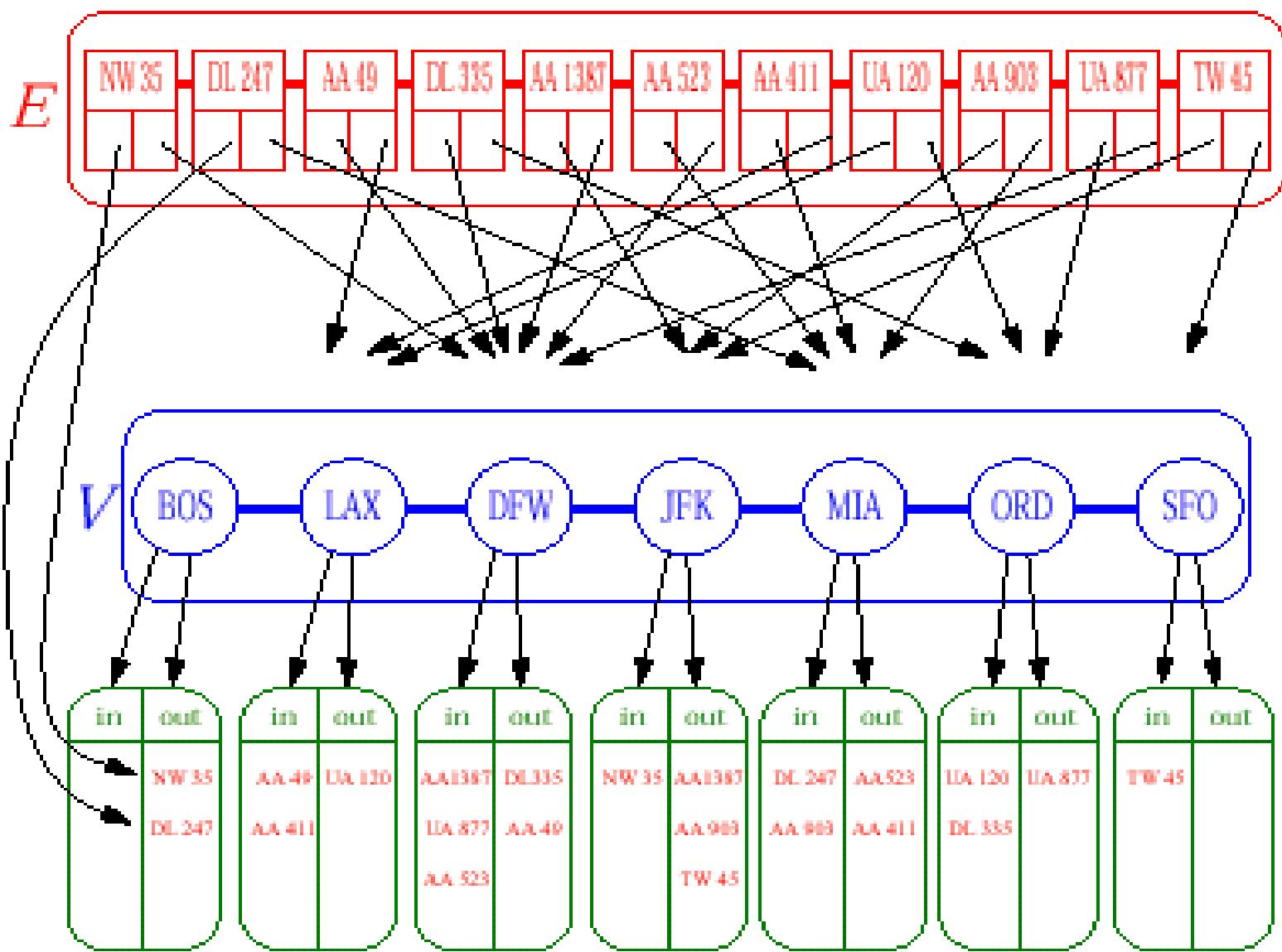
```
class Graph  
{  
    List<Edge> edgeList;  
}
```



# Representasi: Adjacency List (traditional)

- **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.





# Adjacency List: Representation

```
class Node  
{  
    String label;  
}
```

```
class AdjacencyList  
{  
    Node node;  
    List<Edge> adjacent;  
}
```

```
class Graph  
{  
    List<AdjacencyList> adjacencyLists;  
}
```

```
class Edge  
{  
    Node from;  
    Node to;  
    String label;  
    int weight;  
}
```



# Adjacency List: Representation (alt.)

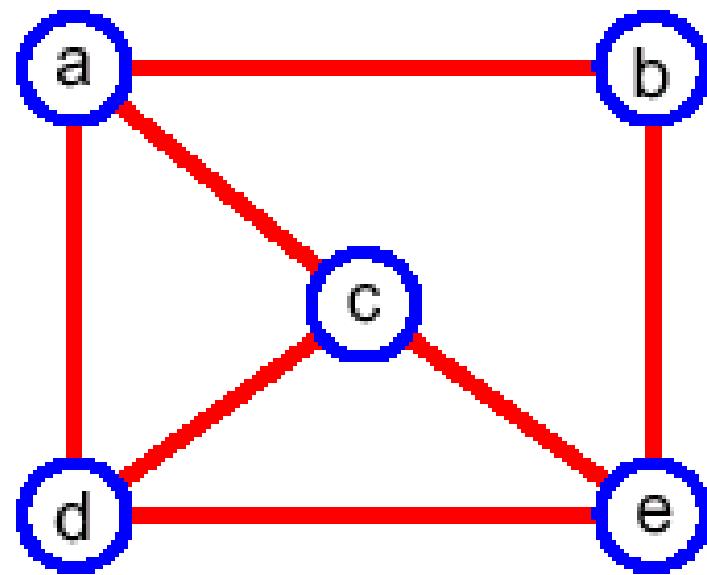
```
class Node  
{  
    String label;  
}
```

```
class Edge  
{  
    Node from;  
    Node to;  
    String label;  
    int weight;  
}
```

```
class Graph  
{  
    Map<Node, List<Edge>> adjacencyLists;  
}
```



# Representasi: Adjacency Matrix (traditional)



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

- matrix M dengan elemen setiap pasang simpul
  - $M[i,j] = \text{true}$  artinya ada sisi dari simpul  $(i,j)$  di graph.
  - $M[i,j] = \text{false}$  artinya tidak ada sisi dari simpul  $(i,j)$  di graph.



# Representation: Adjacency Matrix

	0	1	2	3	4	5	6
0	Ø	Ø	NW 35	Ø	DL 247	Ø	Ø
1	Ø	Ø	Ø	AA 49	Ø	DL 335	Ø
2	Ø	AA 1387	Ø	Ø	AA 903	Ø	TW 45
3	Ø	Ø	Ø	Ø	Ø	UA 120	Ø
4	Ø	AA 523	Ø	AA 411	Ø	Ø	Ø
5	Ø	UA 877	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø

BOS   DFW   JFK   LAX   MIA   ORD   SFO  
0      1      2      3      4      5      6

# Representation: Adjacency Matrix

```
class Node  
{  
    String label;  
}
```

```
class Edge  
{  
    String label;  
    int weight;  
}
```

```
class Graph  
{  
    List<Node> nodeList;  
    Edge [ ] [ ] adjacencyMatrix;  
}
```

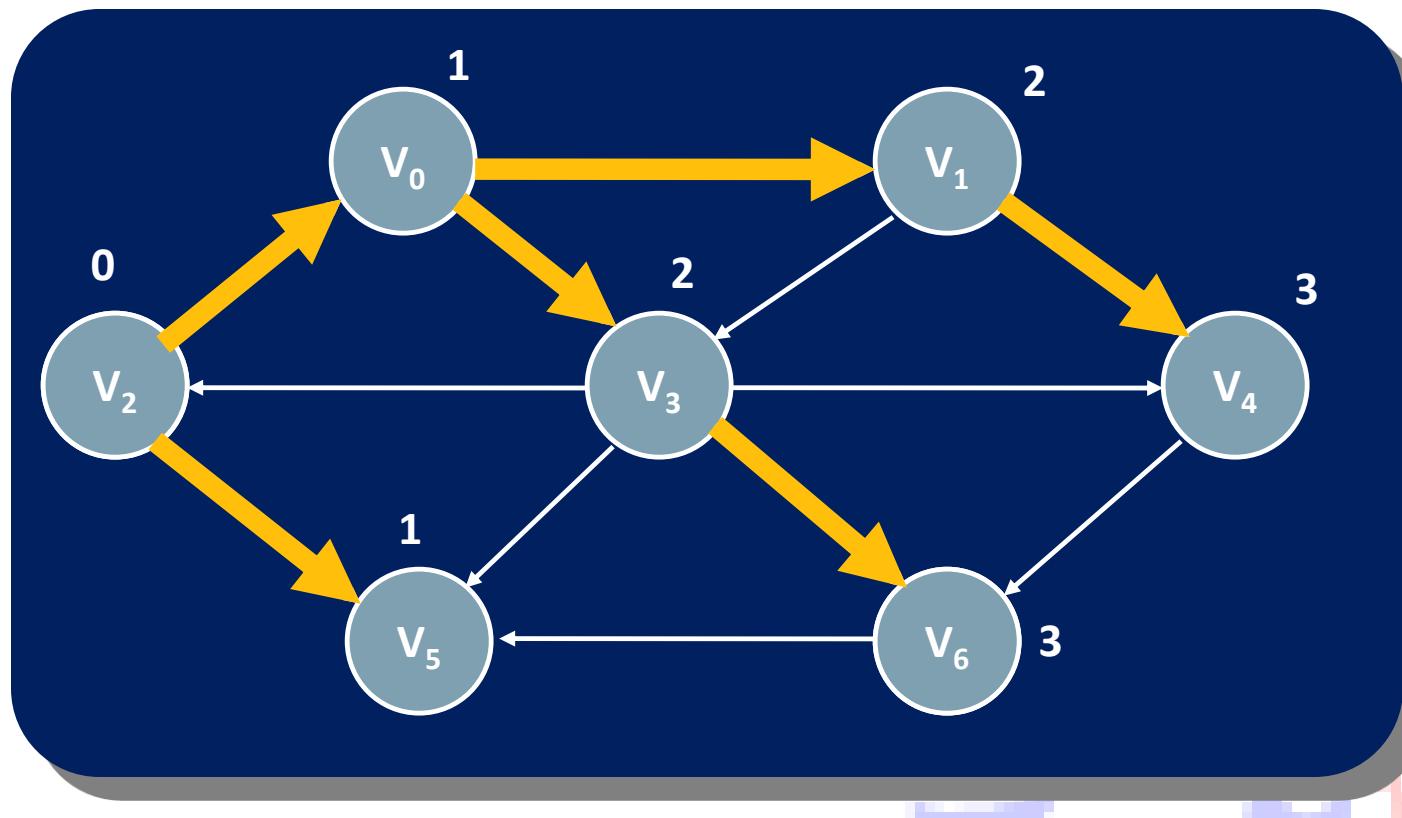
see the other slides

# GRAPH TRAVERSAL

# GRAPH ALGORITHMS

# Shortest Path:

- Vertex awal:  $V_2$
- Bila sisi tidak memiliki bobot, gunakan algoritma BFS (**Breadth First Search**).



# Dijkstra's Algorithm

- Banyak masalah → weighted graph (mis: jaringan transport)
- Algoritma **Dijkstra** menghitung jarak tiap simpul dari simpul awal hingga akhirnya diketahui jarak terpendek simpul akhir yang diinginkan.
- Algoritma mengingat simpul mana saja yang telah dihitung jarak terpendeknya dan dinyatakan dalam kelompok hijau (pada literatur dinyatakan sebagai awan putih/white cloud).
- Untuk simpul yang baru sebagian dihitung jaraknya dan belum bisa dipastikan apakah itu jarak terpendek, dinyatakan dengan kelompok abu-abu.
- Untuk simpul yang sama sekali belum dihitung, dinyatakan dalam kelompok hitam.



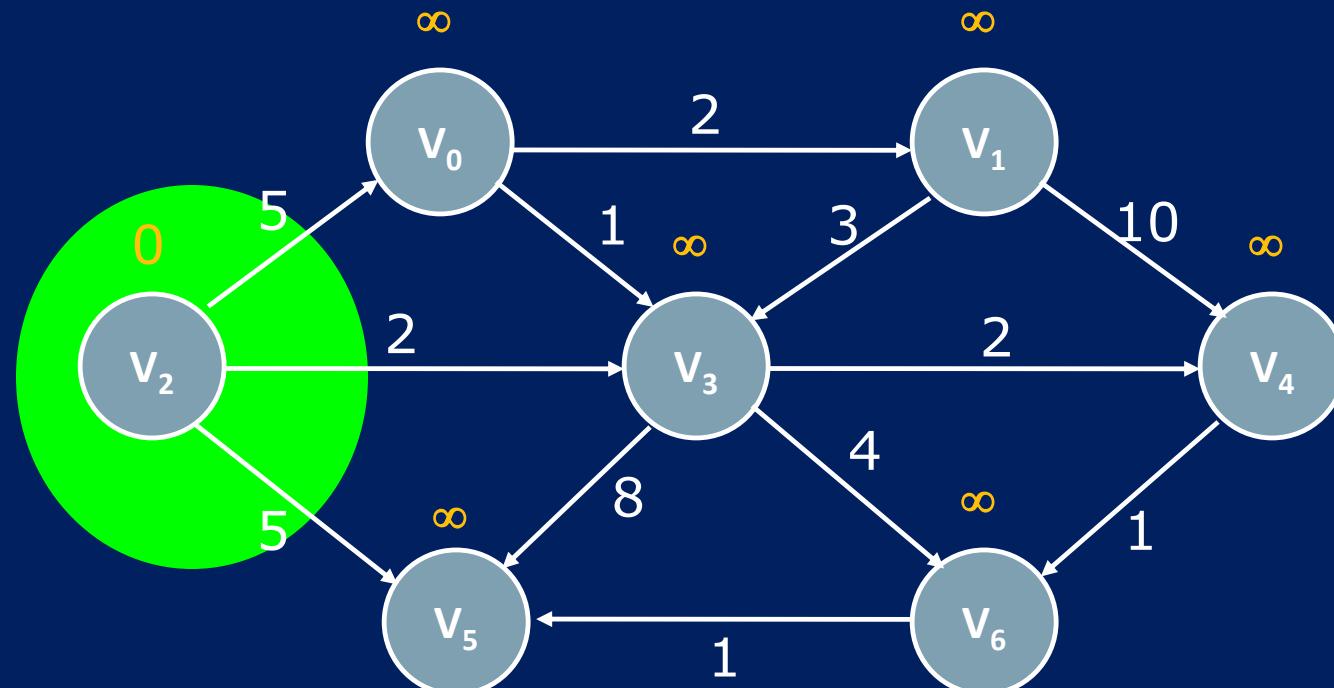
# Dijkstra's Algorithm

- Algoritma menggunakan label  $D[v]$  untuk menyimpan perkiraan jarak terpendek antara  $s$  dan  $v$ .
- Ketika sebuah simpul  $v$  ditambahkan kedalam kelompok abu-abu nilai  $D[v]$  sama dengan bobot antara  $s$  dan  $v$ .
- pada awalnya, nilai label D untuk setiap simpul adalah:
  - $D[s] = 0$
  - $D[v] = \infty$  untuk  $v \neq s$



# Dijkstra's Algorithm: stages

- Awal: Tentukan simpul awal.

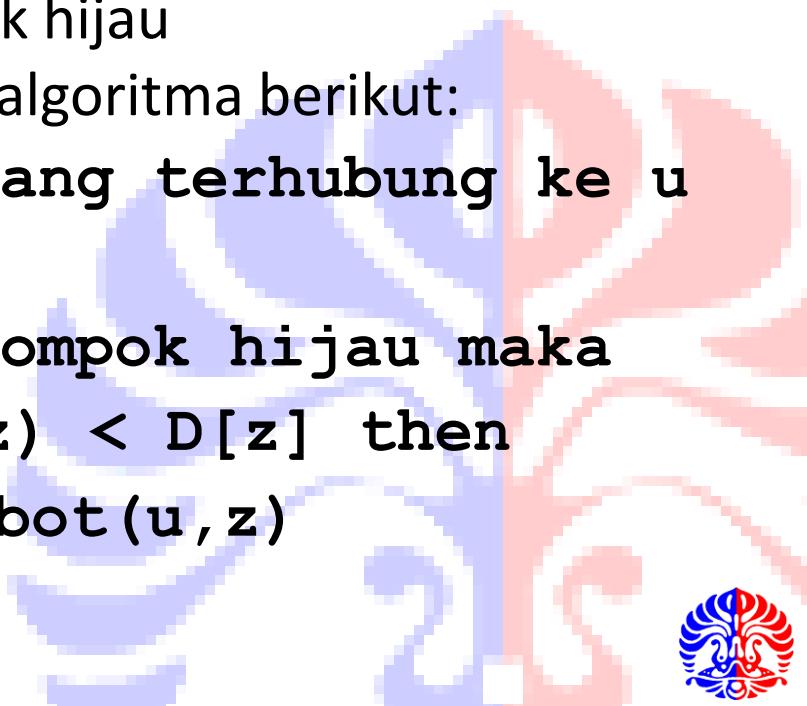


# Expanding the White Cloud

- Setiap penambahan simpul, kita harus uji apakah jalur melalui  $u$  lebih baik.
- Misalkan **u** adalah sebuah simpul yang tidak berada di **kelompok hijau**, tapi sudah diketahui jarak terpendeknya dari s
  - tambahkan **u** ke dalam kelompok hijau
  - hitung jarak **simpul lain** dengan algoritma berikut:

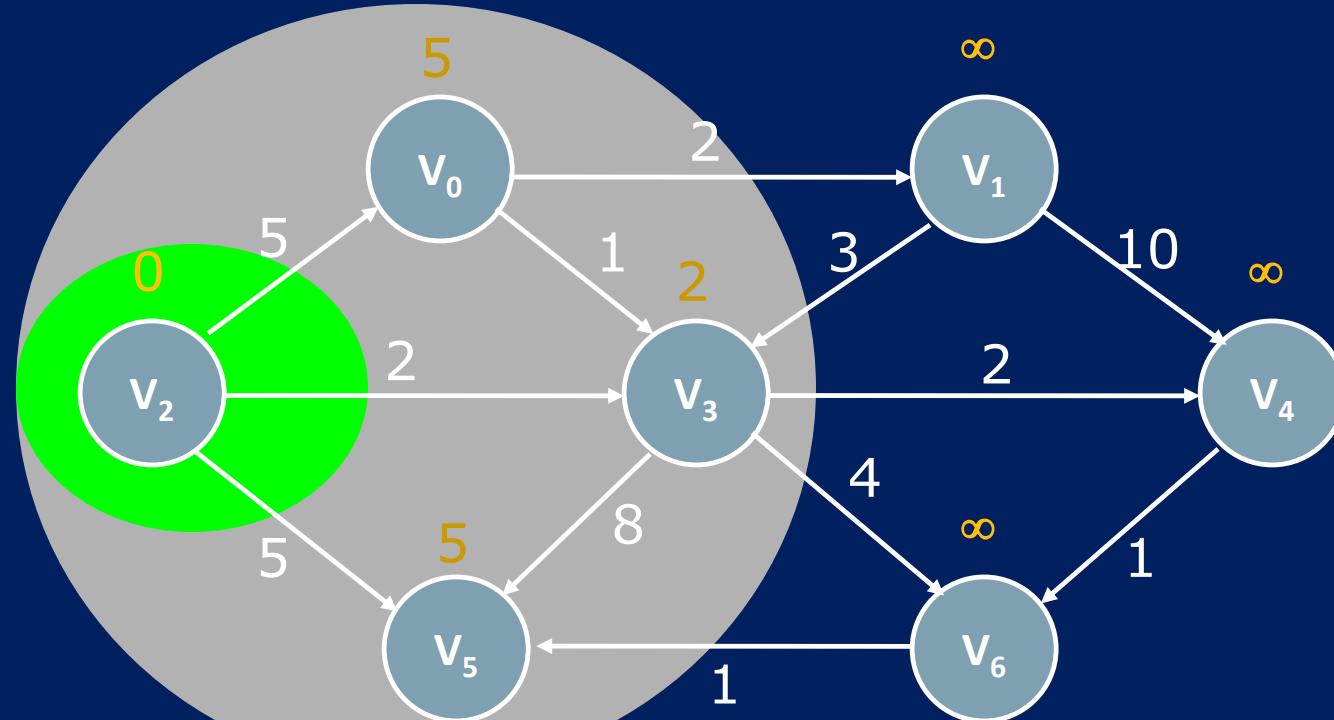
Untuk tiap simpul  $z$  yang terhubung ke  $u$  lakukan:

jika  $z$  tidak di kelompok hijau maka  
if  $D[u] + \text{bobot}(u, z) < D[z]$  then  
 $D[z] = D[u] + \text{bobot}(u, z)$



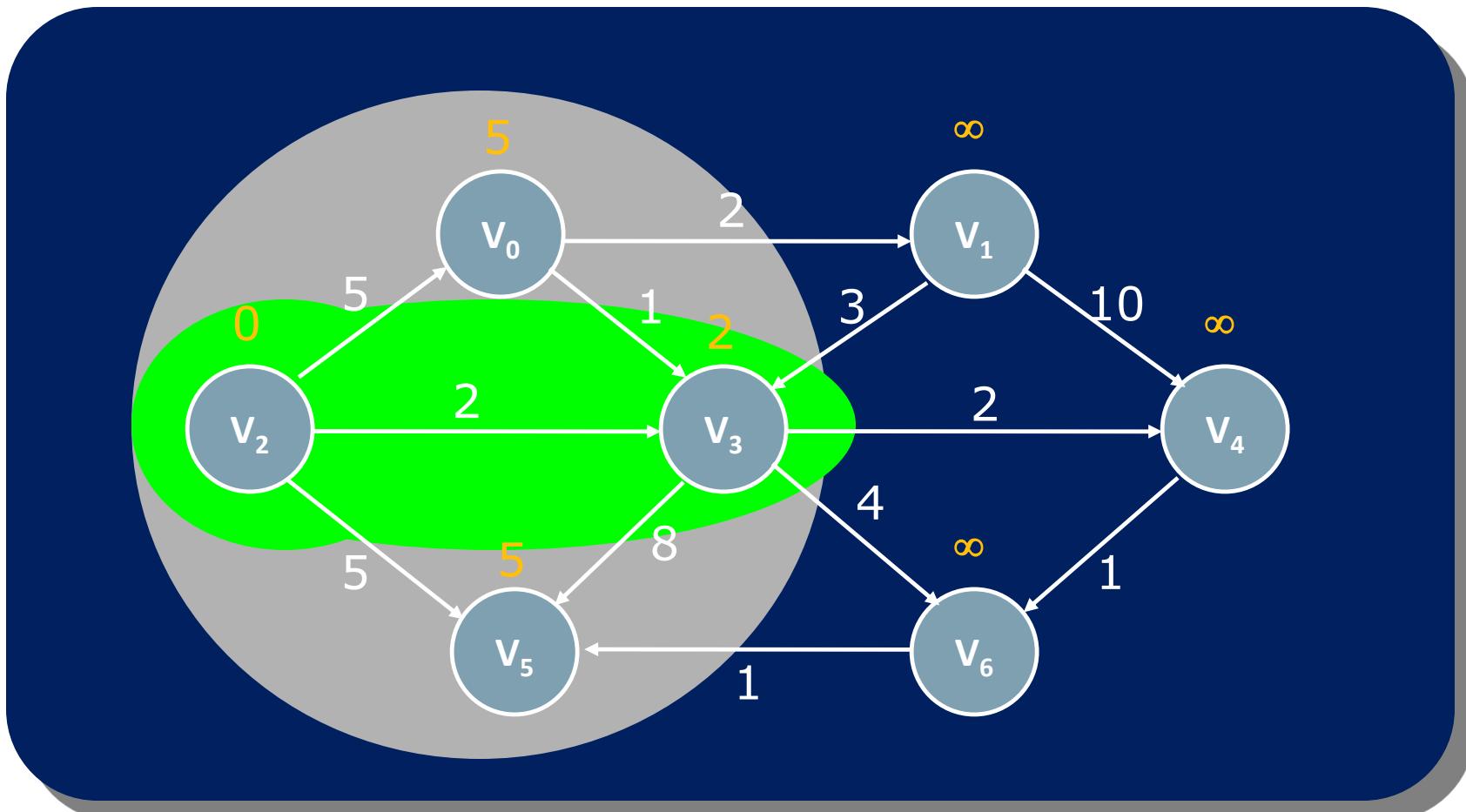
# Dijkstra's Algorithm: stages

- setelah  $V_2$  ditambahkan ke kelompok hijau, hitung  $D[V_x]$  untuk setiap  $V_x$  yang terhubung ke  $V_2$



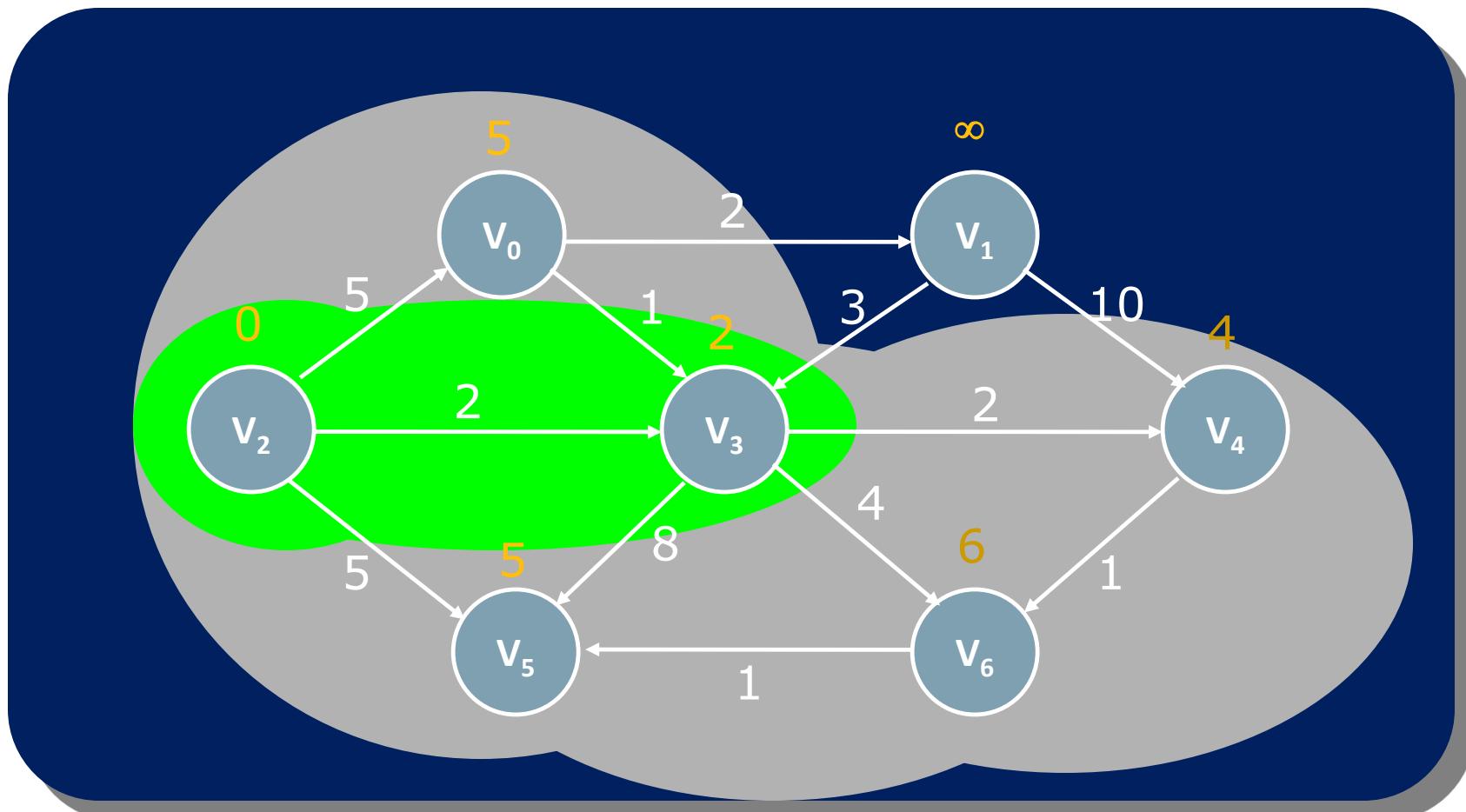
# Dijkstra's Algorithm: stages

- Tambahkan ke dalam kelompok hijau simpul pada kelompok abu-abu yang memiliki nilai  $D[V]$  minimum.
- Pada contoh adalah simpul  $V_3$



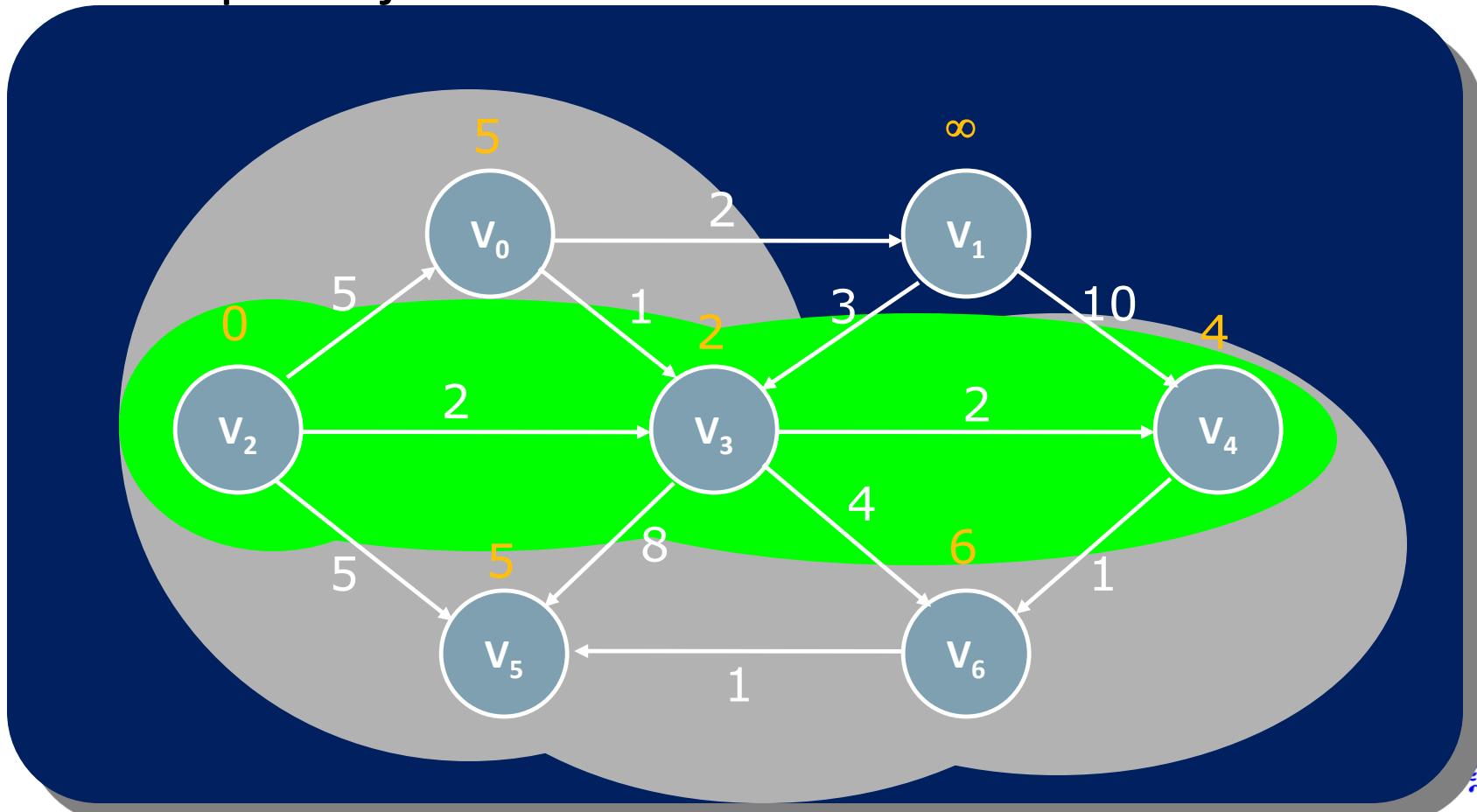
# Dijkstra's Algorithm: stages

- Setelah  $V_3$  ditambahkan ke kelompok hijau, hitung  $D[V_x]$  untuk setiap  $V_x$  yang terhubung dengan  $V_3$ . Simpul-simpul tersebut menjadi kelompok abu-abu.



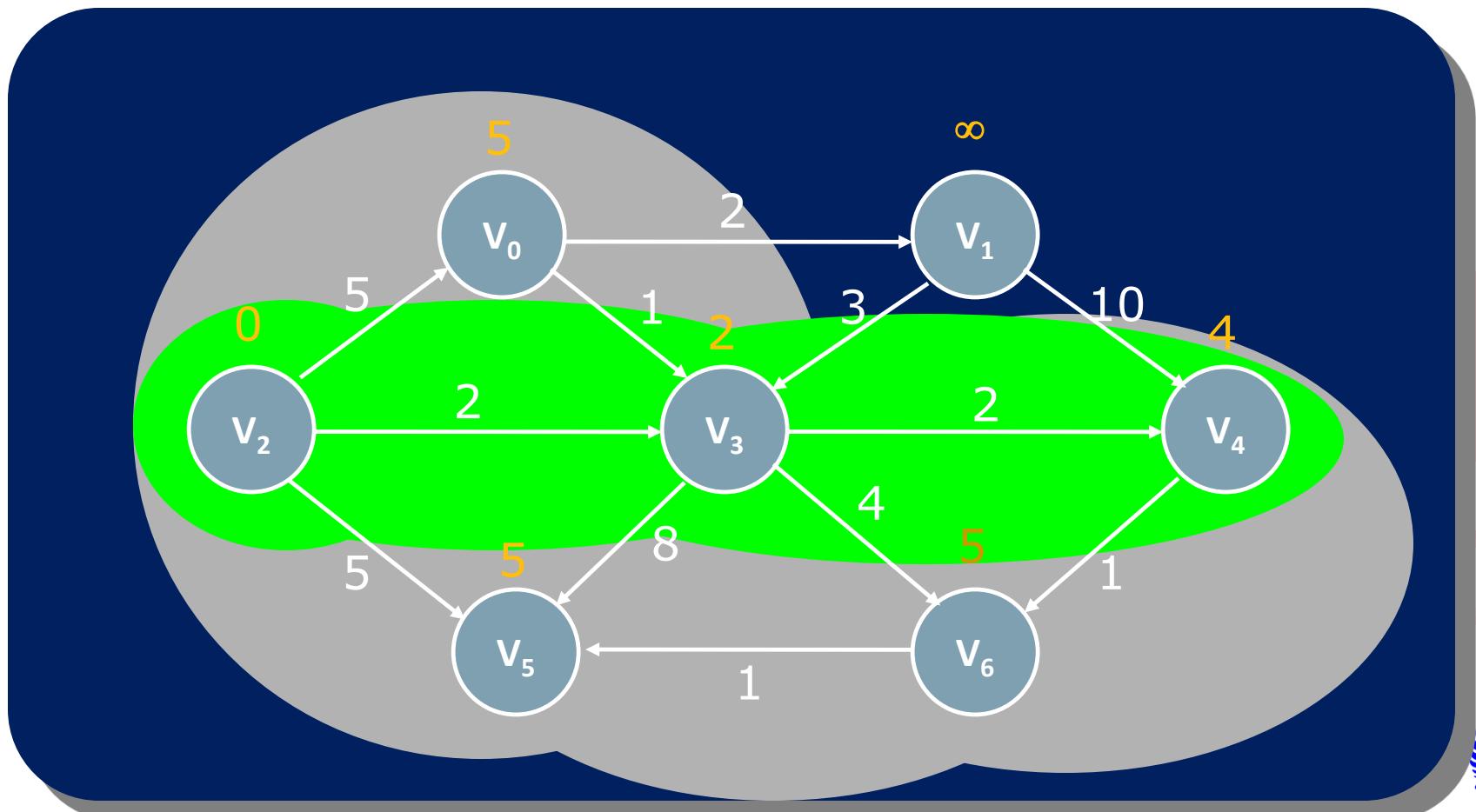
# Dijkstra's Algorithm: stages

- Pilih dari kelompok abu-abu, simpul yang memiliki nilai  $D[V]$  paling minimum dan tambahkan pada kelompok hijau.



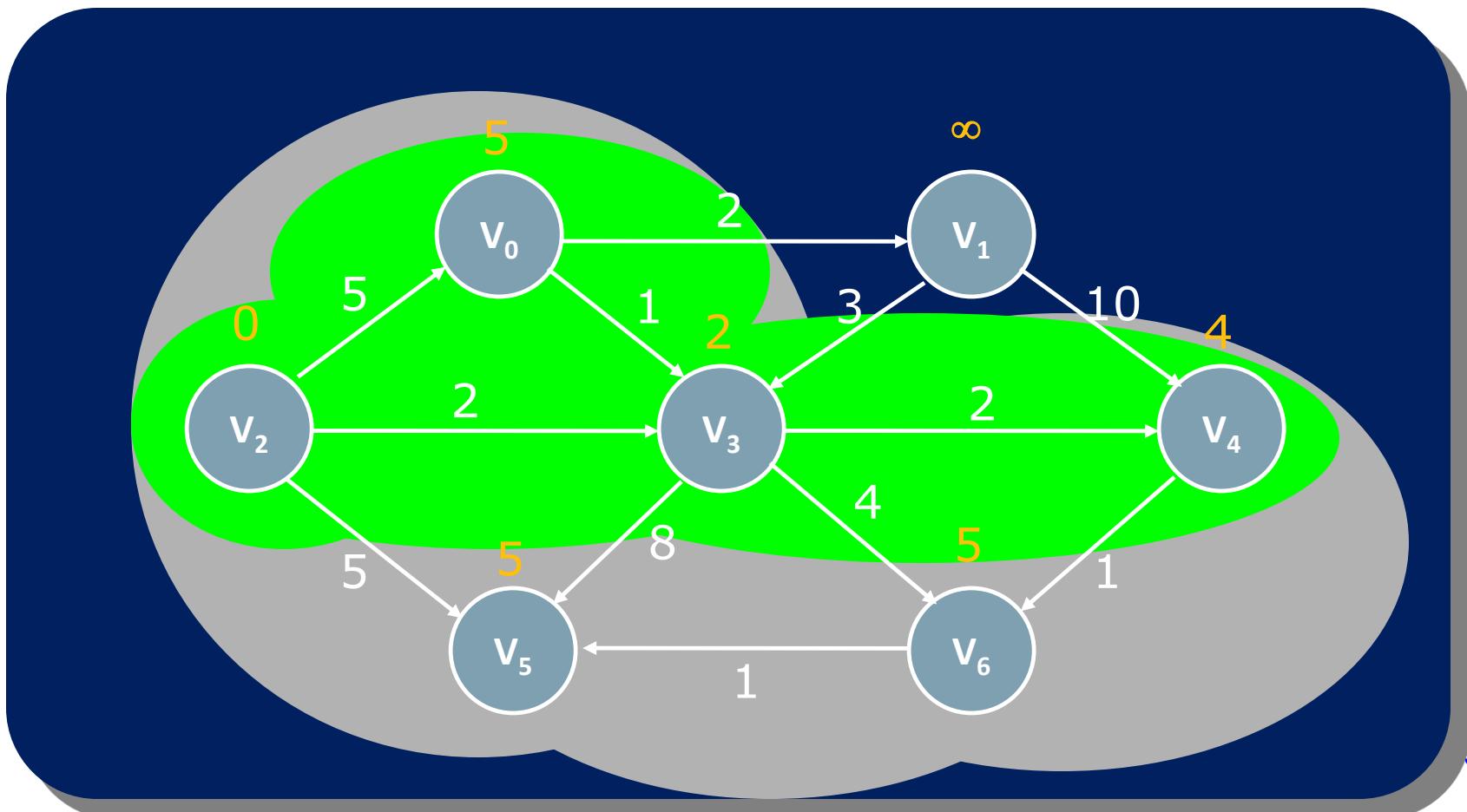
# Dijkstra's Algorithm: stages

- Setelah  $V_4$  ditambahkan ke kelompok hijau, hitung  $D[V_x]$  untuk setiap  $V_x$  yang terhubung dengan  $V_4$ . Simpul-simpul tersebut menjadi kelompok abu-abu.



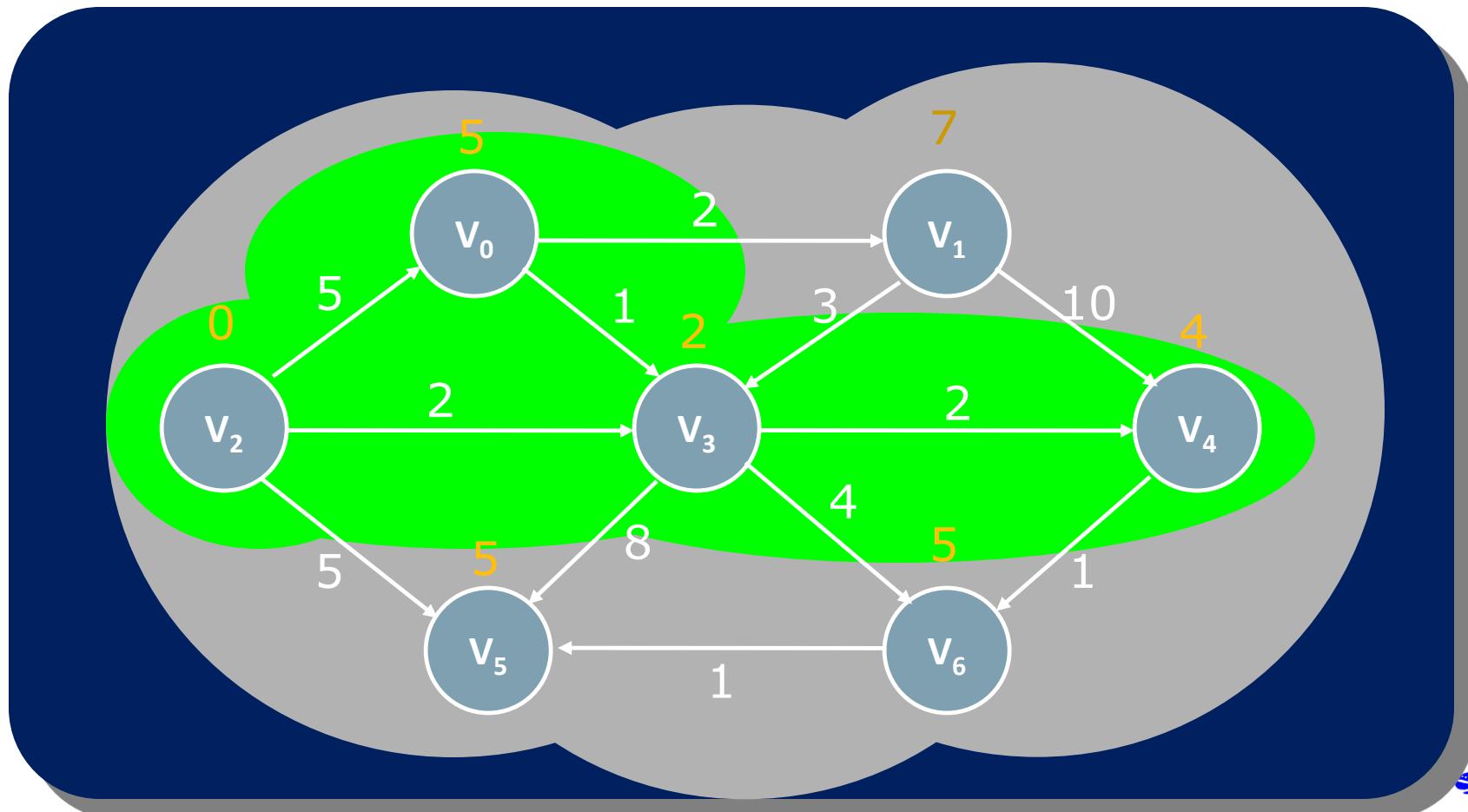
# Dijkstra's Algorithm: stages

- Pilih dari kelompok abu-abu, simpul yang memiliki nilai  $D[V]$  paling minimum dan tambahkan pada kelompok hijau.

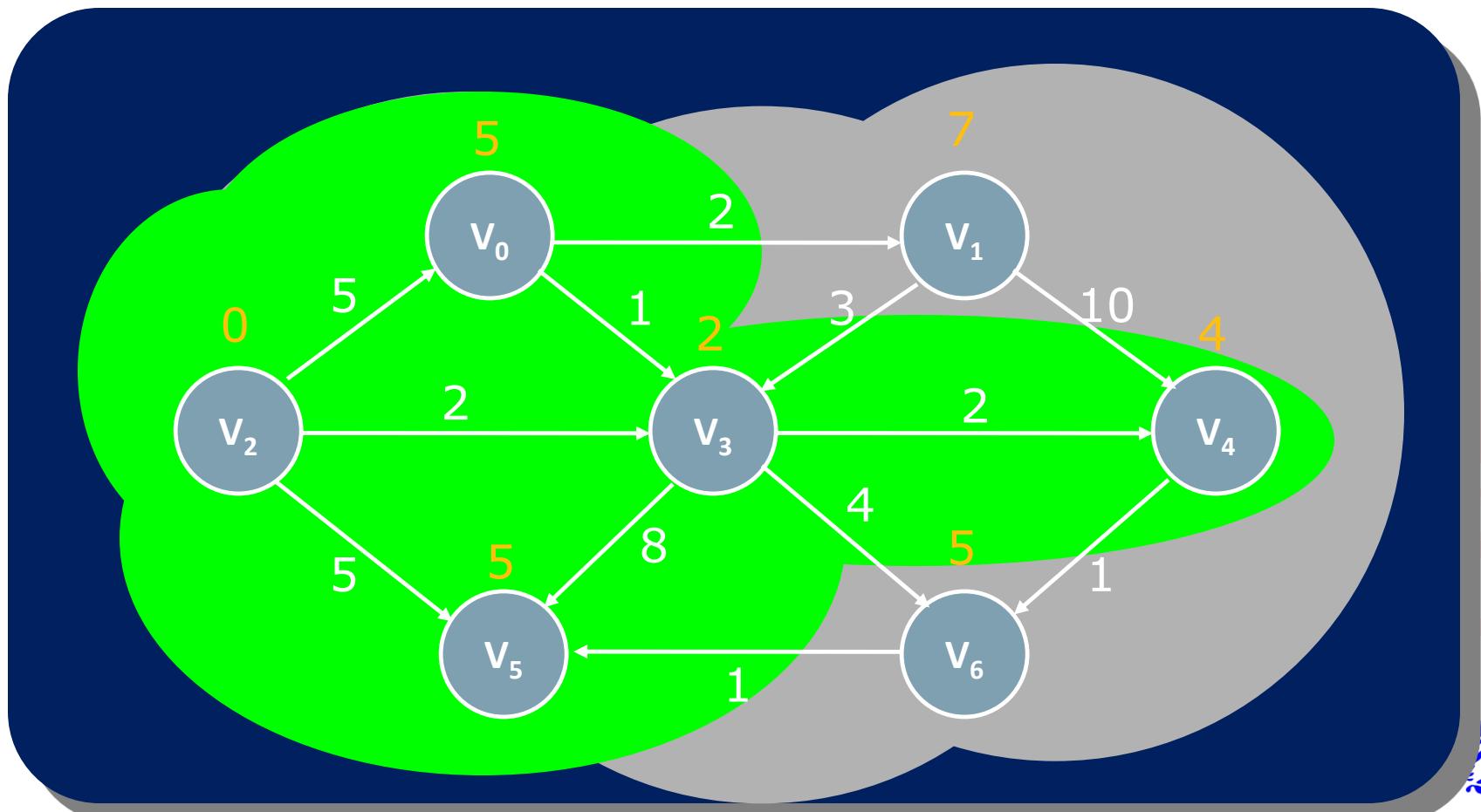


# Dijkstra's Algorithm: stages

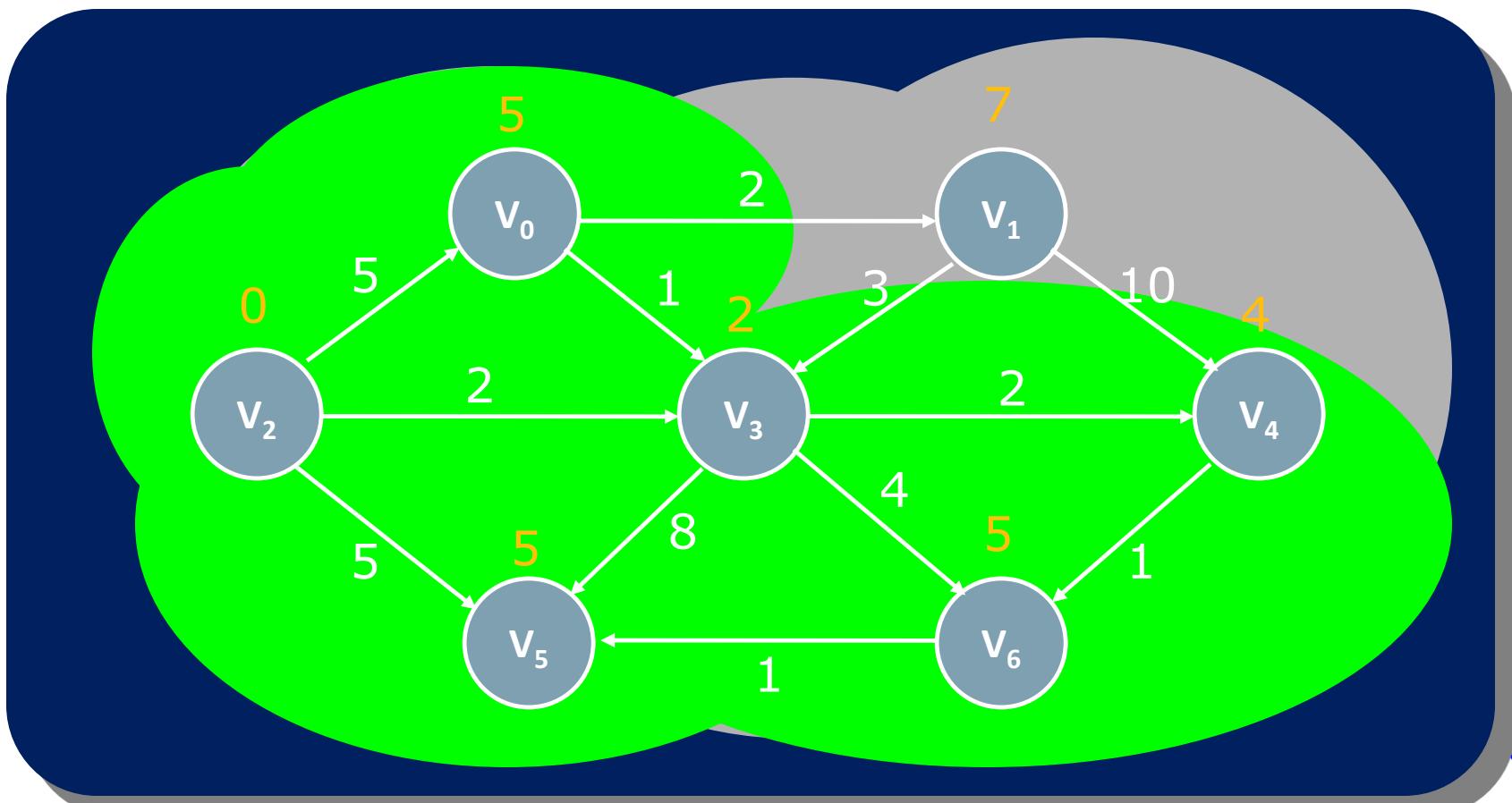
- Setelah  $V_4$  ditambahkan ke kelompok hijau, hitung  $D[V_x]$  untuk setiap  $V_x$  yang terhubung dengan  $V_4$ . Simpul-simpul tersebut menjadi kelompok abu-abu.



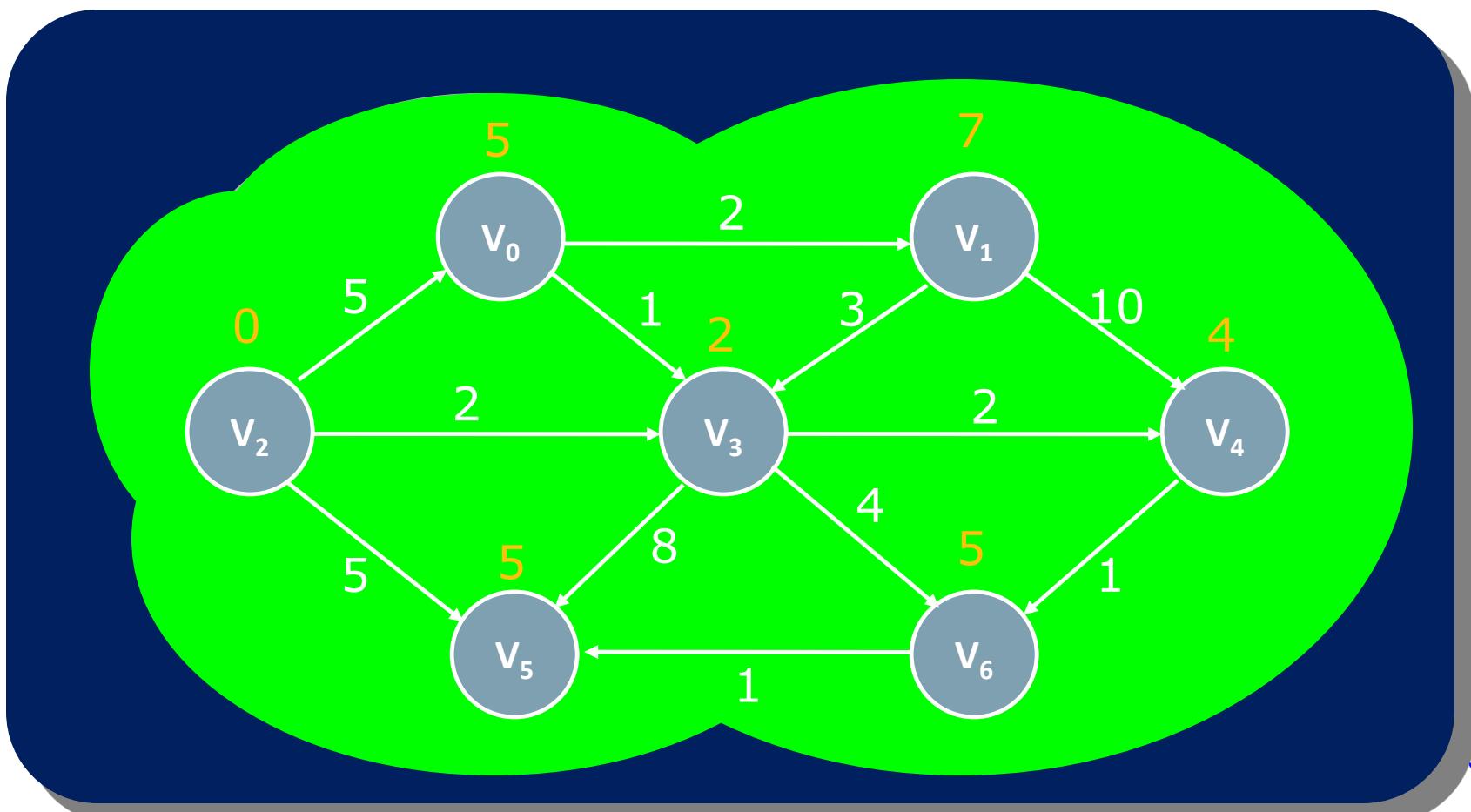
- Pilih dari kelompok abu-abu, simpul yang memiliki nilai  $D[V]$  paling minimum dan tambahkan pada kelompok hijau.
- Setelah  $V_5$  ditambahkan ke kelompok hijau, hitung  $D[V_x]$  untuk setiap  $V_x$  yang terhubung dengan  $V_5$ . Simpul-simpul tersebut menjadi kelompok abu-abu.



- Pilih dari kelompok abu-abu, simpul yang memiliki nilai  $D[V]$  paling minimum dan tambahkan pada kelompok hijau.
- Setelah  $V_6$  ditambahkan ke kelompok hijau, hitung  $D[V_x]$  untuk setiap  $V_x$  yang terhubung dengan  $V_6$ .

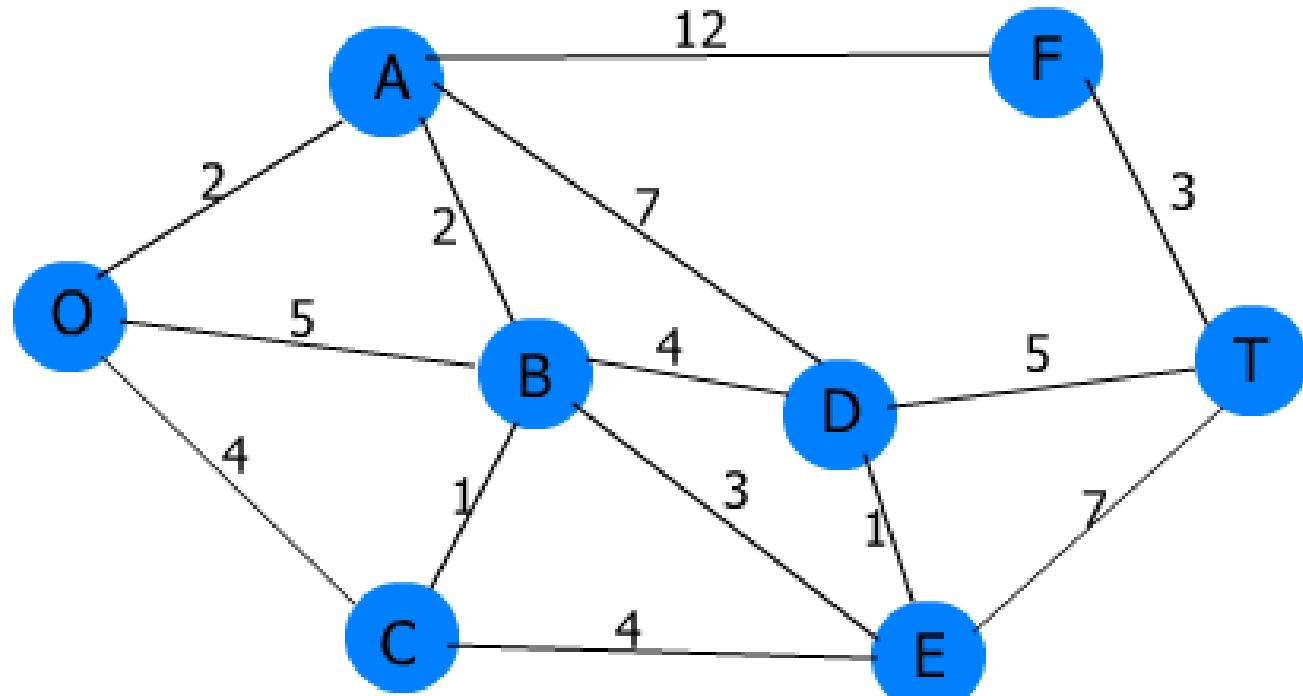


- Pilih dari kelompok abu-abu, simpul yang memiliki nilai  $D[V]$  paling minimum dan tambahkan pada kelompok hijau.
- Setelah  $V_1$  ditambahkan ke kelompok hijau, hitung  $D[V_x]$  untuk setiap  $V_x$  yang terhubung dengan  $V_1$ .

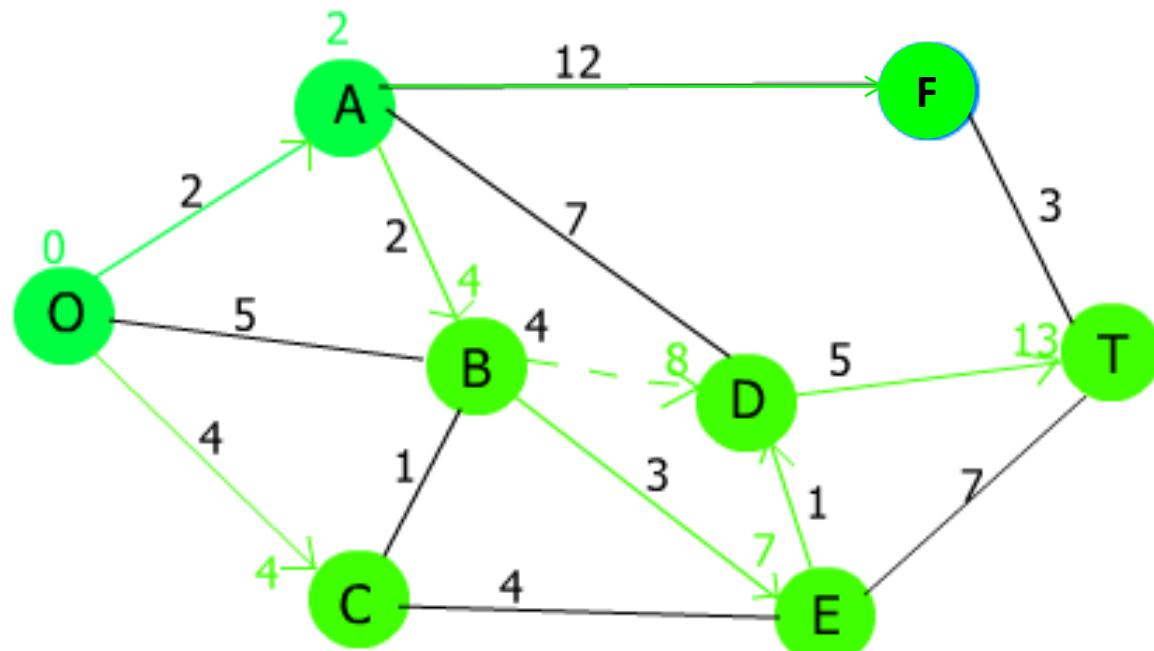


# Latihan

- Tentukan jarak minimum dari verteks O ke setiap verteks pada graph!



# Jawaban



Unsolved Node

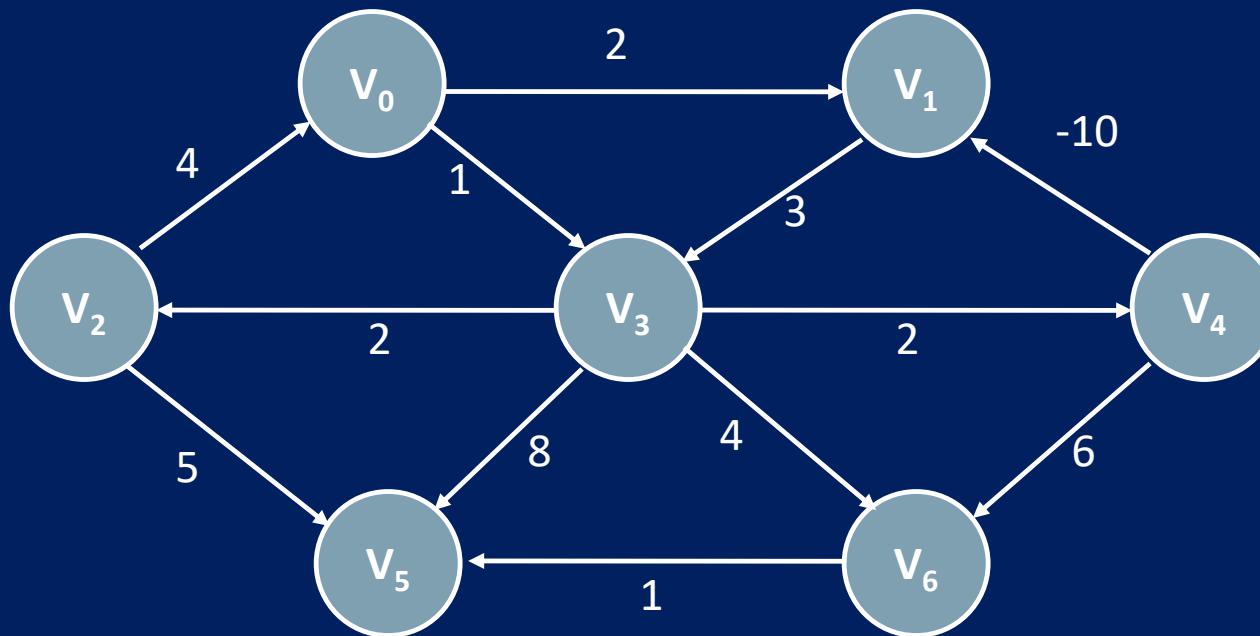


Solved Node



# Variasi shortest path problem

- Negative-weighted Shortest-path



- All-Pair Shortest Path: Floyd

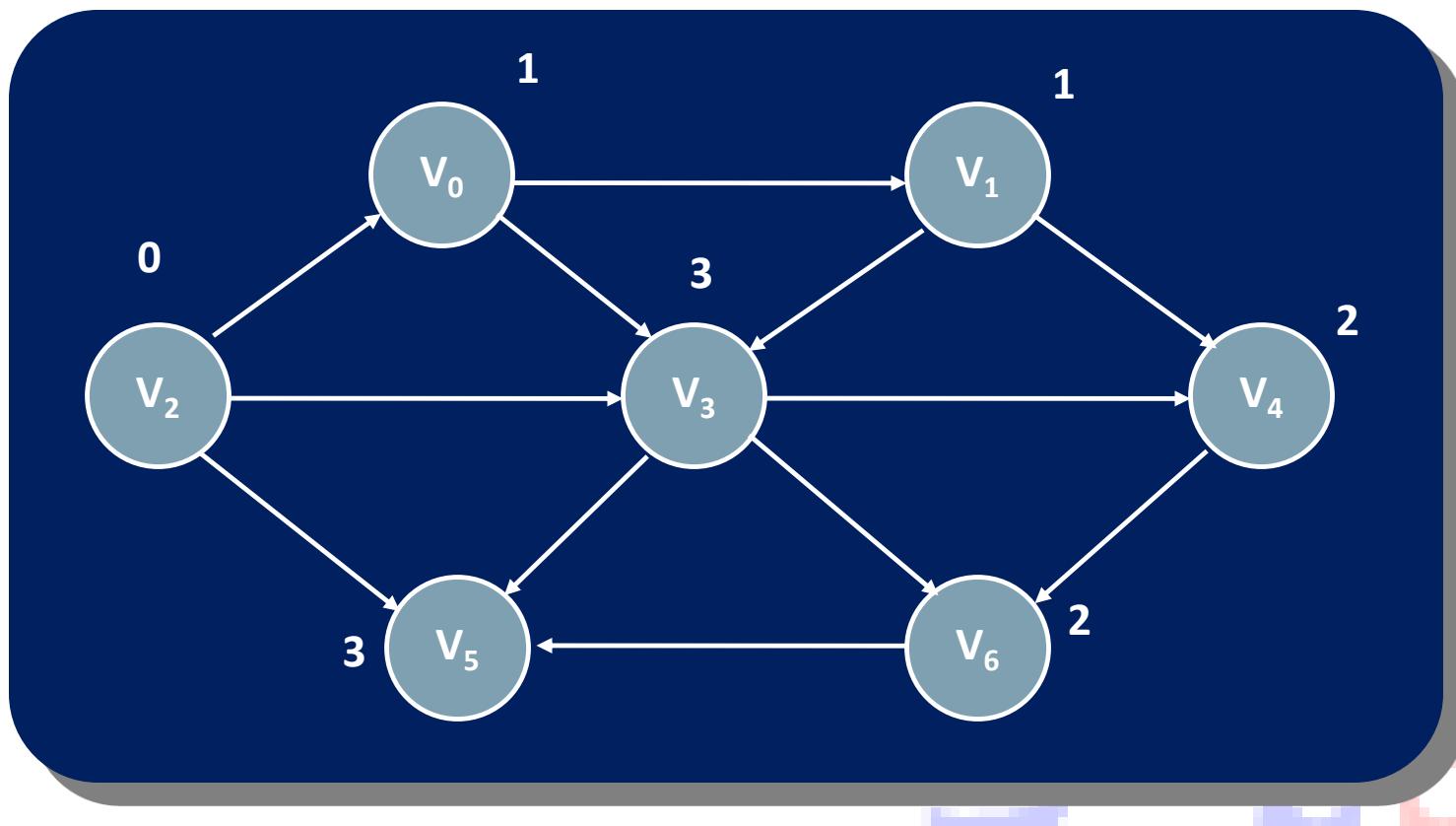
# Topological Sorting

- Sebuah *topological sort* dari sebuah *directed acyclic graph* adalah sebuah urutan simpul sehingga tiap sisi/busur terurut dari kiri ke kanan (atau sebaliknya).
- Setiap DAG memiliki minimal satu topological sort.
- Gunakan algoritma *depth-first search* untuk menentukan topological sort dari sebuah DAG.
- sebuah graph yang memiliki *cycle*, tidak memiliki *topological sort*.
- Contoh permasalahan:
  - Urutan penggerjaan proyek bangunan

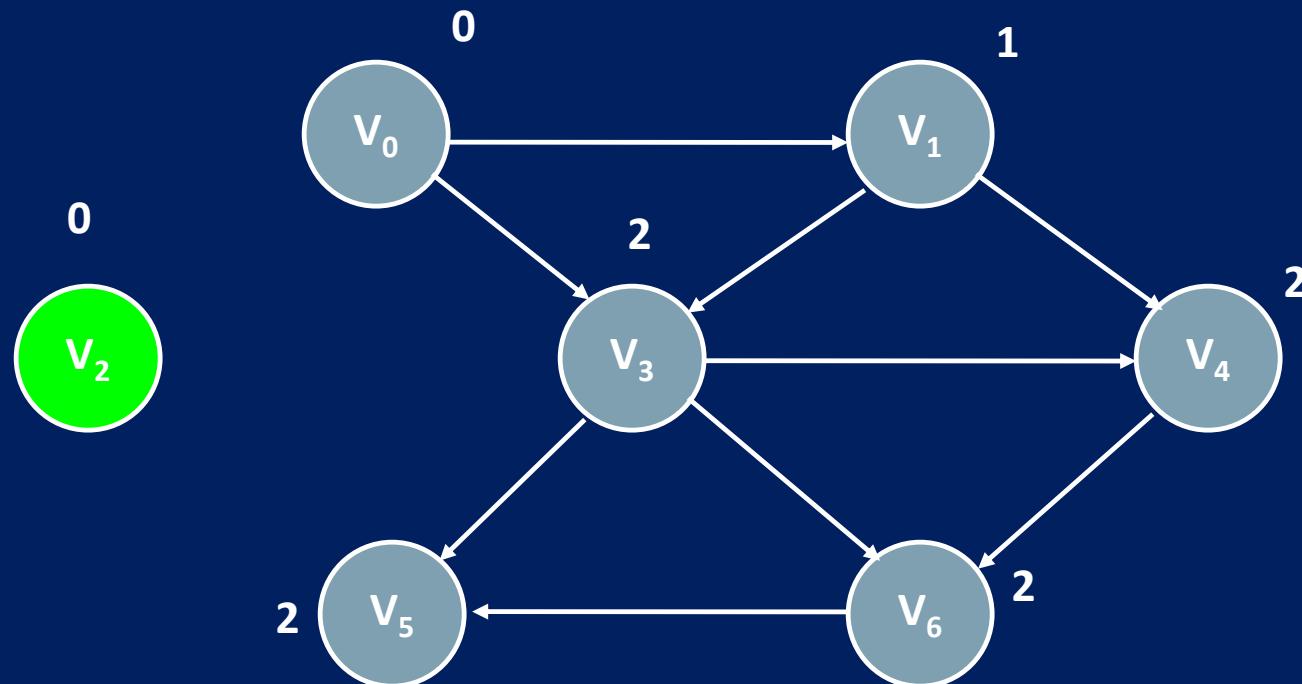


# Topological Sorting: Algoritma

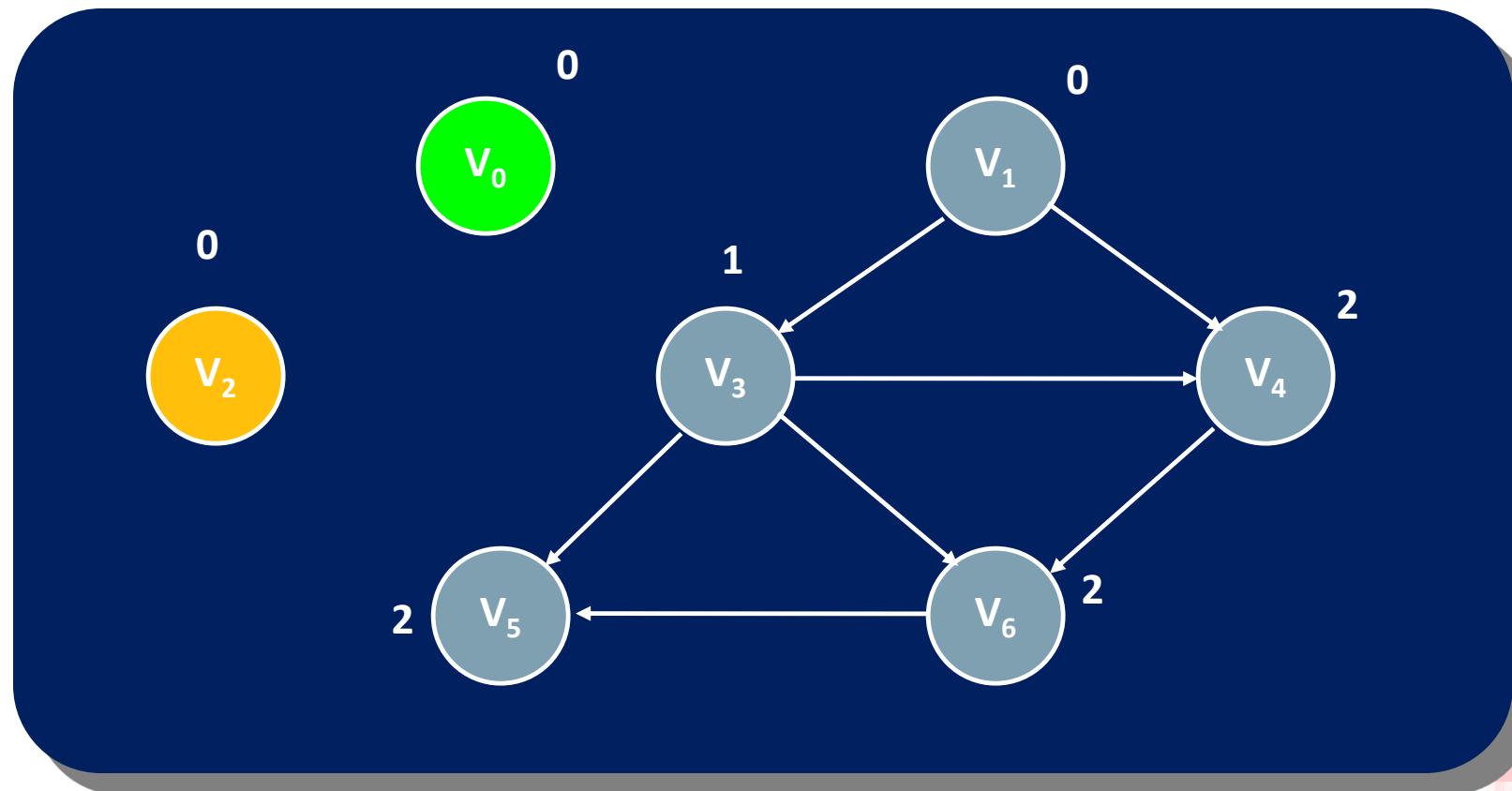
- Mulai dari sebuah simpul dengan in-degree = 0 (Tidak ada panah/sisi yang menuju simpul tersebut.)
- buang semua sisi yang berasal dari simpul tersebut.
- Sesuaikan nilai in-degree simpul lainnya.



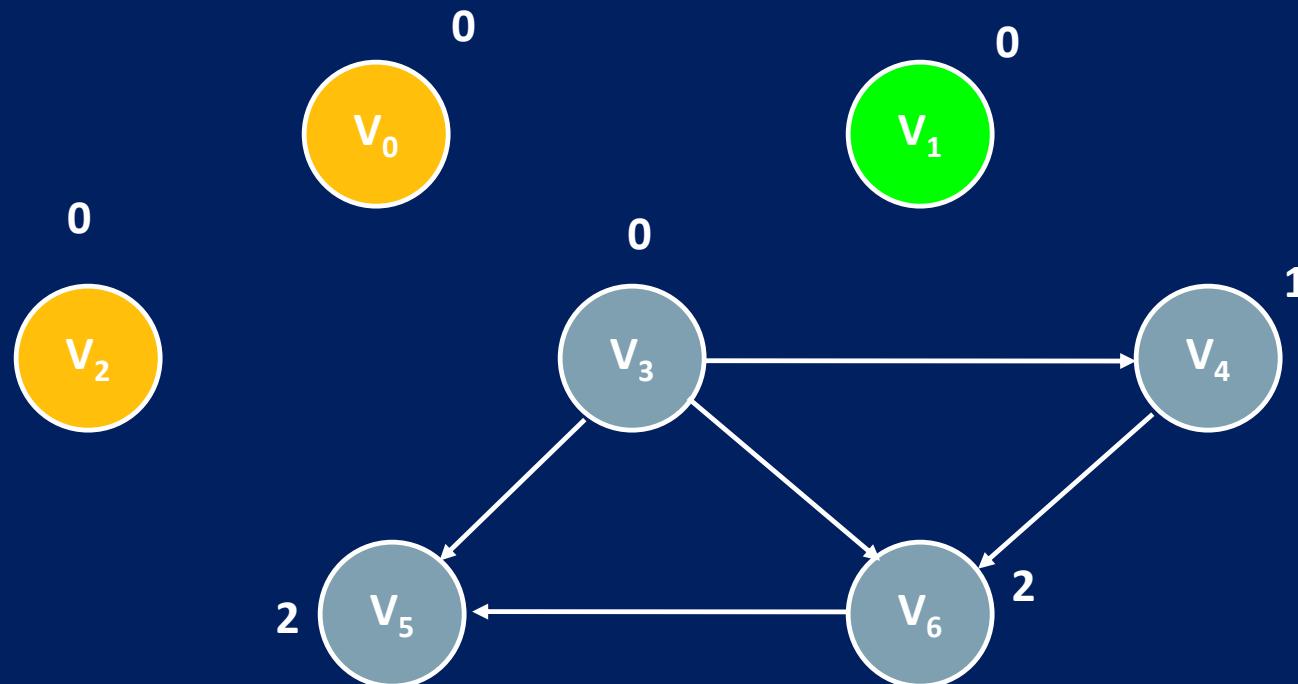
# Topological Sorting



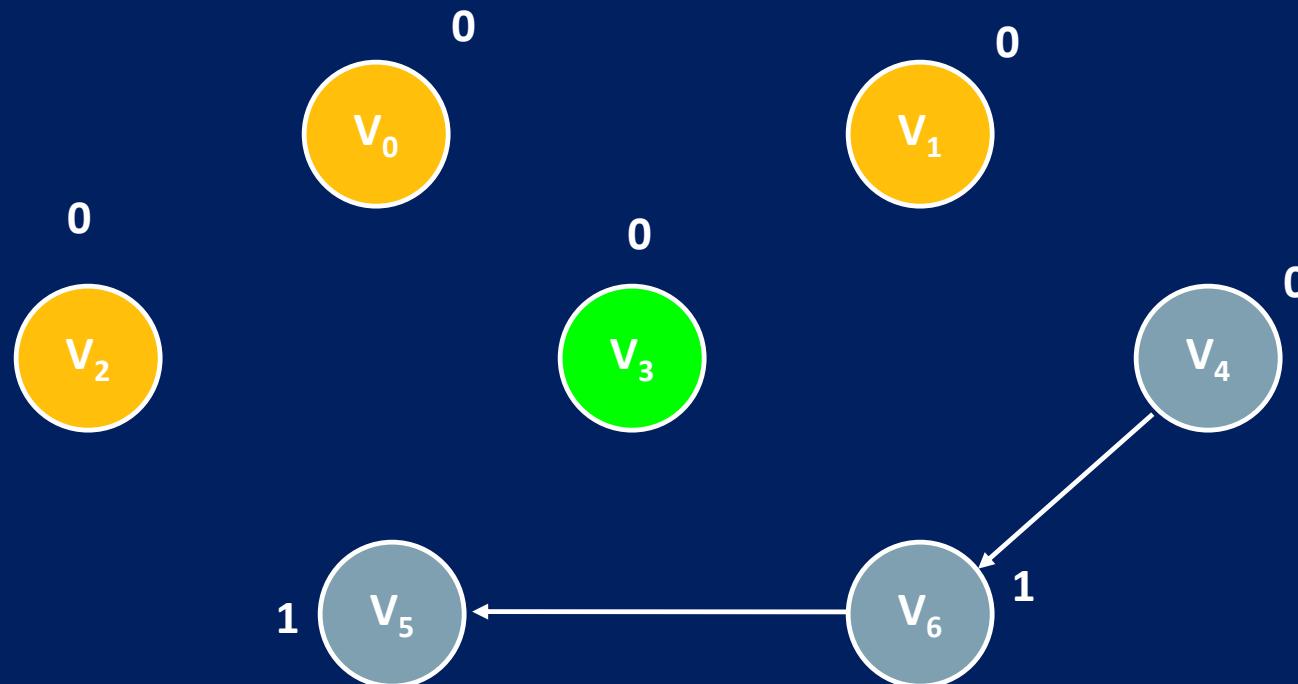
# Topological Sorting



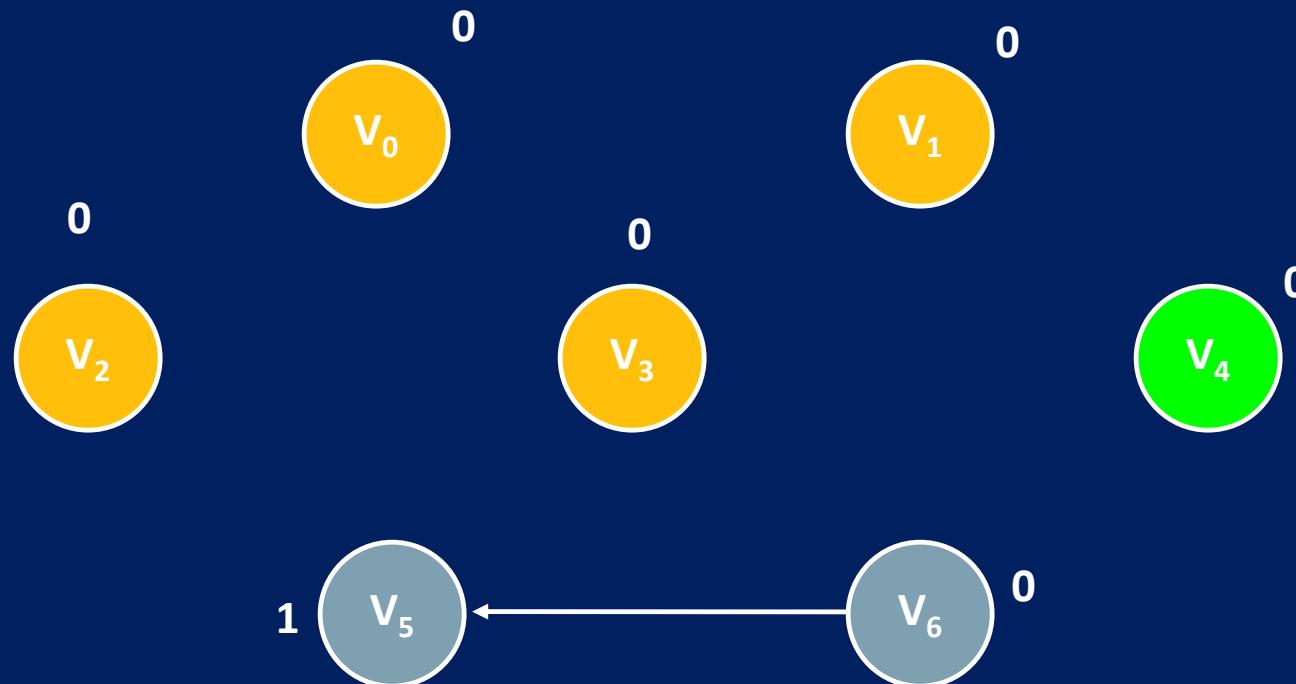
# Topological Sorting



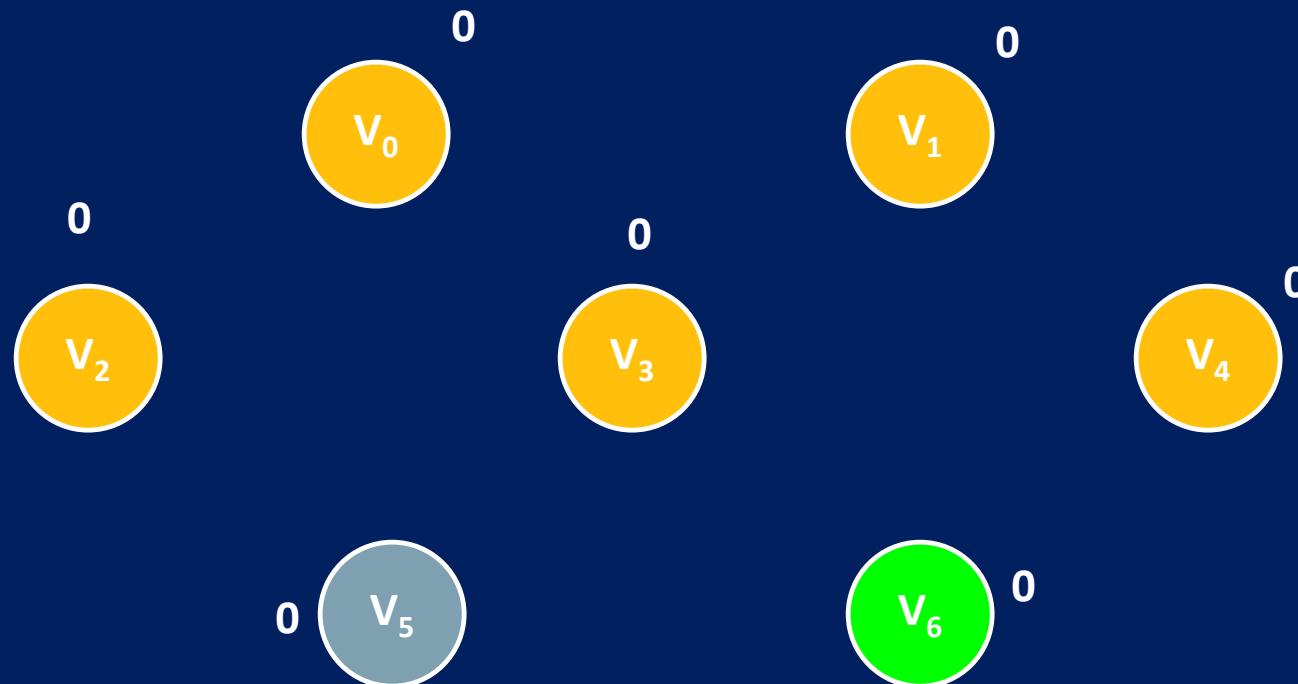
# Topological Sorting



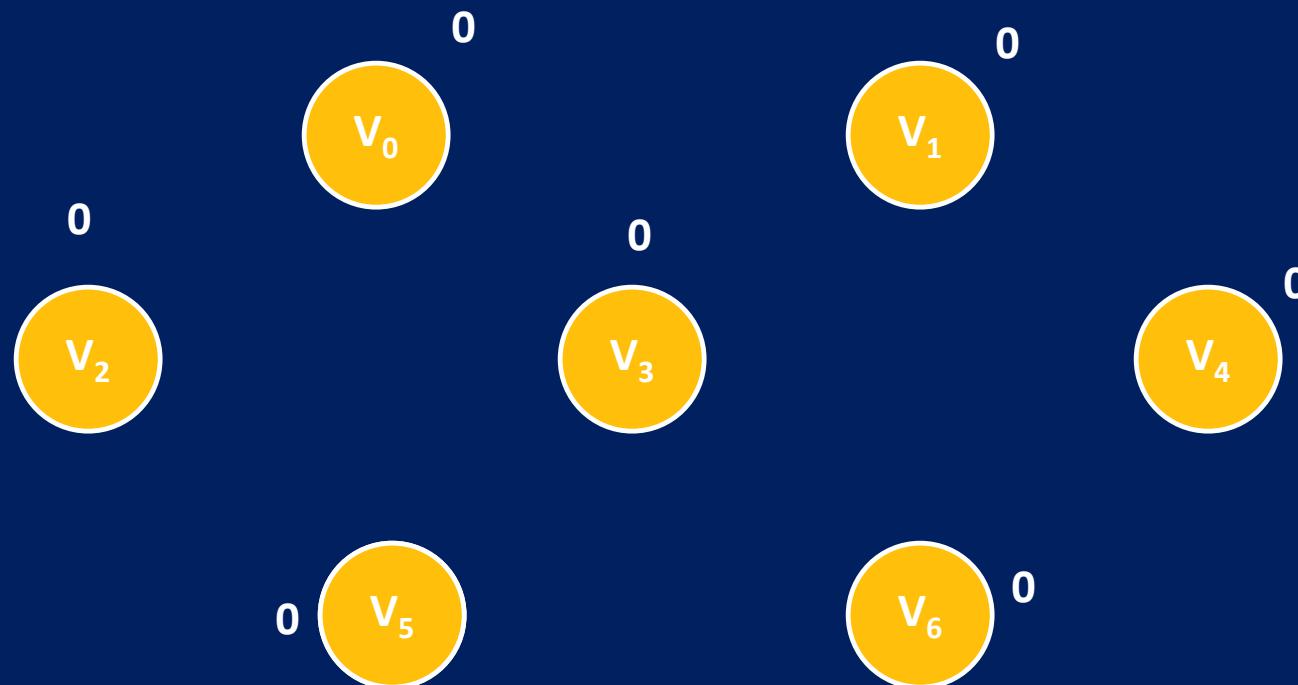
# Topological Sorting



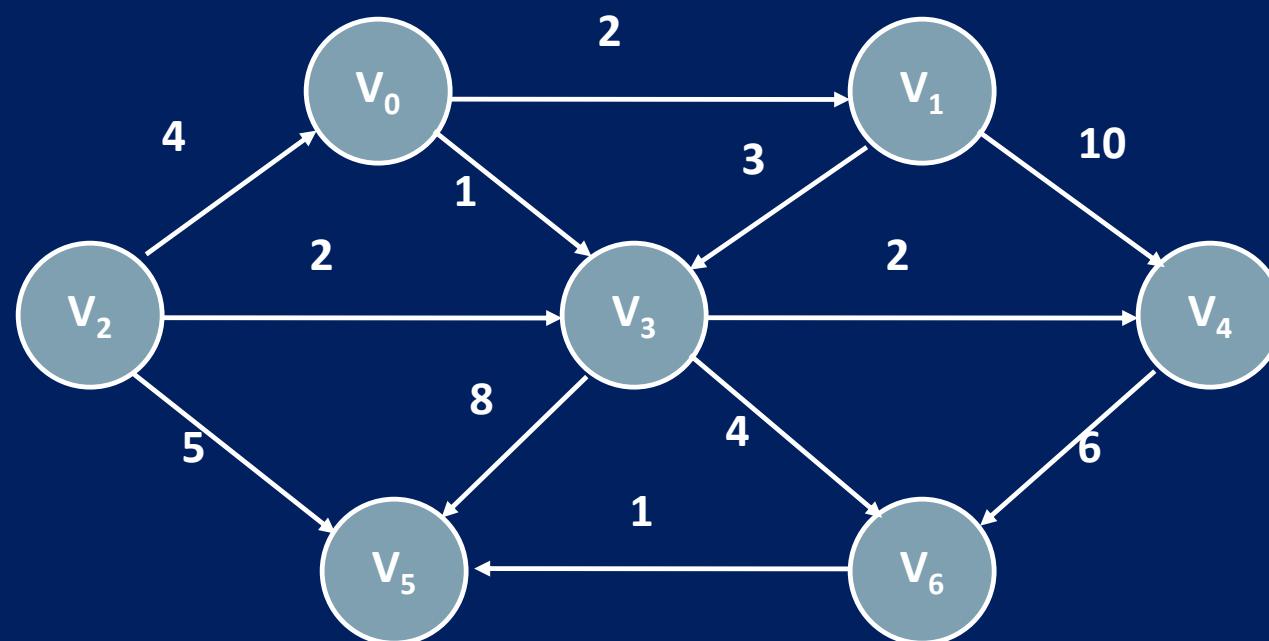
# Topological Sorting



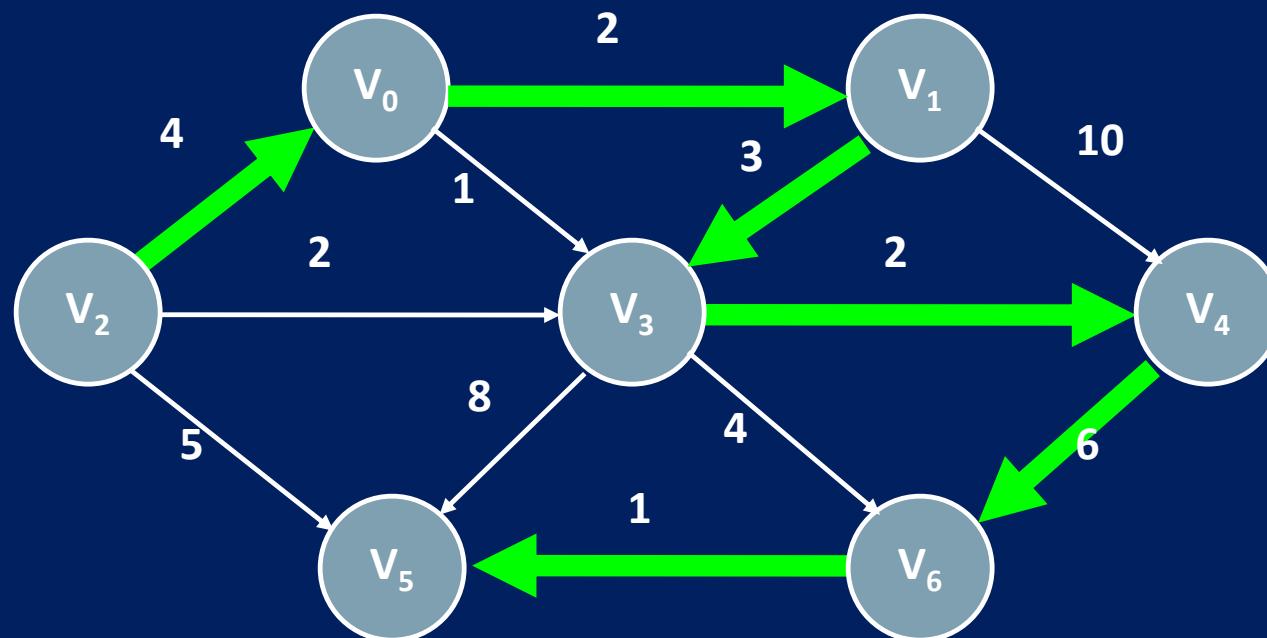
# Topological Sorting



# Topological Sorting

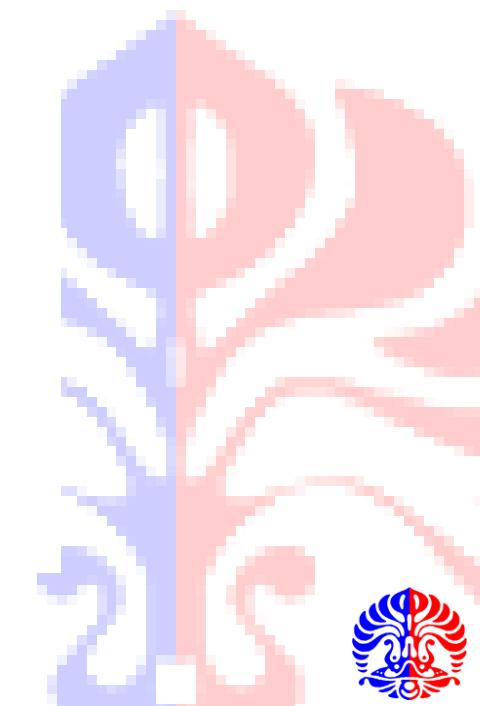
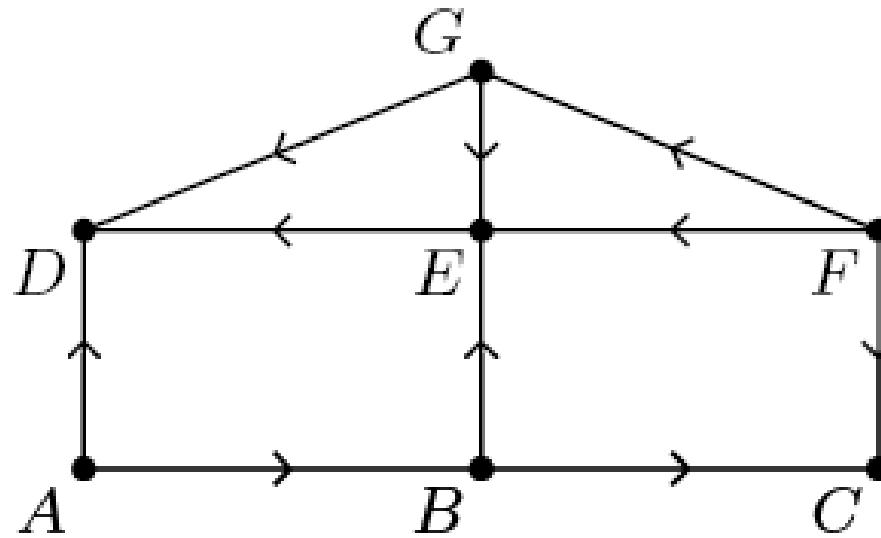


# Topological Sorting



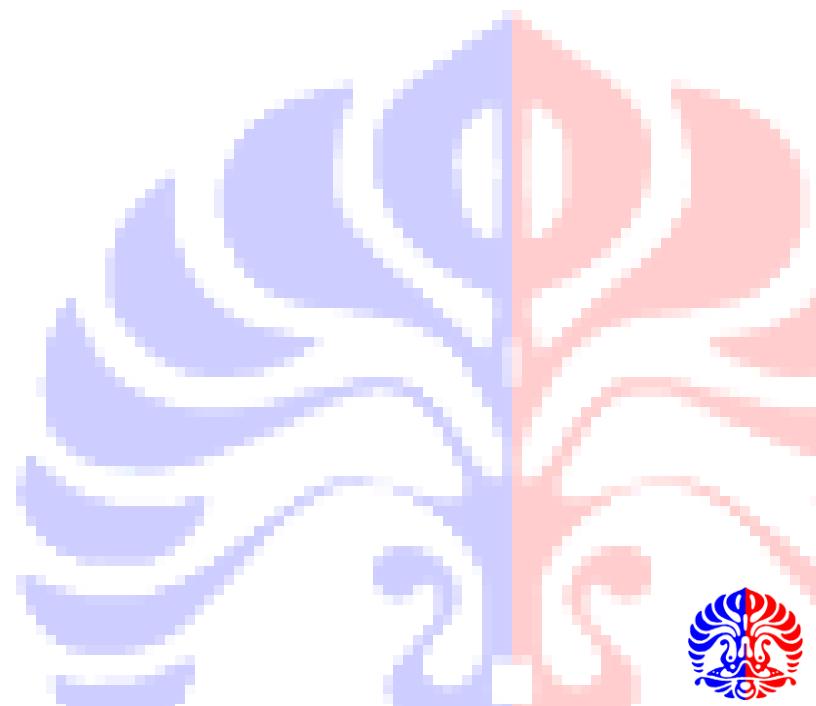
# Latihan

- Lakukan topological sorting pada graph berikut ini, apabila ada pilihan node yang memiliki *in-degree* 0, pilih berdasarkan urutan alfabet.



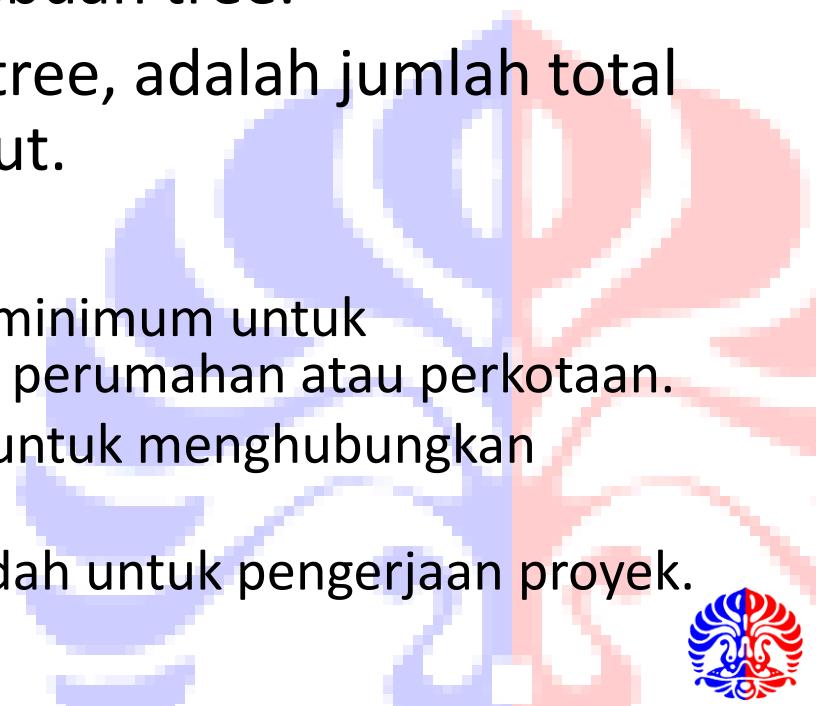
# Jawaban

- Urutan pengeraan:
  - A – B – F – C – G – E – D

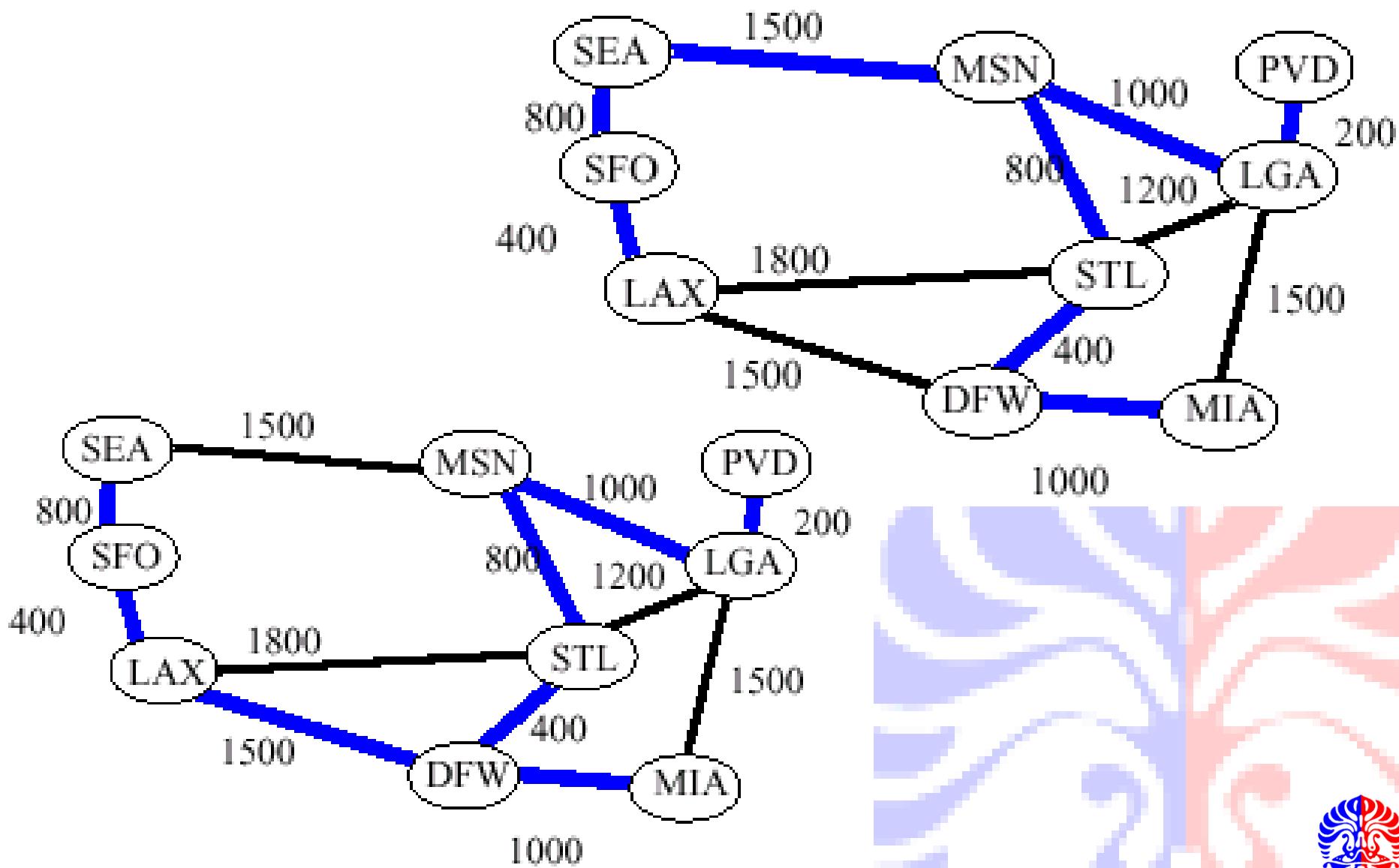


# Minimum Spanning Tree (MST)

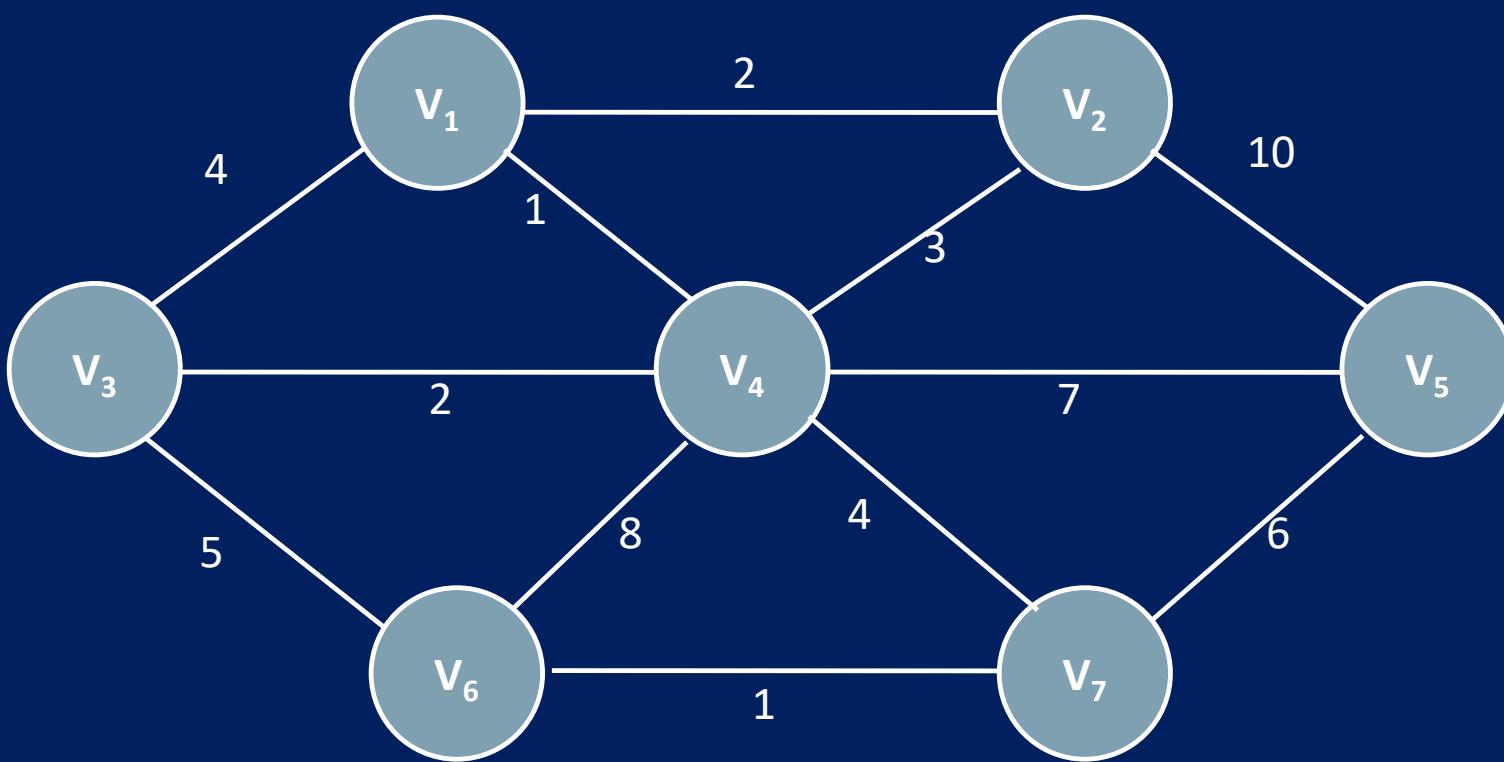
- Adalah sebuah struktur *tree* yang terbentuk dari graph, dimana sisi-sisi yang menghubungkan setiap simpul memiliki nilai total paling kecil.
- “Spanning tree”  $T = (V,F)$  dari graph  $G$  adalah graph dengan verteks yang sama dengan  $G$  dan memiliki  $|V|-1$  buah edges, yang membentuk sebuah tree.
- Nilai total dari sebuah spanning tree, adalah jumlah total bobot tiap sisi dalam *tree* tersebut.
- Penerapan:
  - Mencari jumlah biaya kabel paling minimum untuk menghubungkan sebuah kelompok perumahan atau perkotaan.
  - Mencari biaya minimum terendah untuk menghubungkan jaringan komputer.
  - Mencari biaya produksi total terendah untuk penggeraan proyek.



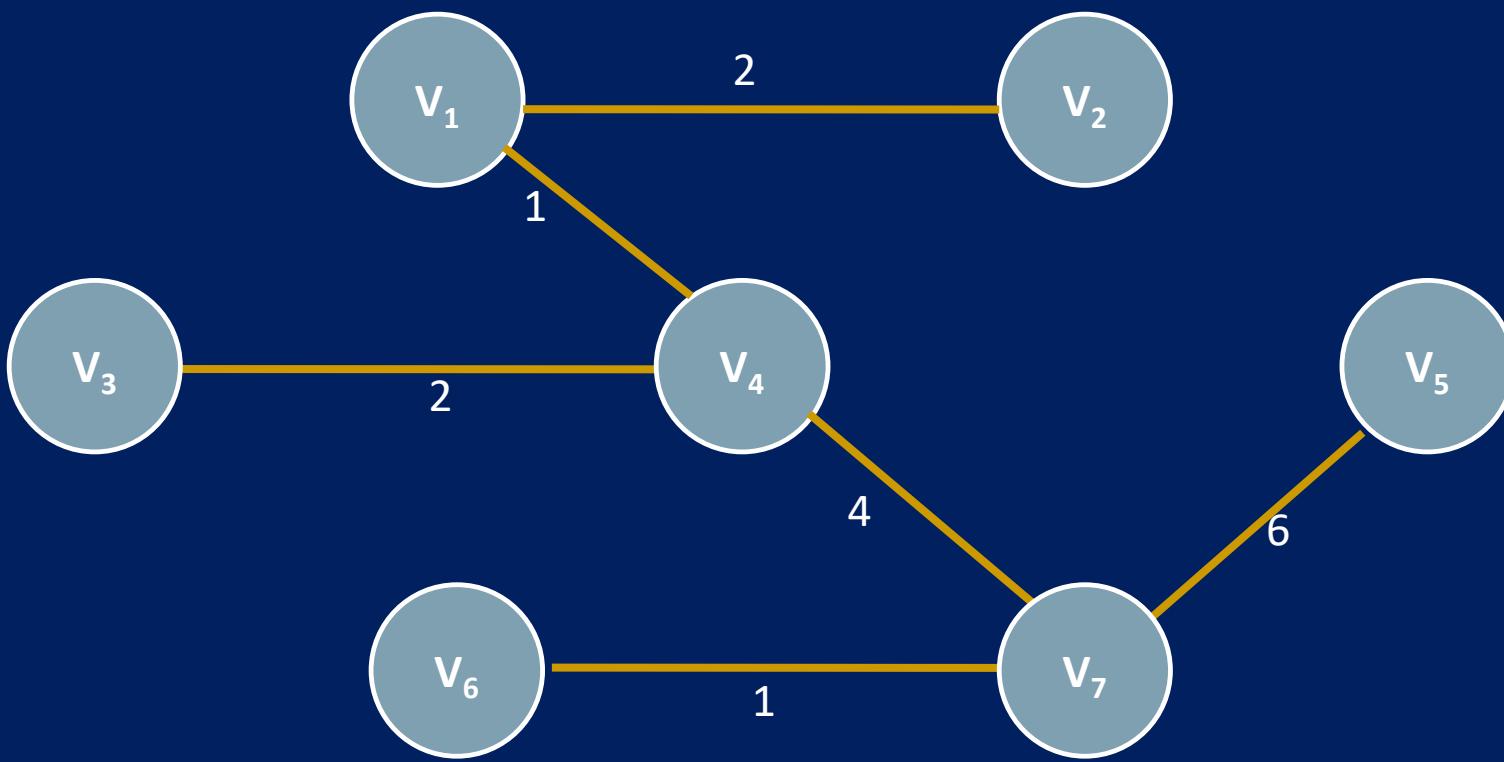
# Minimum Spanning Tree (MST)



# Minimum Spanning Tree: a graph



# Minimum Spanning Tree



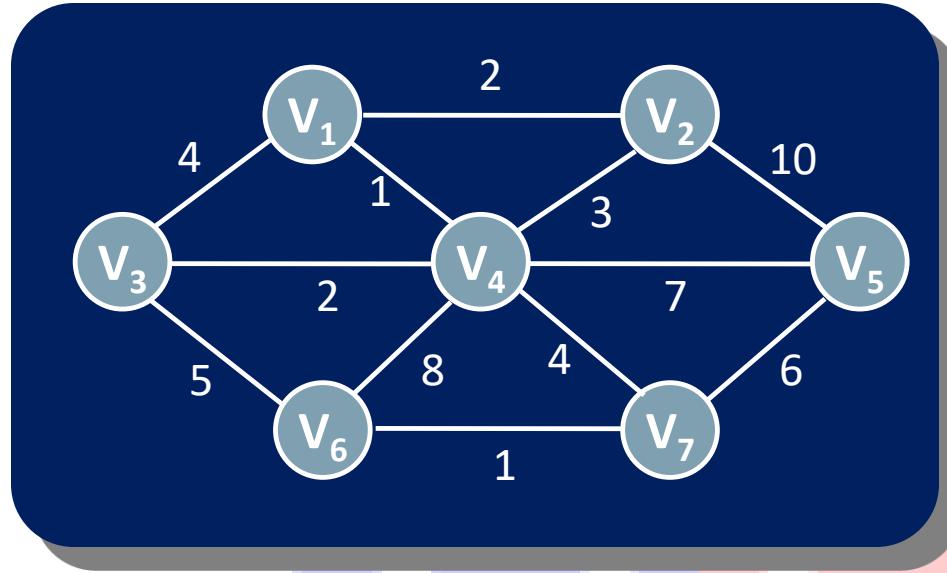
# Prim's Algorithm

- mulai dari sebuah simpul
- bangun tree dengan menambahkan sebuah sisi/busur satu persatu.
  - secara berulang pilih sisi terkecil yang dapat menyambung tree.
- greedy algorithms:
  - Pilihan langkah diambil berdasarkan pilihan terbaik secara local tanpa memperhatikan pengaruhnya secara global.



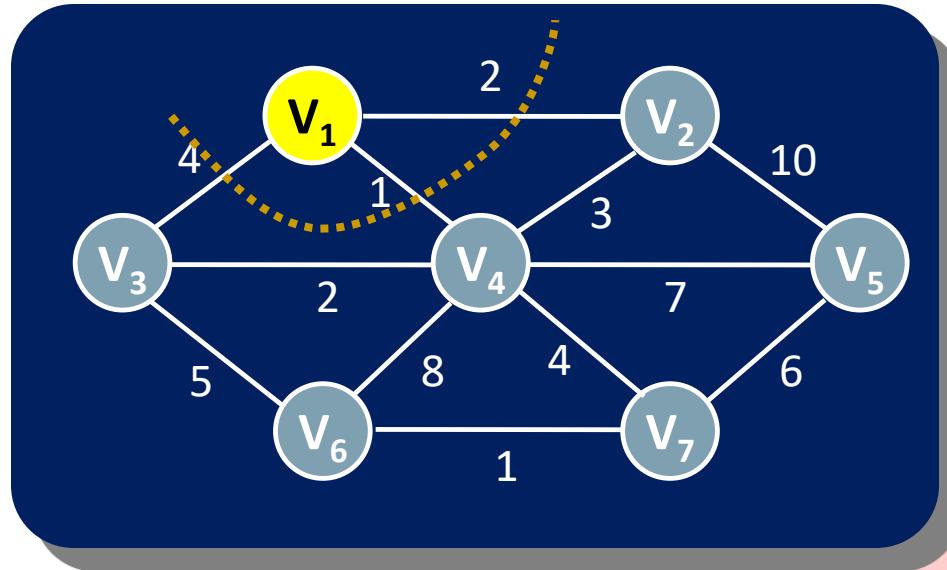
# Prim's Algorithm

<b>V</b>	<b>known</b>	<b>d<sub>v</sub></b>	<b>p<sub>v</sub></b>
<b>V<sub>1</sub></b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>V<sub>2</sub></b>	<b>0</b>	$\infty$	<b>0</b>
<b>V<sub>3</sub></b>	<b>0</b>	$\infty$	<b>0</b>
<b>V<sub>4</sub></b>	<b>0</b>	$\infty$	<b>0</b>
<b>V<sub>5</sub></b>	<b>0</b>	$\infty$	<b>0</b>
<b>V<sub>6</sub></b>	<b>0</b>	$\infty$	<b>0</b>
<b>V<sub>7</sub></b>	<b>0</b>	$\infty$	<b>0</b>



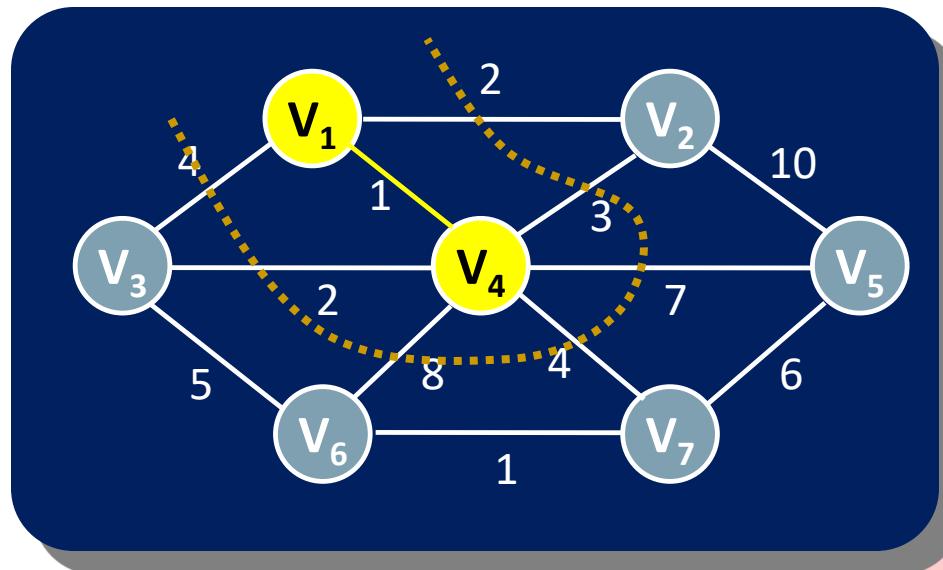
# Prim's Algorithm

<b>v</b>	<b>known</b>	<b>d<sub>v</sub></b>	<b>p<sub>v</sub></b>
$v_1$	1	0	0
$v_2$	0	2	$v_1$
$v_3$	0	4	$v_1$
$v_4$	0	1	$v_1$
$v_5$	0	$\infty$	0
$v_6$	0	$\infty$	0
$v_7$	0	$\infty$	0



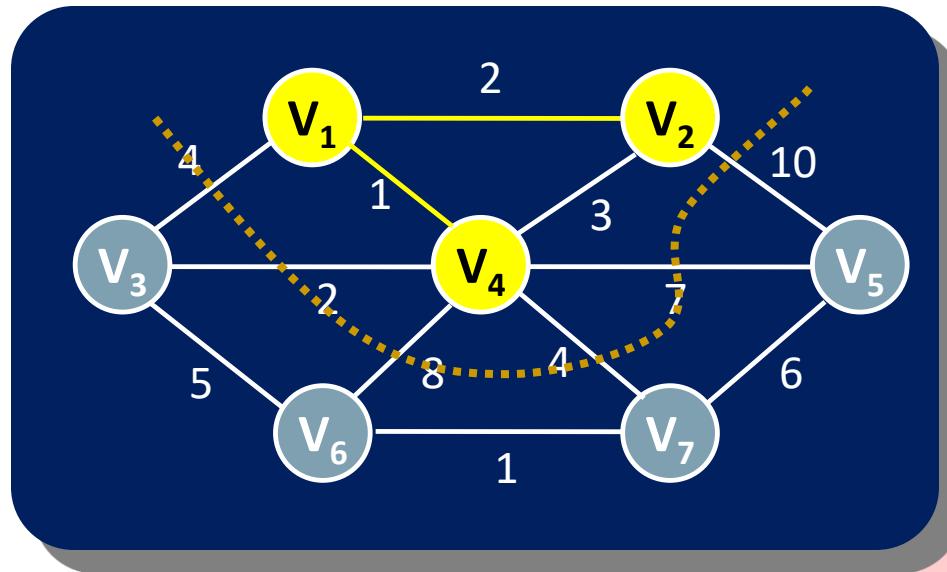
# Prim's Algorithm

$v$	known	$d_v$	$p_v$
$v_1$	1	0	0
$v_2$	0	2	$v_1$
$v_3$	0	2	$v_1$
$v_4$	1	1	$v_1$
$v_5$	0	7	$v_4$
$v_6$	0	8	$v_4$
$v_7$	0	4	$v_4$



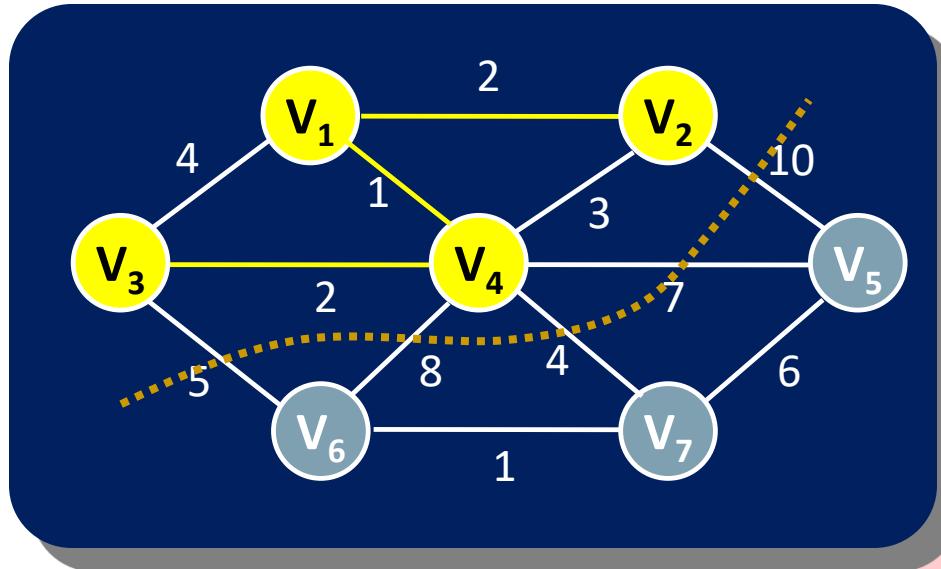
# Prim's Algorithm

$v$	known	$d_v$	$p_v$
$v_1$	1	0	0
$v_2$	1	2	$v_1$
$v_3$	0	2	$v_1$
$v_4$	1	1	$v_1$
$v_5$	0	7	$v_4$
$v_6$	0	8	$v_4$
$v_7$	0	4	$v_4$



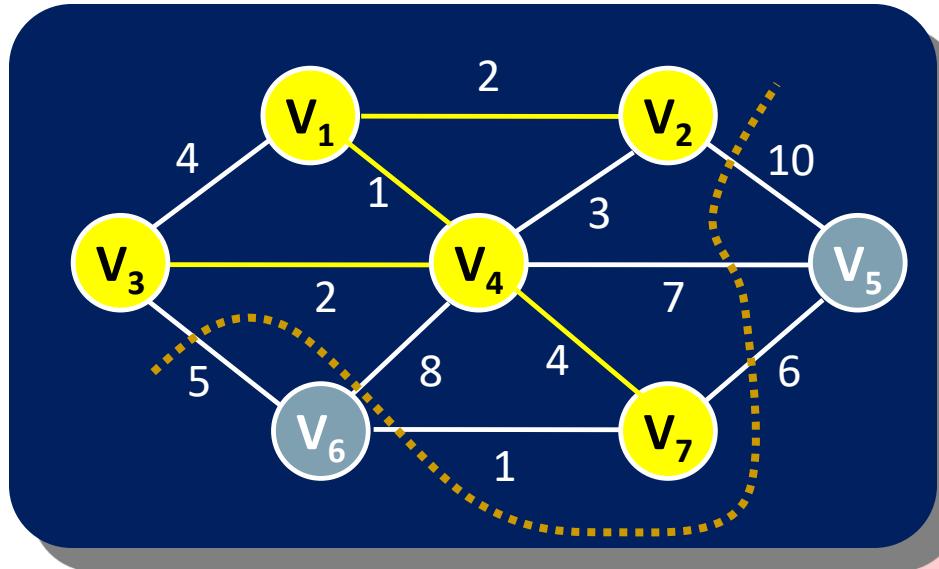
# Prim's Algorithm

<b>v</b>	<b>known</b>	<b>d<sub>v</sub></b>	<b>p<sub>v</sub></b>
$v_1$	1	0	0
$v_2$	1	2	$v_1$
$v_3$	1	2	$v_1$
$v_4$	1	1	$v_1$
$v_5$	0	7	$v_4$
$v_6$	0	5	$v_3$
$v_7$	0	4	$v_4$



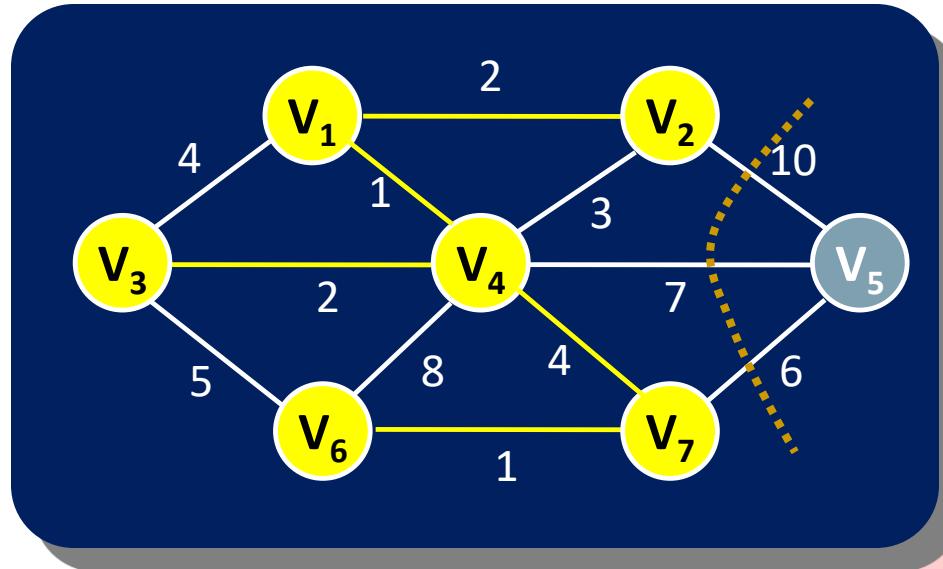
# Prim's Algorithm

$v$	known	$d_v$	$p_v$
$v_1$	1	0	0
$v_2$	1	2	$v_1$
$v_3$	1	2	$v_1$
$v_4$	1	1	$v_1$
$v_5$	0	6	$v_7$
$v_6$	0	1	$v_7$
$v_7$	1	4	$v_4$



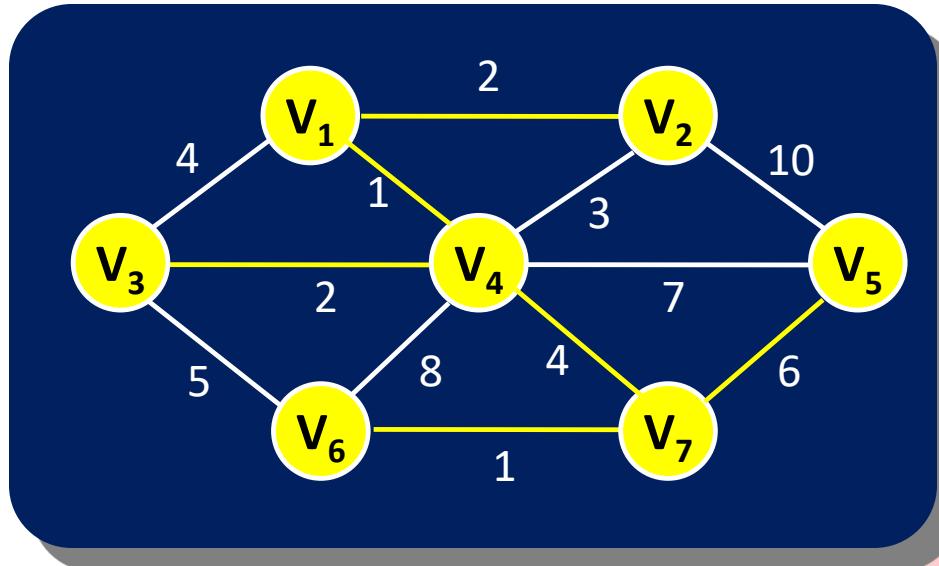
# Prim's Algorithm

<b>v</b>	<b>known</b>	<b>d<sub>v</sub></b>	<b>p<sub>v</sub></b>
$v_1$	1	0	0
$v_2$	1	2	$v_1$
$v_3$	1	2	$v_1$
$v_4$	1	1	$v_1$
$v_5$	0	6	$v_7$
$v_6$	1	1	$v_7$
$v_7$	1	4	$v_4$



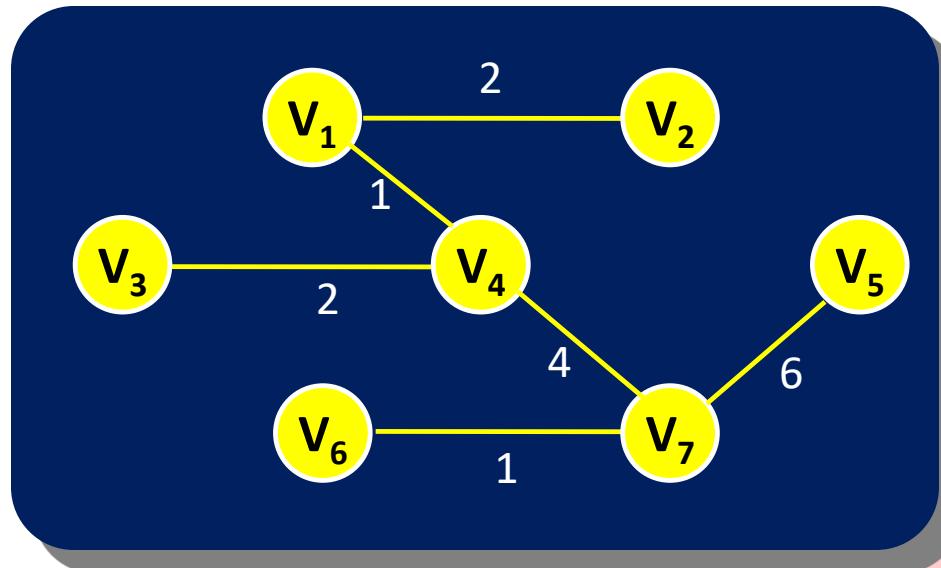
# Prim's Algorithm

<b>v</b>	<b>known</b>	<b>d<sub>v</sub></b>	<b>p<sub>v</sub></b>
$v_1$	1	0	0
$v_2$	1	2	$v_1$
$v_3$	1	2	$v_4$
$v_4$	1	1	$v_1$
$v_5$	1	6	$v_7$
$v_6$	1	1	$v_7$
$v_7$	1	4	$v_4$



# Prim's Algorithm

<b>v</b>	<b>known</b>	<b>d<sub>v</sub></b>	<b>p<sub>v</sub></b>
$v_1$	1	0	0
$v_2$	1	2	$v_1$
$v_3$	1	2	$v_4$
$v_4$	1	1	$v_1$
$v_5$	1	6	$v_7$
$v_6$	1	1	$v_7$
$v_7$	1	4	$v_4$



# Kruskal's Algorithm

- Dari sebuah graph  $G = (V,E)$ , buatlah graph baru  $T$  dengan verteks yang sama dengan  $G$  namun belum memiliki edges.
- List semua *edges* yang terdapat pada  $G$ , urutkan berdasarkan bobot, dari yang terkecil hingga yang terbesar
- Lakukan iterasi untuk setiap *edge* secara terurut. Untuk setiap *edge*  $(v,u)$ :
  - Jika  $u$  dan  $v$  tidak terhubung oleh suatu *path* pada  $T$ , tambahkan  $(u,v)$  ke dalam  $T$ , atau dengan kata lain tambahkan edge ke dalam graph  $T$  **apabila tidak menimbulkan cycle.**

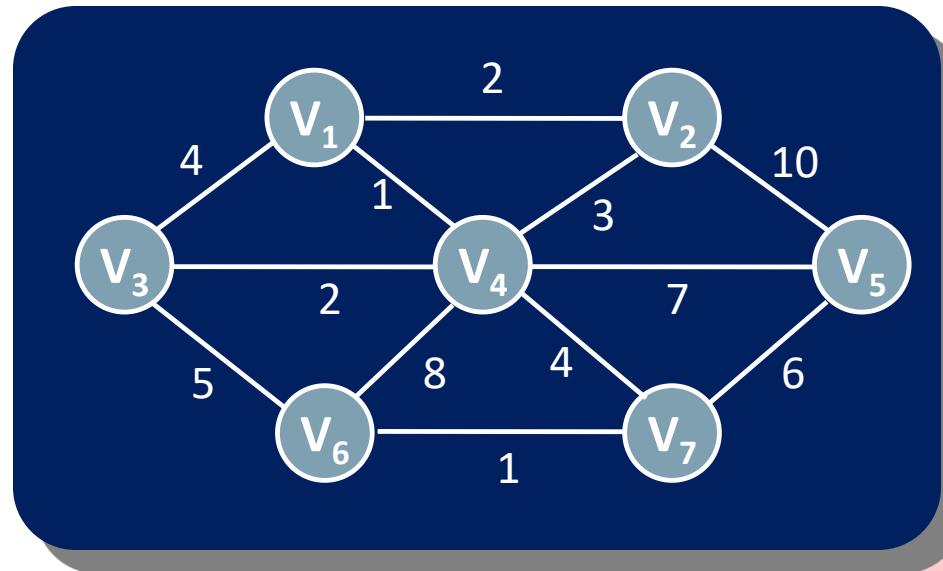
**Iterasi dilakukan hingga semua verteks terhubung (jumlah edge = jumlah verteks – 1)**

- Pemeriksaan cycle dapat menggunakan struktur data “Union-Find Disjoint Sets”
- Graph  $T$  yang terbentuk merupakan MST dari graph  $G$ .



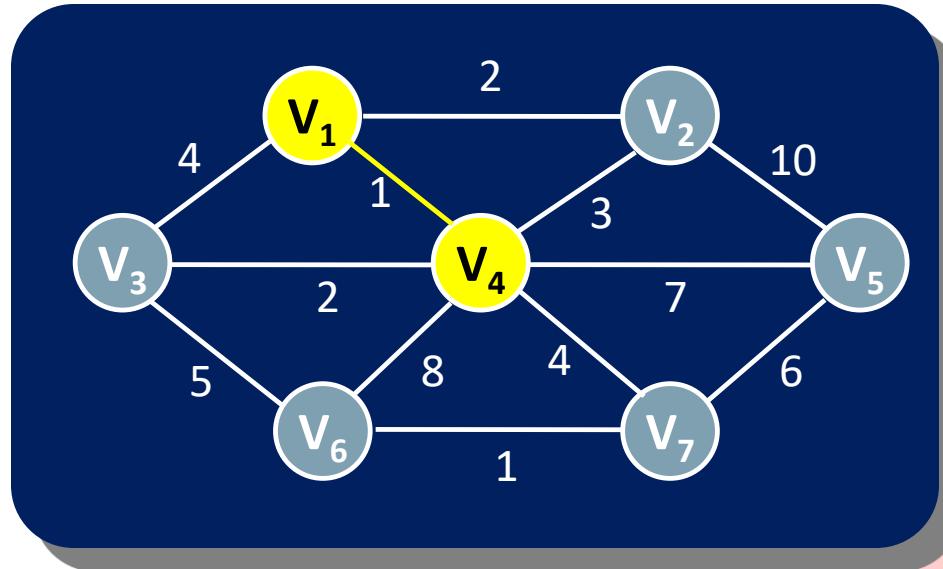
# Kruskal's Algorithm

Edge	Weight	Action
$(V_1, V_4)$	1	-
$(V_6, V_7)$	1	-
$(V_1, V_2)$	2	-
$(V_3, V_4)$	2	-
$(V_2, V_4)$	3	-
$(V_1, V_3)$	4	-
$(V_4, V_7)$	4	-
$(V_3, V_6)$	5	-
$(V_5, V_7)$	6	-



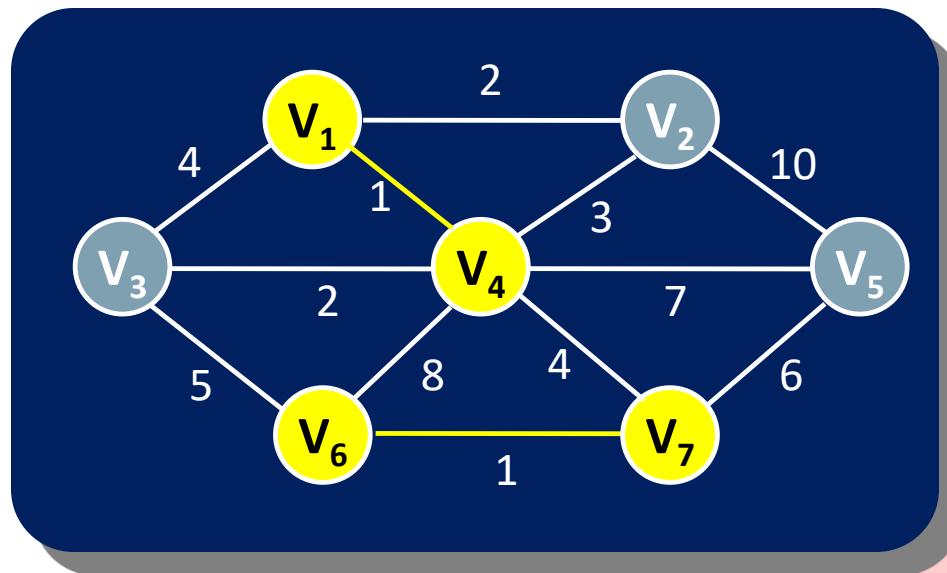
# Kruskal's Algorithm

Edge	Weight	Action
$(V_1, V_4)$	1	A
$(V_6, V_7)$	1	-
$(V_1, V_2)$	2	-
$(V_3, V_4)$	2	-
$(V_2, V_4)$	3	-
$(V_1, V_3)$	4	-
$(V_4, V_7)$	4	-
$(V_3, V_6)$	5	-
$(V_5, V_7)$	6	-



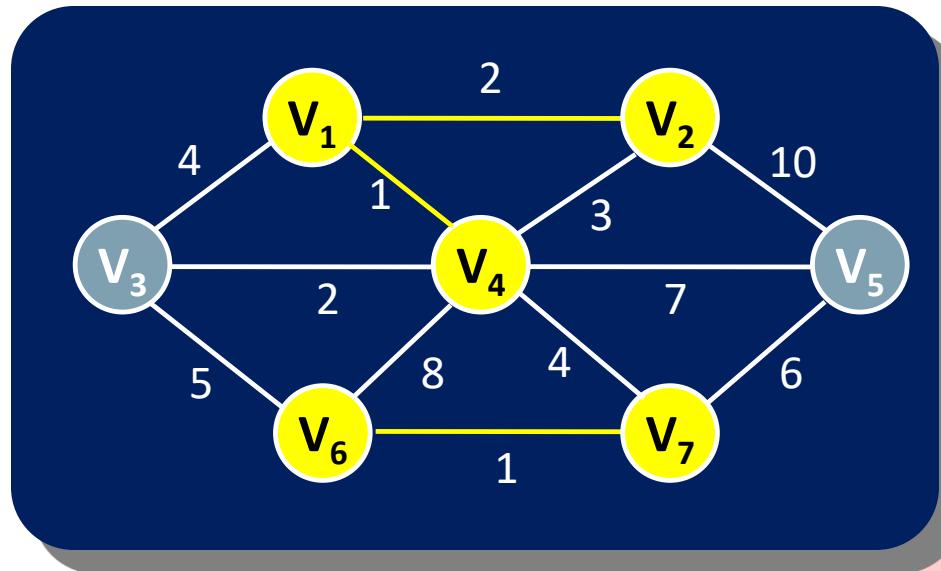
# Kruskal's Algorithm

Edge	Weight	Action
$(V_1, V_4)$	1	A
$(V_6, V_7)$	1	A
$(V_1, V_2)$	2	-
$(V_3, V_4)$	2	-
$(V_2, V_4)$	3	-
$(V_1, V_3)$	4	-
$(V_4, V_7)$	4	-
$(V_3, V_6)$	5	-
$(V_5, V_7)$	6	-



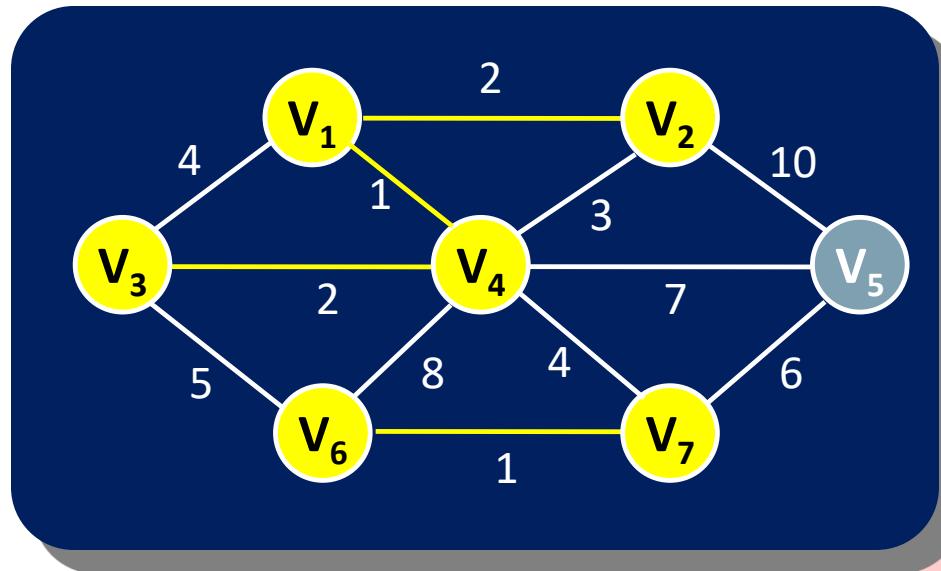
# Kruskal's Algorithm

Edge	Weight	Action
$(V_1, V_4)$	1	A
$(V_6, V_7)$	1	A
$(V_1, V_2)$	2	A
$(V_3, V_4)$	2	-
$(V_2, V_4)$	3	-
$(V_1, V_3)$	4	-
$(V_4, V_7)$	4	-
$(V_3, V_6)$	5	-
$(V_5, V_7)$	6	-



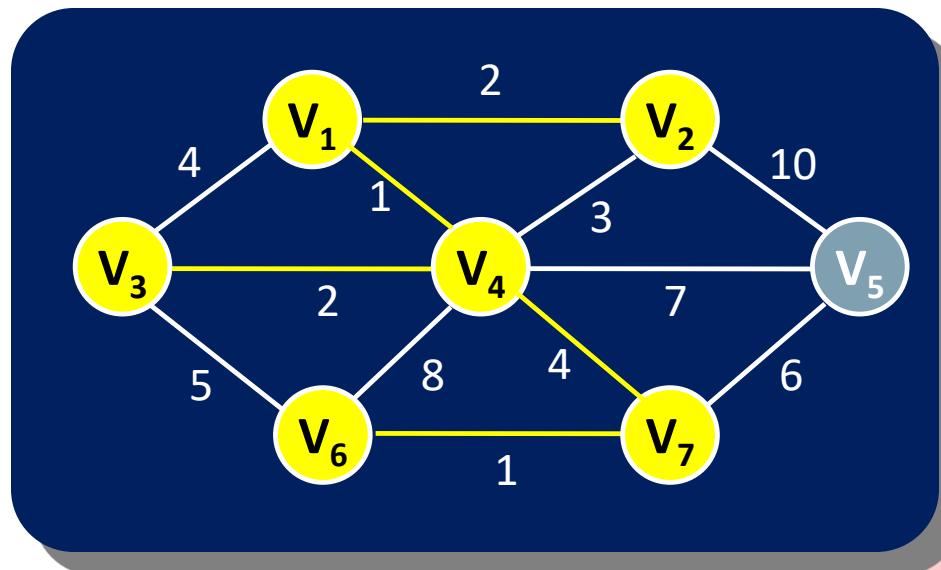
# Kruskal's Algorithm

Edge	Weight	Action
$(V_1, V_4)$	1	A
$(V_6, V_7)$	1	A
$(V_1, V_2)$	2	A
$(V_3, V_4)$	2	A
$(V_2, V_4)$	3	-
$(V_1, V_3)$	4	-
$(V_4, V_7)$	4	-
$(V_3, V_6)$	5	-
$(V_5, V_7)$	6	-



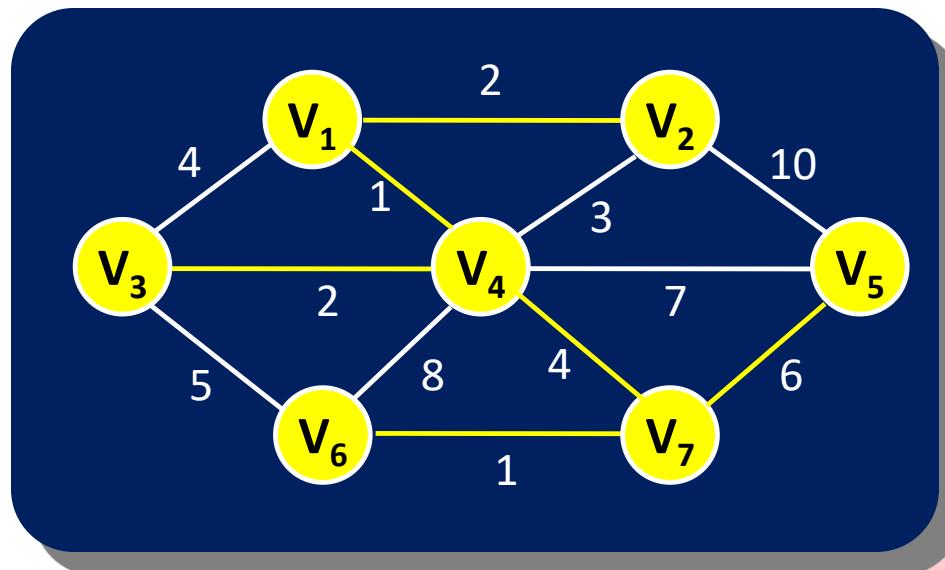
# Kruskal's Algorithm

Edge	Weight	Action
$(V_1, V_4)$	1	A
$(V_6, V_7)$	1	A
$(V_1, V_2)$	2	A
$(V_3, V_4)$	2	A
$(V_2, V_4)$	3	R
$(V_1, V_3)$	4	R
$(V_4, V_7)$	4	A
$(V_3, V_6)$	5	-
$(V_5, V_7)$	6	-



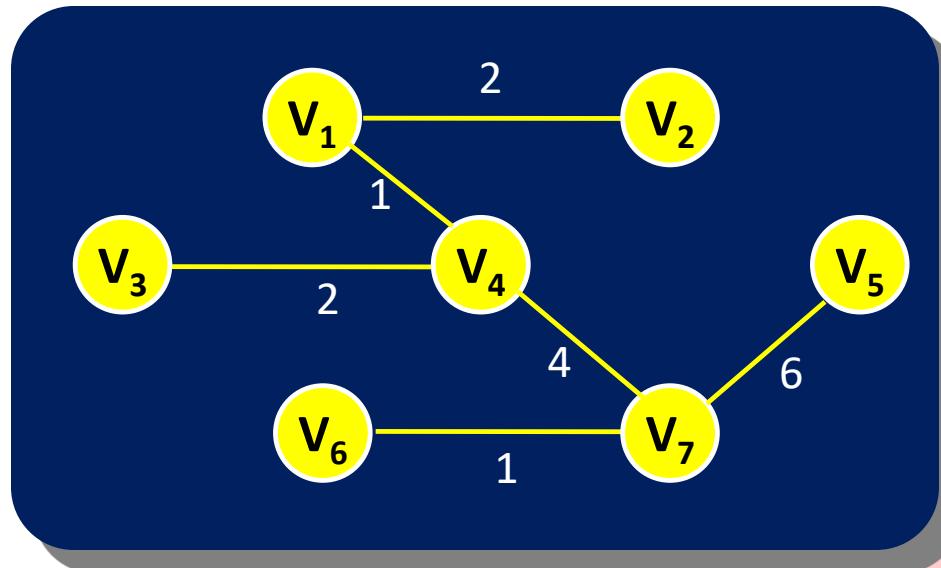
# Kruskal's Algorithm

Edge	Weight	Action
(V1, V4)	1	A
(V6, V7)	1	A
(V1, V2)	2	A
(V3, V4)	2	A
(V2, V4)	3	R
(V1, V3)	4	R
(V4, V7)	4	A
(V3, V6)	5	R
(V5, V7)	6	A



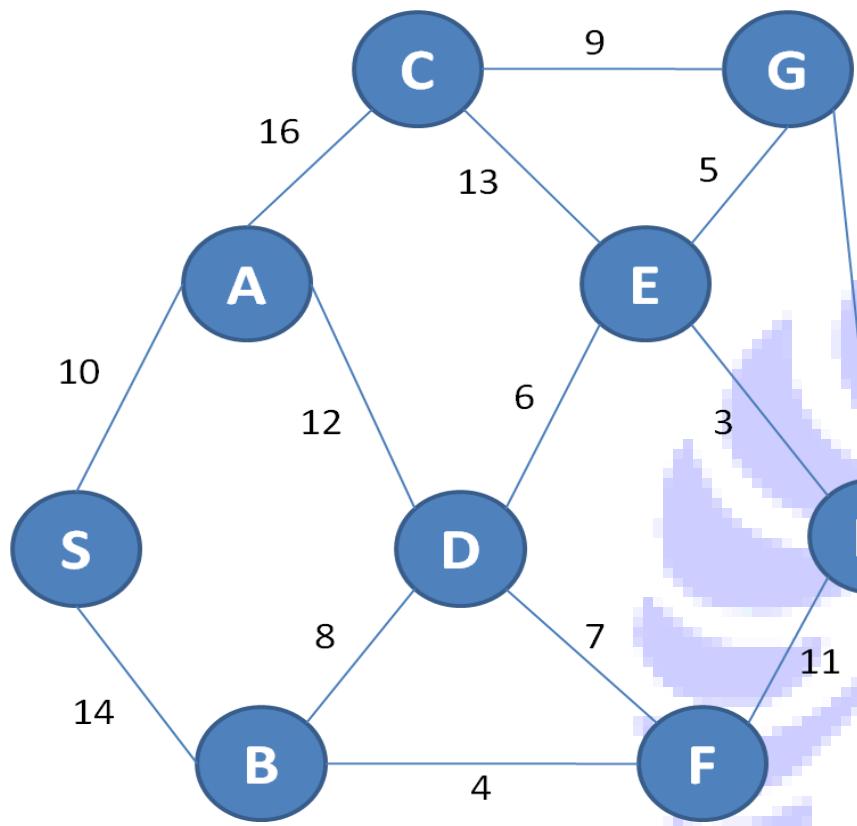
# Kruskal's Algorithm

Edge	Weight	Action
(V1, V4)	1	A
(V6, V7)	1	A
(V1, V2)	2	A
(V3, V4)	2	A
(V2, V4)	3	R
(V1, V3)	4	R
(V4, V7)	4	A
(V3, V6)	5	R
(V5, V7)	6	A

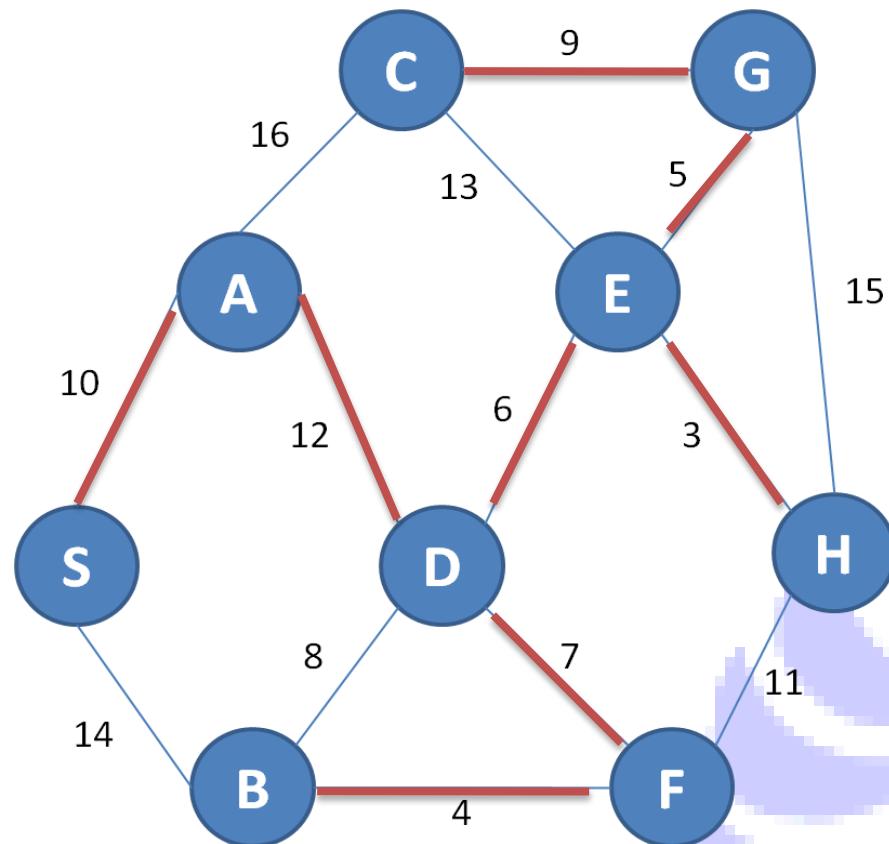


# Latihan

- Gambarkan *minimum spanning tree* yang terbentuk dari graph berikut beserta total bobot minimum yang dicapai. Gunakan Prim's dan Kruskal algorithm!



# Jawaban



# Implementasi Graph

```
1 // Represents an edge in the graph.  
2 class Edge  
3 {  
4     public Vertex dest;          // Second vertex in Edge  
5     public double cost;         // Edge cost  
6  
7     public Edge( Vertex d, double c )  
8     {  
9         dest = d;  
10        cost = c;  
11    }  
12 }
```

**figure 14.6**

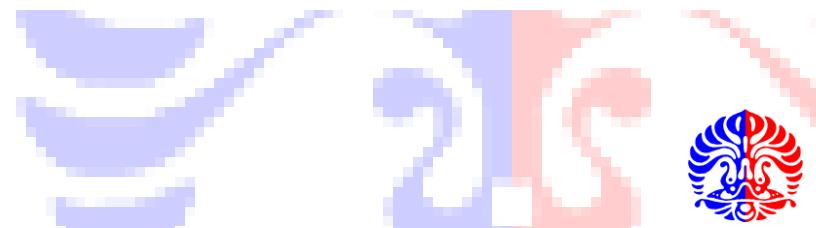
The basic item stored  
in an adjacency list

# Implementasi Graph

```
1 // Represents a vertex in the graph.  
2 class Vertex  
3 {  
4     public String      name;    // Vertex name  
5     public List<Edge> adj;     // Adjacent vertices  
6     public double      dist;    // Cost  
7     public Vertex      prev;    // Previous vertex on shortest path  
8     public int         scratch; // Extra variable used in algorithm  
9  
10    public Vertex( String nm )  
11        { name = nm; adj = new LinkedList<Edge>(); reset( ); }  
12  
13    public void reset( )  
14        { dist = Graph.INFINITY; prev = null; pos = null; scratch = 0; }  
15 }
```

**figure 14.7**

The `Vertex` class stores information for each vertex

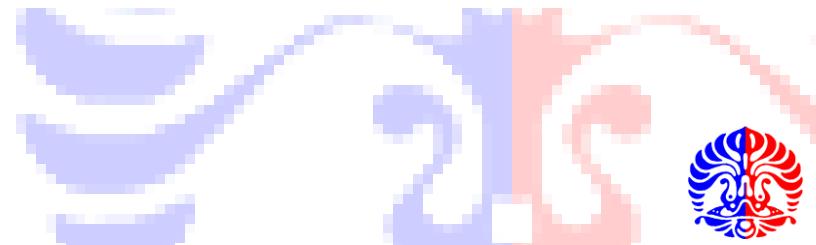


# Implementasi Graph

**figure 14.12**

A recursive routine for printing the shortest path

```
1  /**
2   * Recursive routine to print shortest path to dest
3   * after running shortest path algorithm. The path
4   * is known to exist.
5   */
6  private void printPath( Vertex dest )
7  {
8      if( dest.prev != null )
9      {
10         printPath( dest.prev );
11         System.out.print( " to " );
12     }
13     System.out.print( dest.name );
14 }
```



# Implementasi Graph

```
1 // Represents an entry in the priority queue for Dijkstra's algorithm.
2 class Path implements Comparable<Path>
3 {
4     public Vertex    dest;    // w
5     public double    cost;    // d(w)
6
7     public Path( Vertex d, double c )
8     {
9         dest = d;
10        cost = c;
11    }
12
13    public int compareTo( Path rhs )
14    {
15        double otherCost = rhs.cost;
16
17        return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
18    }
19 }
```

**figure 14.26**

Basic item stored in the priority queue

```

1 /**
2  * Single-source weighted shortest-path algorithm.
3  */
4 public void dijkstra( String startName )
5 {
6     PriorityQueue<Path> pq = new PriorityQueue<Path>();
7
8     Vertex start = vertexMap.get( startName );
9     if( start == null )
10        throw new NoSuchElementException( "Start vertex not found" );
11
12    clearAll( );
13    pq.add( new Path( start, 0 ) ); start.dist = 0;
14
15    int nodesSeen = 0;
16    while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )
17    {
18        Path vrec = pq.remove( );
19        Vertex v = vrec.dest;
20        if( v.scratch != 0 ) // already processed v
21            continue;
22
23        v.scratch = 1;
24        nodesSeen++;
25
26        for( Edge e : v.adj )
27        {
28            Vertex w = e.dest;
29            double cvw = e.cost;
30
31            if( cvw < 0 )
32                throw new GraphException( "Graph has negative edges" );
33
34            if( w.dist > v.dist + cvw )
35            {
36                w.dist = v.dist + cvw;
37                w.prev = v;
38                pq.add( new Path( w, w.dist ) );
39            }
40        }
41    }
42 }

```



**figure 14.27**

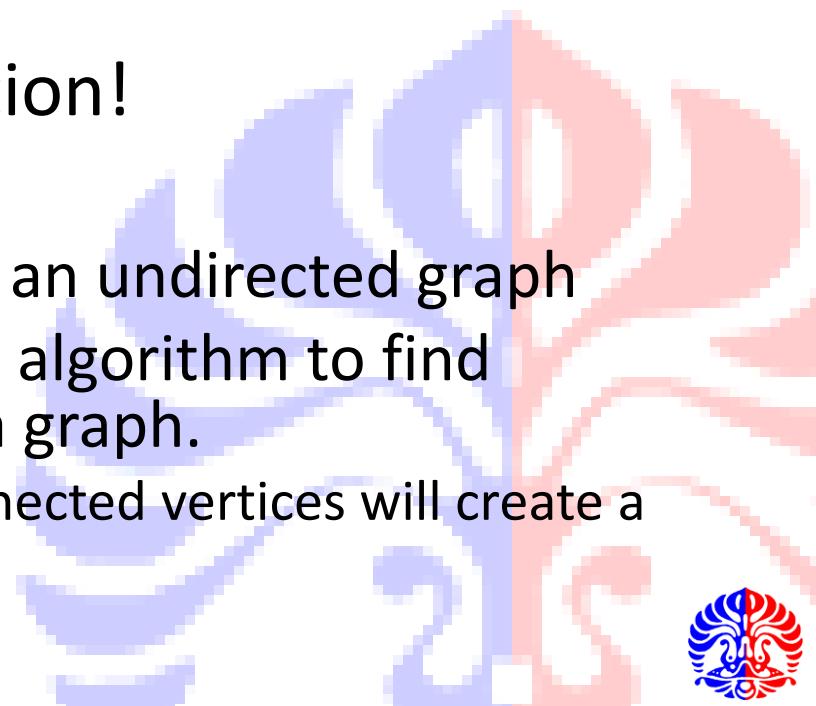
A positive-weighted, shortest-path algorithm: Dijkstra's algorithm

Extra material

# UNION FIND DISJOINT SET

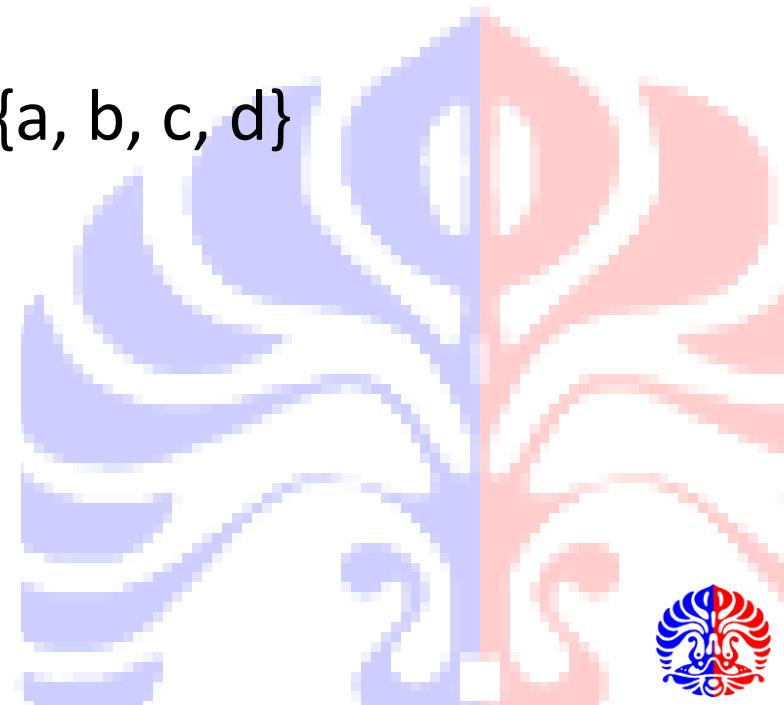
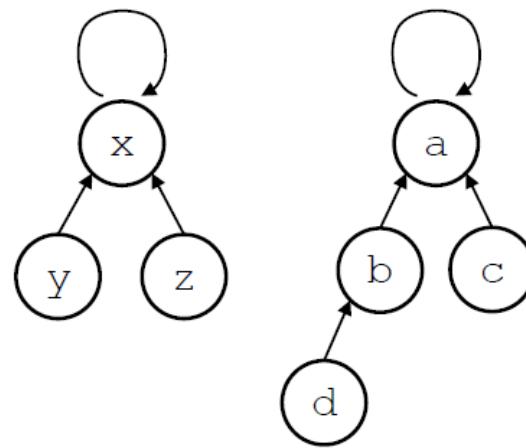
# Union-Find Disjoint Sets

- Can support two types of operations efficiently
  - $\text{find}(x)$ : returns the “representative” of the set that  $x$  belongs
  - $\text{union}(x, y)$ : merges two sets that contain  $x$  and  $y$
- Both operations can be done in (essentially) constant time
- Simple and short implementation!
- Applications:
  - track connected components of an undirected graph
  - used for implementing Kruskal's algorithm to find the minimum spanning tree of a graph.
    - connecting an edge from two connected vertices will create a cycle



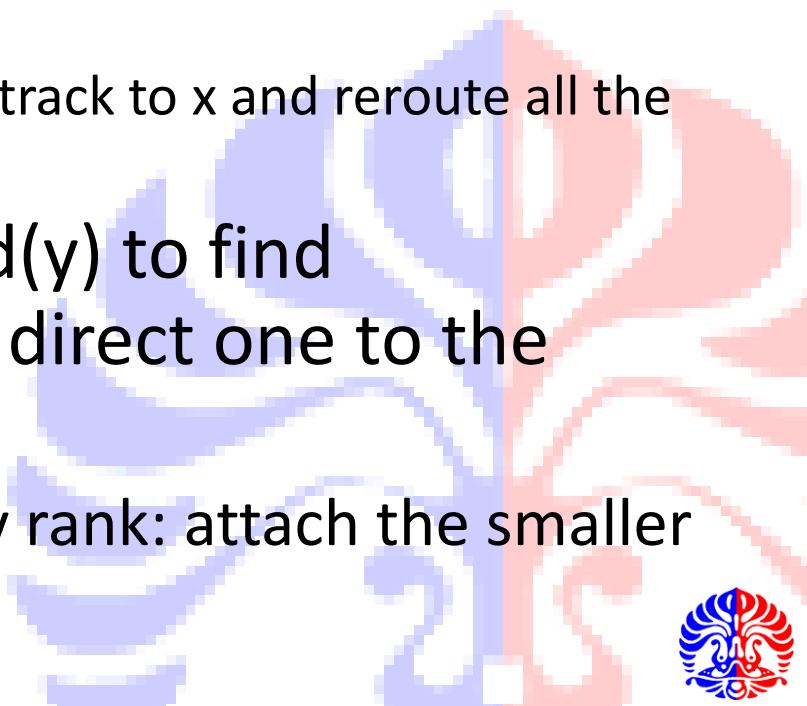
# Data Structure

- Main idea: represent each set by a rooted tree
  - Every node maintains a link to its parent
    - initially, each node points to itself
  - A root node is the “representative” of the corresponding set
  - Example: two sets  $\{x, y, z\}$  and  $\{a, b, c, d\}$



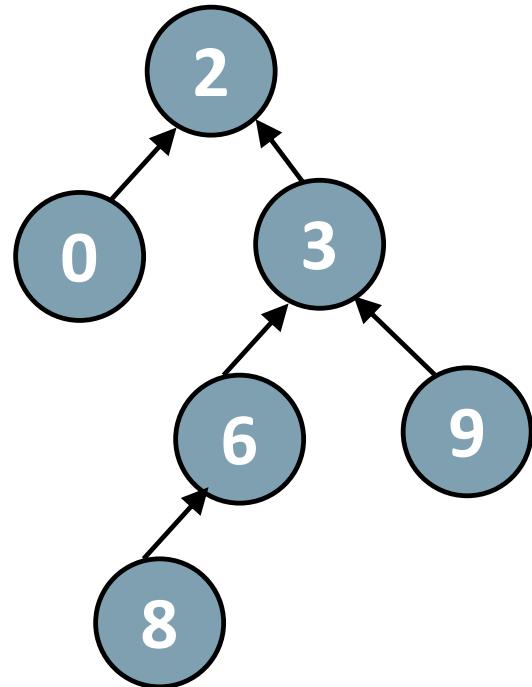
# Implementation

- $\text{find}(x)$ : follow the links from  $x$  until a node points itself (*the representative node*)
  - This can take  $O(n)$  time but we will make it faster
  - Path compression
    - The shape of the tree is not important as long as the root stays the same
    - After  $\text{find}(x)$  returns the root, backtrack to  $x$  and reroute all the links to the root
- $\text{union}(x, y)$ : run  $\text{find}(x)$  and  $\text{find}(y)$  to find corresponding root nodes and direct one to the other
  - Another improvement: Union by rank: attach the smaller tree to the larger tree

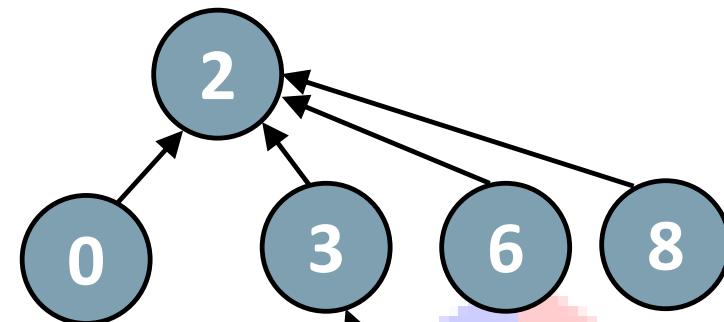


# Path Compression

Initially



After calling findSet (8)



# Java Implementation

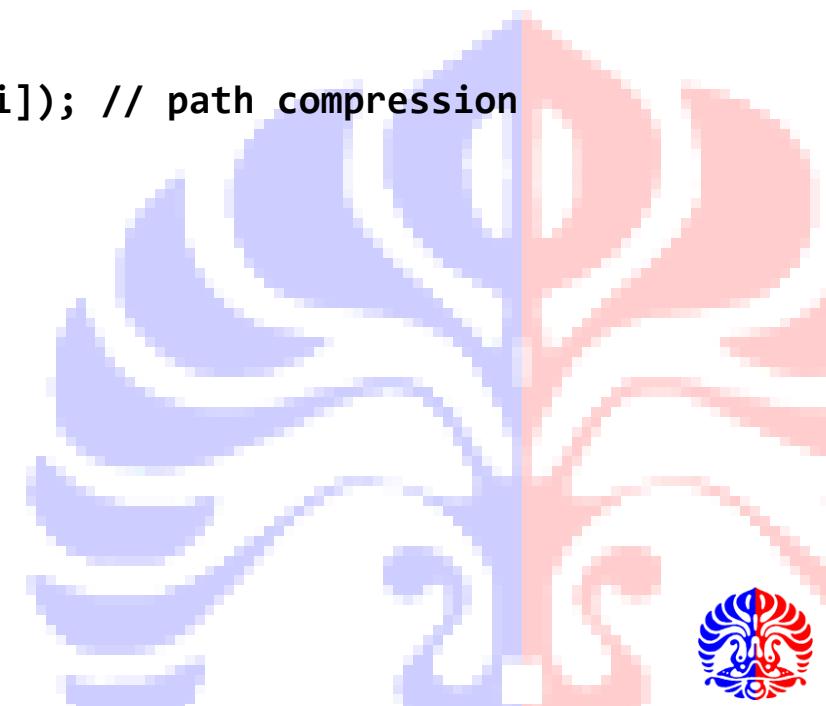
```
public class UnionFindDisjointSet
{
    private int parent[];

    public UnionFindDisjointSet (int size) {
        parent = new int[size];
        for (int ii = 0; ii < size; ii++) {
            parent[ii] = ii;
        }
    }

    public int findSet (int i) {
        if (parent[i] == i) {
            return i;
        } else {
            return parent[i] = findSet (parent[i]); // path compression
        }
    }

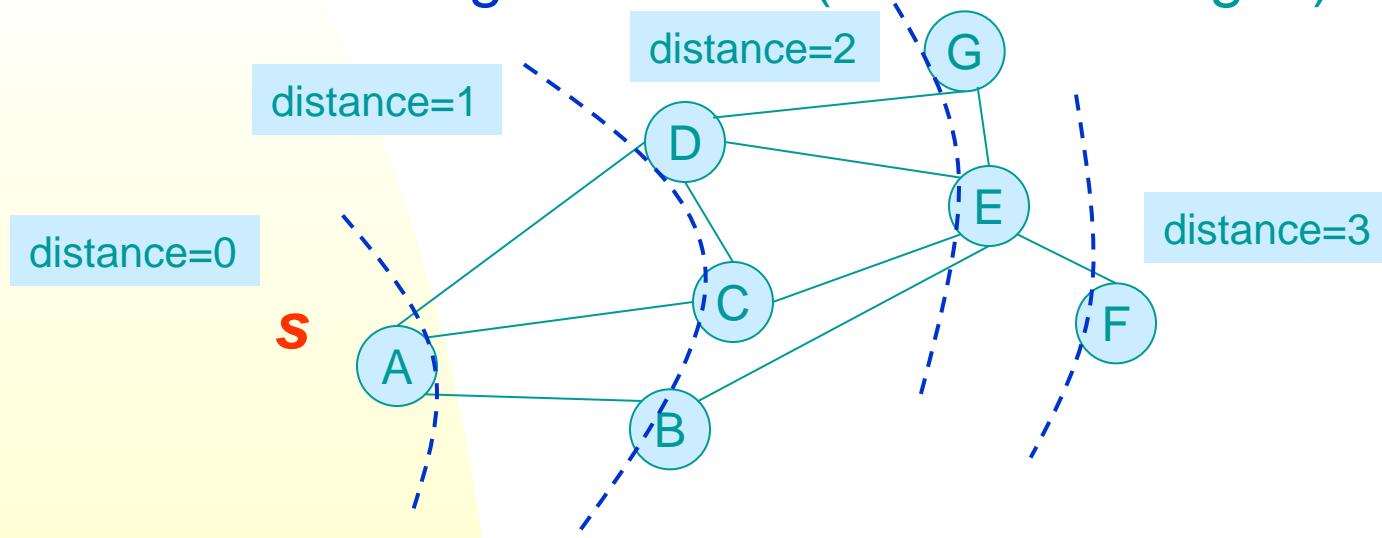
    public void unionSet (int i, int j) {
        parent[findSet (i)] = findSet (j);
    }

    public boolean isSameSet (int i, int j) {
        return findSet (i) == findSet (j);
    }
}
```



# BFS : Breadth-First Search

- Given any source  $s$  (vertex), BFS visits the other vertices at increasing distances (number of edges) from  $s$ .



BFS visit sequence = [A, | D, B, C | G, E, F ]

BFS visit sequence = [A, | C, D, B | E, G, F ]

Breadth-First Search

# BFS : Breadth-First Search

- The situation is pretty much like a water drop falling into a pond.
- At all times, BFS maintains a subset of vertices at the frontier. This frontier moves outward to discover new vertices. Algorithmically, this frontier is maintained as a **queue** (FIFO) of vertices. Those vertices in the queue are waiting to be visited.
- In doing so, BFS discovers **(shortest)** paths from **s** to other vertices.

# Algorithm

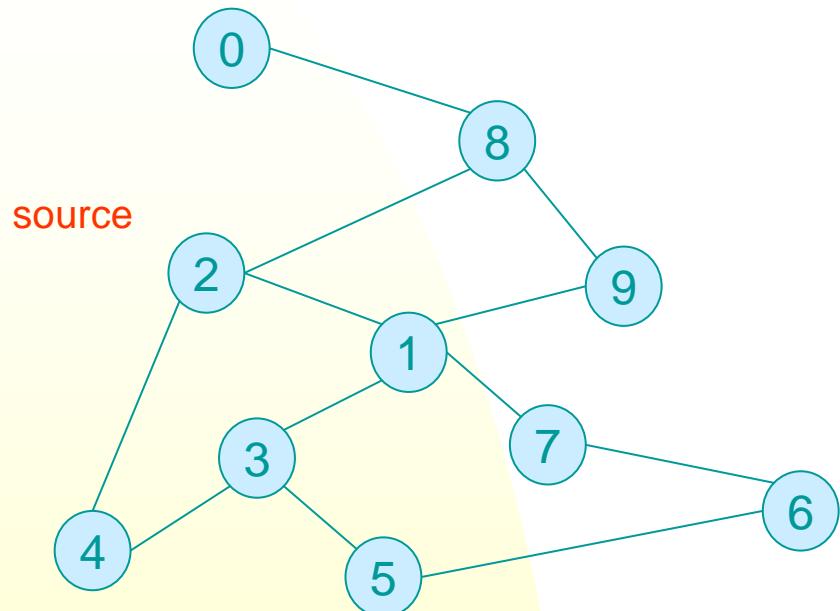
**Algorithm**  $BFS(s)$

**Input:**  $s$  is the source vertex

**Output:** Mark all vertices that can be visited from  $s$ .

1. **for** each vertex  $v$
2.     **do**  $flag[v] := \text{false}$ ; ← initialization
3.      $Q = \text{empty queue}$ ;
4.      $flag[s] := \text{true}$ ;
5.      $\text{enqueue}(Q, s)$ ;
6. **while**  $Q$  is not empty
7.     **do**  $v := \text{dequeue}(Q)$ ;
8.         **for** each  $w$  adjacent to  $v$  ←  $v$  is visited here.
9.             **do if**  $flag[w] = \text{false}$
10.                 **then**  $flag[w] := \text{true}$ ;
11.                  $\text{enqueue}(Q, w)$

# Example



visit sequence ={ }

$Q=\{ \}$

Initialize Q to be empty

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

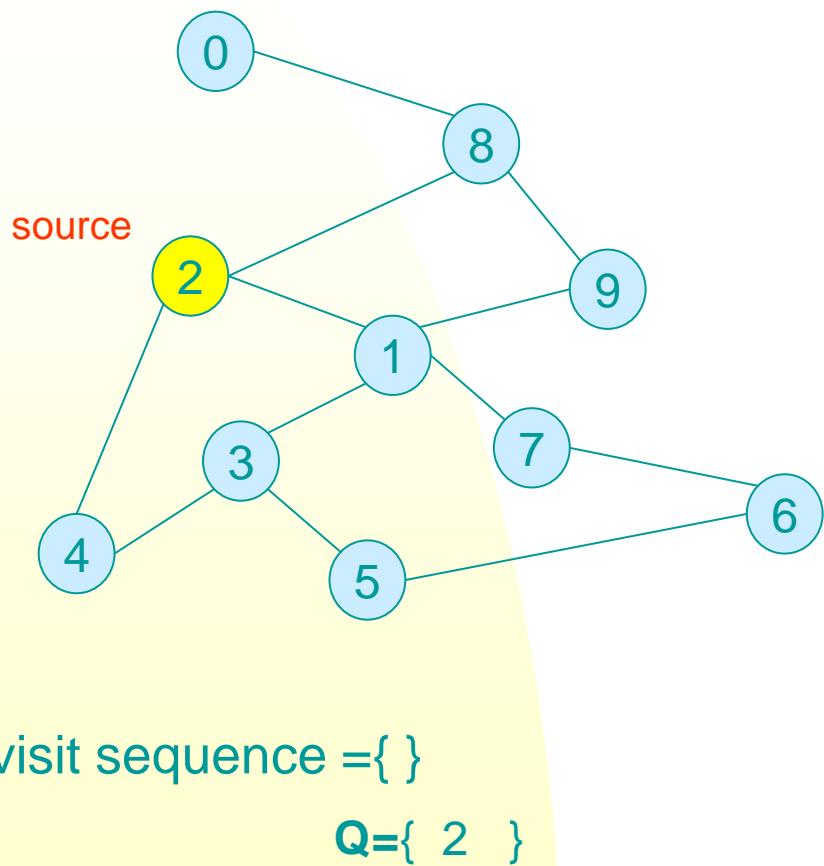
Flag Table (T/F)

0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Initialize visited table (all empty F)

Breadth-First Search

# Example



Place source 2 on the queue.

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

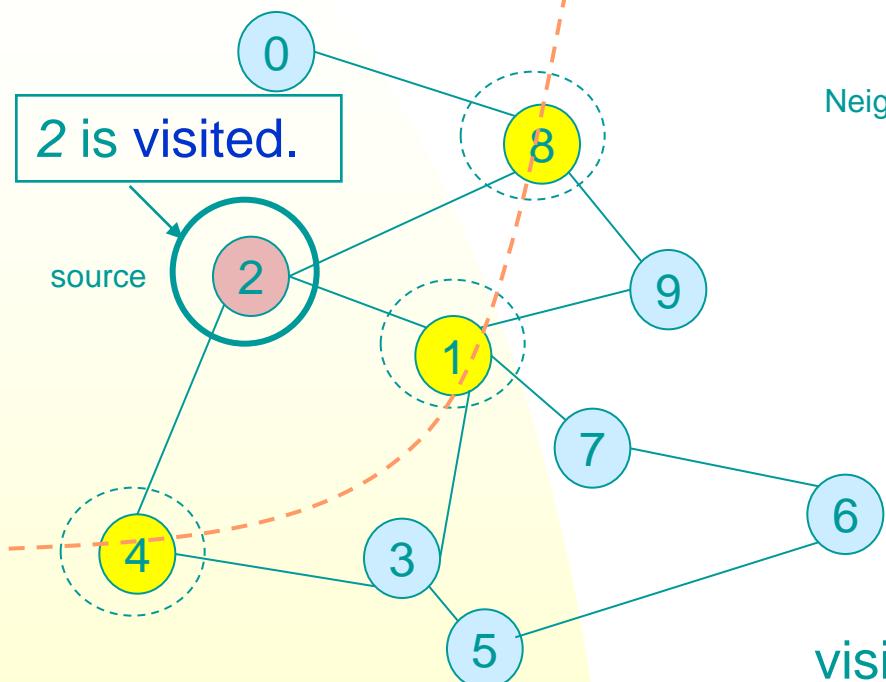


mark Flag[2].

Breadth-First Search

# Example

*a frontier of vertices waiting to be visited (marked yellow)*



$Q=\{2\} \rightarrow \{8, 1, 4\}$  Dequeue 2.

Place all previously unmarked neighbors of 2 on the queue.

Breadth-First Search

Adjacency List

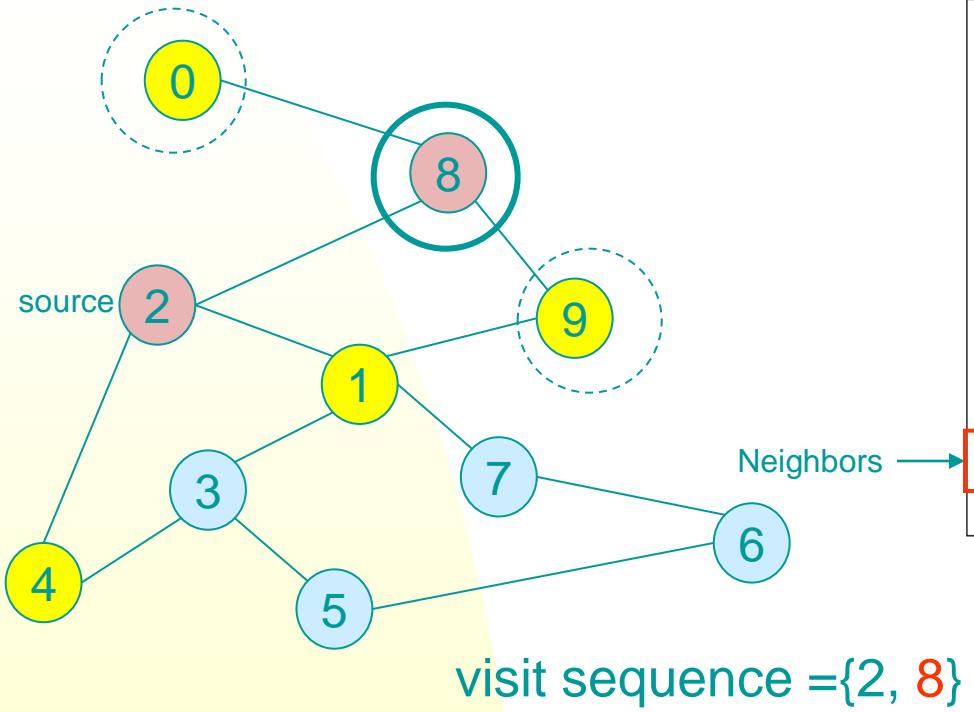
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	F
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	F

Mark unmarked neighbors.

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

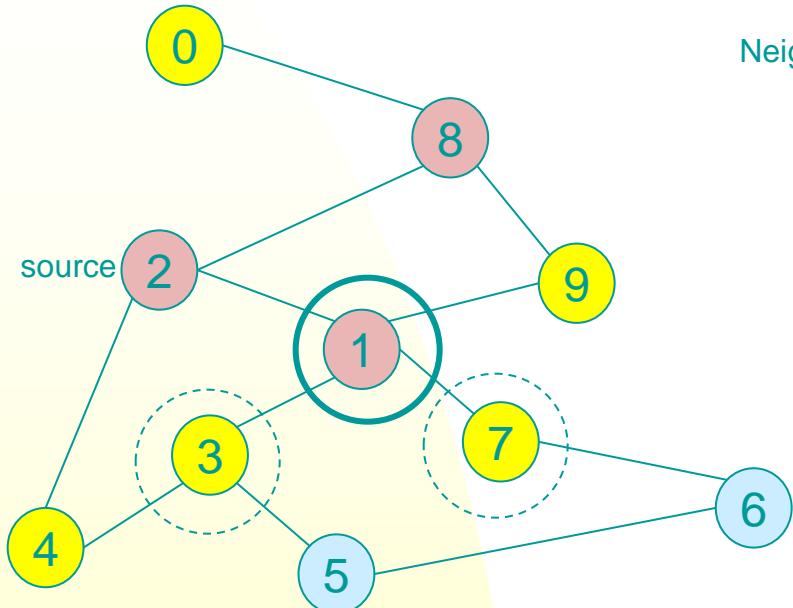
0	T
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	T

Mark unmarked neighbors.

Dequeue 8.

- Place all unmarked neighbors of 8 on the queue.
- Notice that 2 is not placed on the queue again, as it has been marked before!

# Example



$$Q = \{ 1, 4, 0, 9 \} \rightarrow \{ 4, 0, 9, 3, 7 \}$$

Dequeue 1.

- Place all previously unmarked neighbors of 1 on the queue.
- Only nodes 3 and 7 haven't been marked previously.

Neighbors

Adjacency List

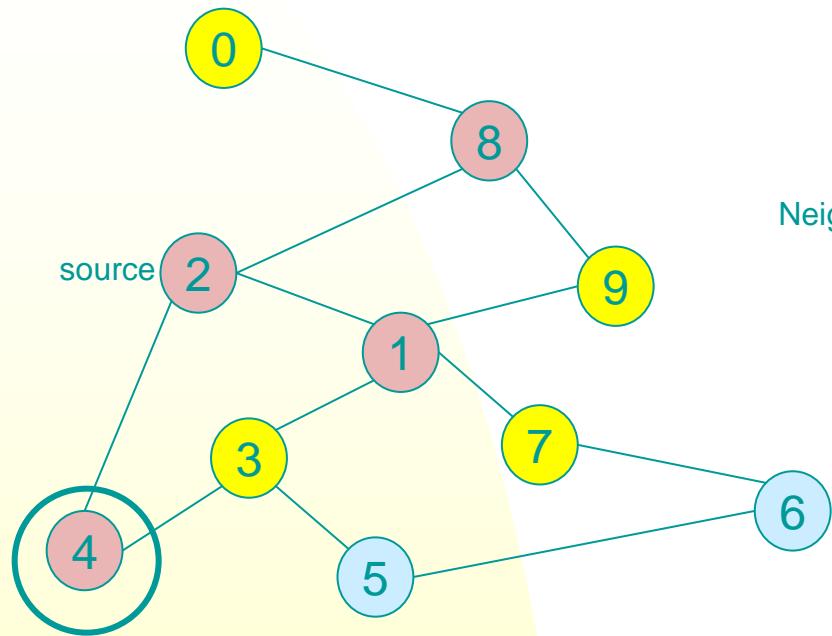
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Mark unmarked neighbors.

# Example



visit sequence = {2, 8, 1, 4}

$$Q = \{ 4, 0, 9, 3, 7 \} \rightarrow \{ 0, 9, 3, 7 \}$$

Dequeue 4.

-- 4 has no unmarked neighbors!

Breadth-First Search

Adjacency List

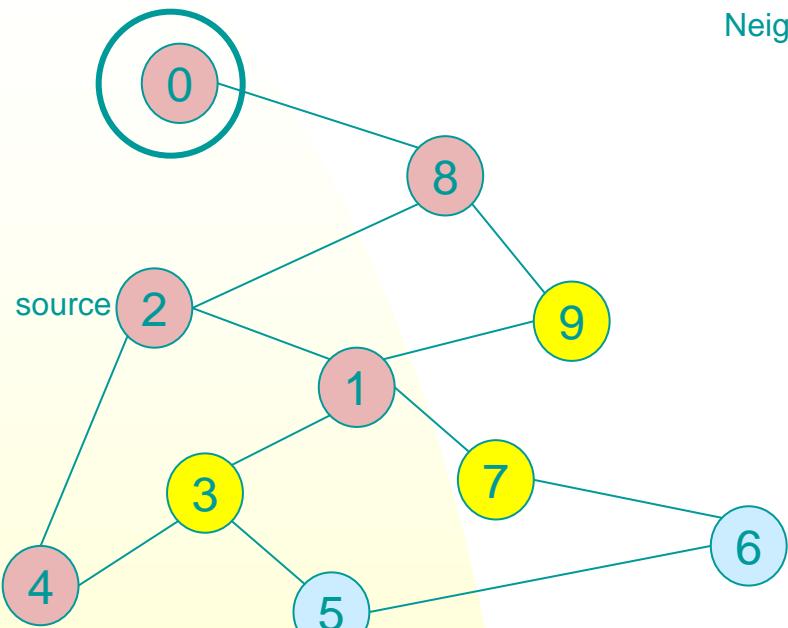
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Neighbors

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

# Example



Adjacency List

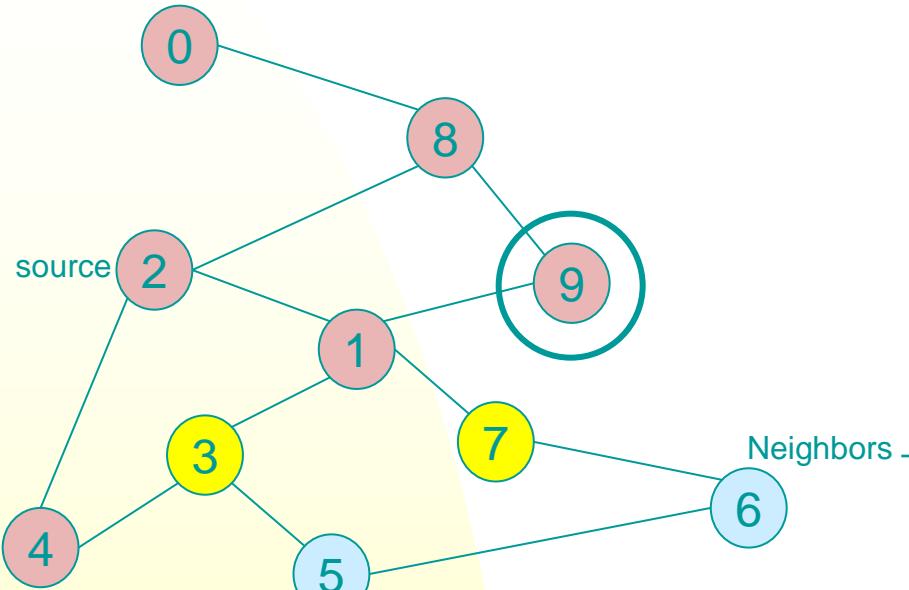
Neighbors

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

# Example



visit sequence = {2, 8, 1, 4, 0, 9}

$Q = \{ 9, 3, 7 \} \rightarrow \{ 3, 7 \}$

Dequeue 9.  
-- 9 has no unmarked neighbors!

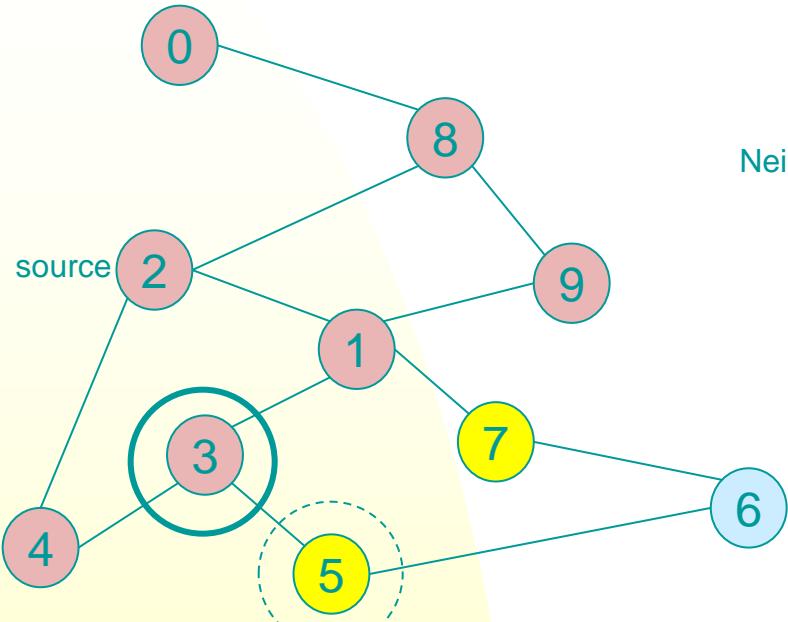
Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

# Example



Neighbors

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

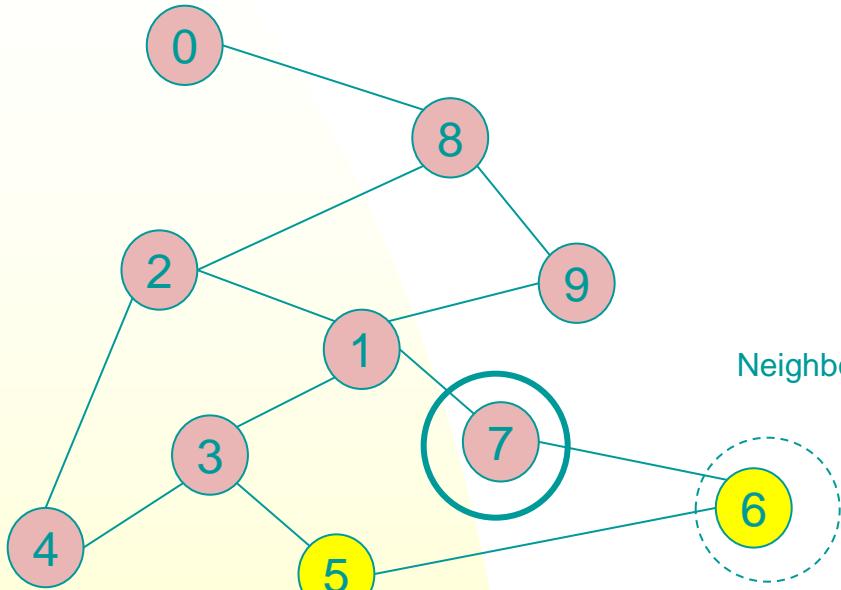
0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T

Mark unmarked neighbor.

$Q=\{ 3, 7 \} \rightarrow \{ 7, 5 \}$  Dequeue 3.

-- place neighbor 5 on the queue.

# Example



$$Q = \{ 7, 5 \} \rightarrow \{ 5, 6 \}$$

Dequeue 7.

-- place neighbor 6 on the queue.

Breadth-First Search

Adjacency List

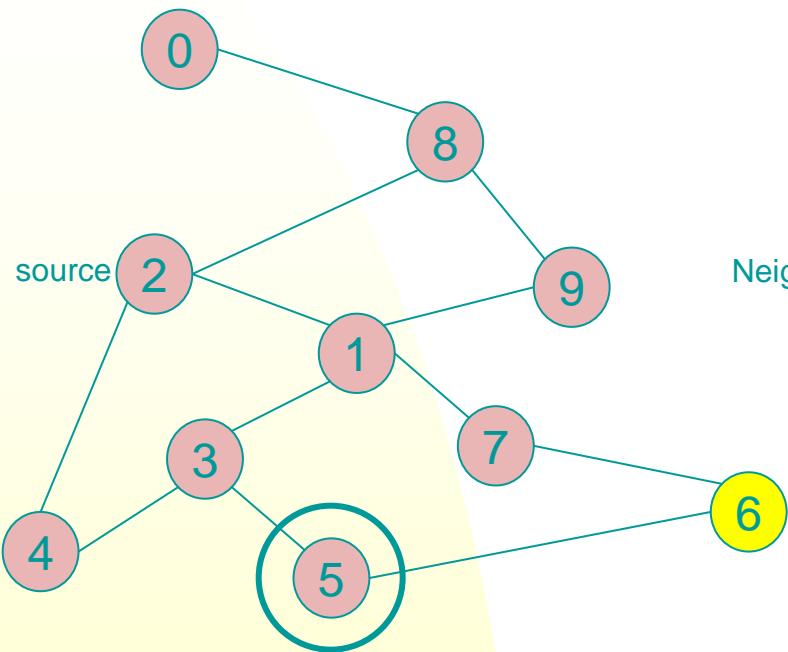
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Mark unmarked neighbor.

# Example



visit sequence = {2, 8, 1, 4, 0, 9, 3, 7, 5}

$$Q = \{ 5, 6 \} \rightarrow \{ 6 \}$$

Dequeue 5.

-- no unmarked neighbors of 5

Breadth-First Search

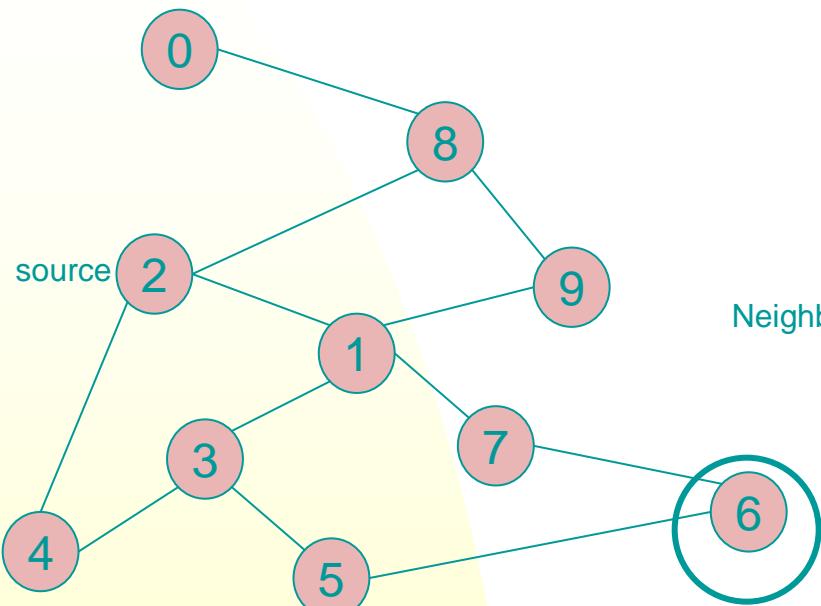
Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

# Example



visit sequence = {2, 8, 1, 4, 0, 9, 3, 7, 5, 6}

$Q = \{ 6 \} \rightarrow \{ \ }$

Dequeue 6.

-- no unmarked neighbors of 6.

Breadth-First Search

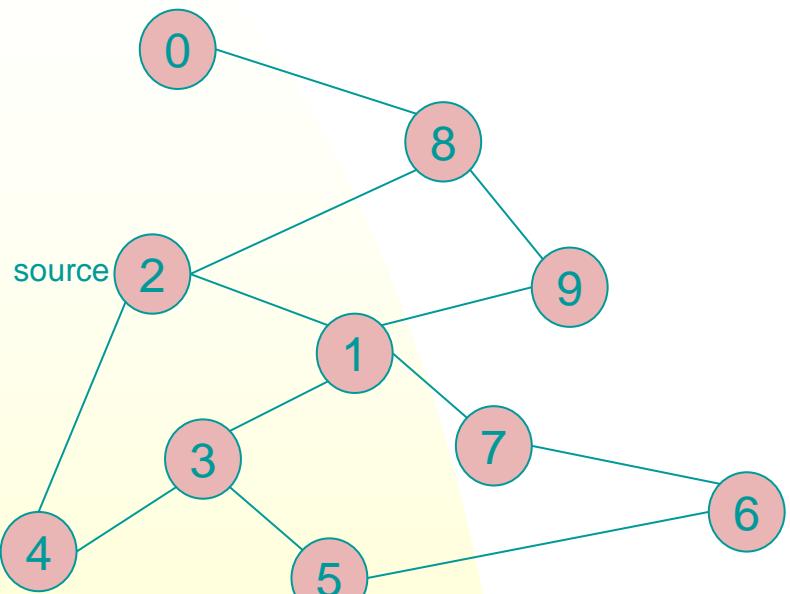
Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

# Example



visit sequence = {2, 8, 1, 4, 0, 9, 3, 7, 5, 6}

$Q = \{ \}$  Q is empty, exit the while loop.

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

There exist a path from source vertex 2 to all vertices in the graph!

# Remarks

- The unmarked neighbors enter to  $Q$  in the same order as in appear in the adjacent list.
  - ◆ It follows that if the order in the adjacent list is different, the output visit sequence will also be different.
- Starting at source  $s$ , BFS visits all the other (connected) vertices at increasing distance from  $s$ .

# Running Time

Assume the graph is represented by an **adjacency list**. Let  $n$  and  $m$  represent the number of vertices and edges respectively.

1. **for** each vertex  $v$
  2.     **do**  $\text{flag}[v] := \text{false}$ ; ← 

It loops  $O(n)$  times.
  3.  $Q = \text{empty queue}$ ;
  4.  $\text{flag}[s] := \text{true}$ ;
  5.  $\text{enqueue}(Q, s)$ ;
  6. **while**  $Q$  is not empty
  7.     **do**  $v := \text{dequeue}(Q)$ ;
  8.         **for** each  $w$  adjacent to  $v$
  9.             **do if**  $\text{flag}[w] = \text{false}$
  10.                 **then**  $\text{flag}[w] := \text{true}$ ;
  11.                  $\text{enqueue}(Q, w)$
- For a particular  $v$ , the **for**-loop loops exactly  $O(\text{degree}(v))$  times (which is the size of that linked-list).
- For a particular  $v$ , it loops at most  $O(\text{degree}(v))$  times (which is the number of neighbors).

# Running time

- Observe that whenever a vertex is marked for the first time, it is put inside  $Q$  in line 11. A marked vertex in  $Q$  will eventually be dequeued in line 7 and it will never be put inside  $Q$  again.
  - ◆ a vertex can only be dequeued (enqueued) one time
- Whenever a vertex  $v$  is dequeued,
  - ◆ we first find out all neighbors of  $v$ . For adjacency list representation, it needs to access the whole linked-list which has size  $O(\deg(v))$ .
  - ◆ It follows the **total time** needed for all vertex is :

$$\sum_{\text{vertex } v} O(\deg(v)) = O(2m) = O(m)$$

# Running time

- Moreover,
  - ◆ the neighbors ( $w$ ) may be enqueued. For one vertex  $v$ , then it may has  $O(\deg(v))$  operations. However, since every vertex is enqueued (dequeued) exactly once, it follows the total number of enqueued (dequeue) operations is

$$O(n)$$

- Hence the running time for BFS for adjacency list representation is

$$O(n) + O(n) + O(2m) = O(n+m)$$

initialization

enqueued/dequeued operations

find out all the neighbors

# Running time

If the graph is represented by an adjacent matrix, the analysis is the same, except:

- To find out all neighbors of  $v$ , for adjacency matrix representation, it needs to access a row in the matrix, which has size  $O(n)$ .

It follows the total time needed for all vertex is :

$$\sum_{\text{vertex } v} O(n) = O(n^2)$$

- Hence the total running time for BFS is

$$O(n) + O(n) + O(n^2) = O(n^2)$$

initialization

enqueued/dequeued operation

find out all the neighbors

Breadth-First Search

# Path recording

- BFS only tells us if a path exists from source  $s$  to other vertices  $v$ .
  - ◆ It doesn't tell us the path!
  - ◆ We need to modify the algorithm to record the (shortest) path from  $s$  to  $v$ .
- The trick is to keep one additional piece of information with each vertex.

# Path recording

- Let  $\text{pred}[0..n-1]$  be an array indexed by the vertices. The entry  $\text{pred}[w]$  contains the vertex  $v$  from where  $w$  is discovered, i.e.,  $w$  was put inside the  $Q$  in line 11 because  $w$  is discovered by  $v$ .

```
6.  while  $Q$  is not empty
7.      do  $v := \text{dequeue}(Q)$ ;
8.          for each  $w$  adjacent to  $v$ 
9.              do if  $\text{flag}[w] = \text{false}$ 
10.                 then  $\text{flag}[w] := \text{true}$ ;
11.                     $\text{enqueue}(Q, w)$ 
```

$w$  is ‘*discovered*’ by  $v$ ,  
hence the path from  $s$  to  $w$   
must pass through  $v$ , i.e.,  
 $s \rightarrow \dots \rightarrow v \rightarrow w$

# BFS and Path recording

**Algorithm**  $BFS(s)$

1. **for** each vertex  $v$
2.     **do**  $flag(v) := \text{false};$
3.          $pred[v] := -1;$  Initialization
4.      $Q = \text{empty queue};$
5.      $flag[s] := \text{true};$
6.      $\text{enqueue}(Q, s);$
7.     **while**  $Q$  is not empty
8.         **do**  $v := \text{dequeue}(Q);$
9.         **for** each  $w$  adjacent to  $v$
10.             **do if**  $flag[w] = \text{false}$
11.                 **then**  $flag[w] := \text{true};$
12.                  $pred[w] := v;$  prev[w] stores which vertex discovers w.
13.                  $\text{enqueue}(Q, w)$

# Path Reporting

- After running the modified BFS, if  $\text{flag}[w] = \text{true}$  (it means there exists a path from  $s$  to  $w$ ), one can call  $\text{Path}(w)$  to output the vertices on the path from  $s$  to  $w$  **in this order**.

**Algorithm**  $\text{Path}(w)$

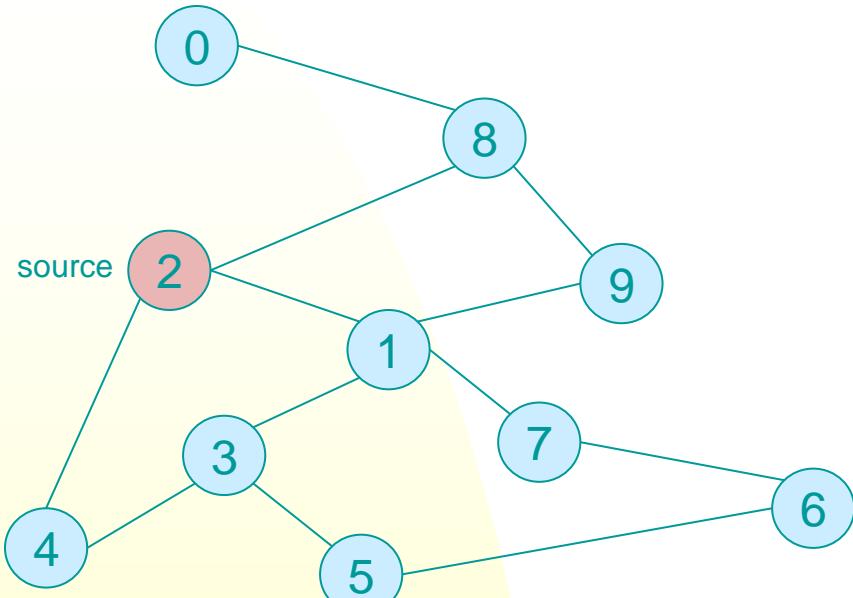
1. **if**  $\text{pred}[w] \neq -1$
2. **then**
3.      $\text{Path}(\text{pred}[w]);$
4. **output**  $w$

Notice the recursive structure which outputs a shortest path from  $s$  to  $w$  (not from  $w$  to  $s$ ).

# Shortest Path Reporting

- The running time is proportional to the length of the path from  $s$  to  $w$ .
- The path returned is actually the **shortest** from  $s$  to  $w$ . That is, among all possible paths from  $s$  to  $w$ , it has the minimum number of edges.

# Example



$$Q = \{ \quad \}$$

Initialize **Q** to be empty

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

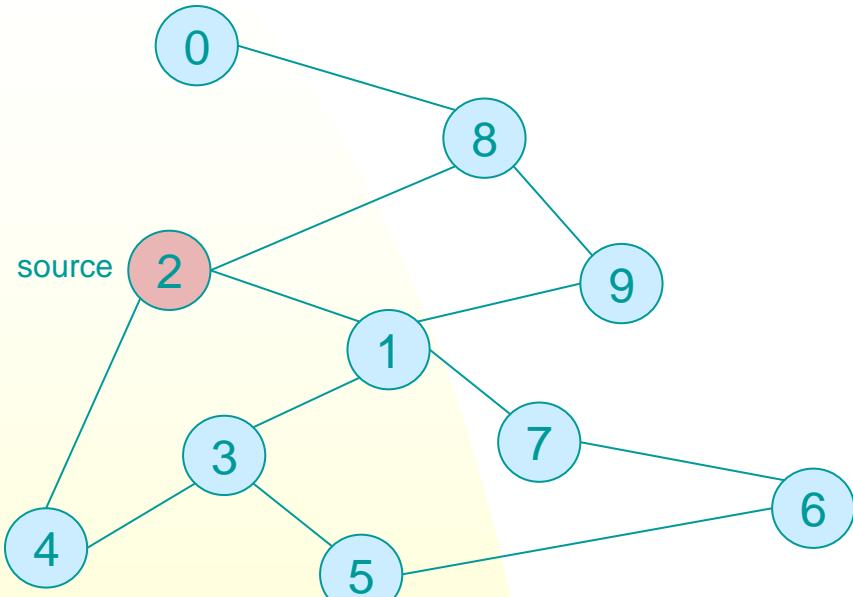
0	F	-1
1	F	-1
2	F	-1
3	F	-1
4	F	-1
5	F	-1
6	F	-1
7	F	-1
8	F	-1
9	F	-1

Pred

Initialize flag table (all F)

Initialize Pred to -1

# Example



$$Q = \{ 2 \}$$

Place source 2 on the queue.

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

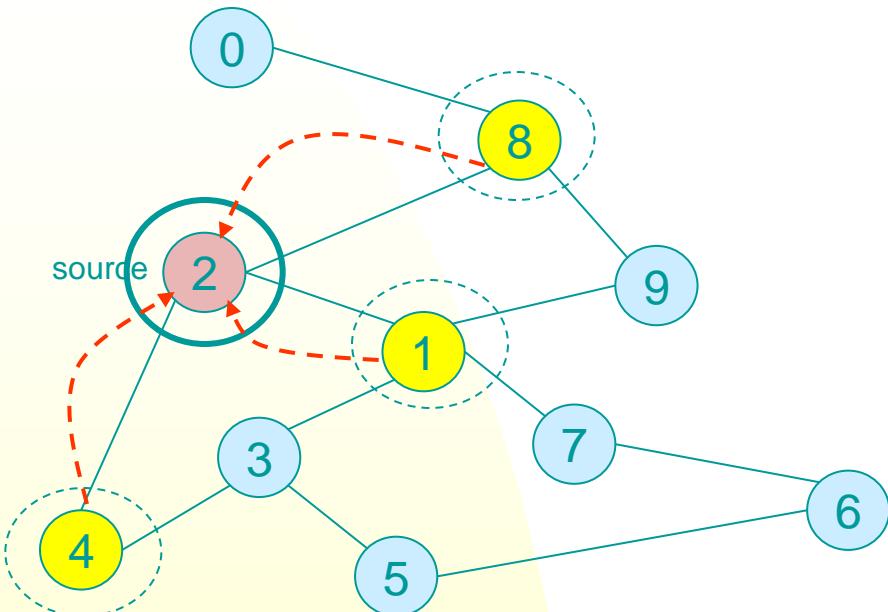
0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

-1
-1
-1
-1
-1
-1
-1
-1
-1
-1

Pred

Flag that 2 has been marked.

# Example



$$Q = \{2\} \rightarrow \{ 8, 1, 4 \}$$

Dequeue 2.  
Place all unmarked neighbors of 2 on the queue

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	F
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	F

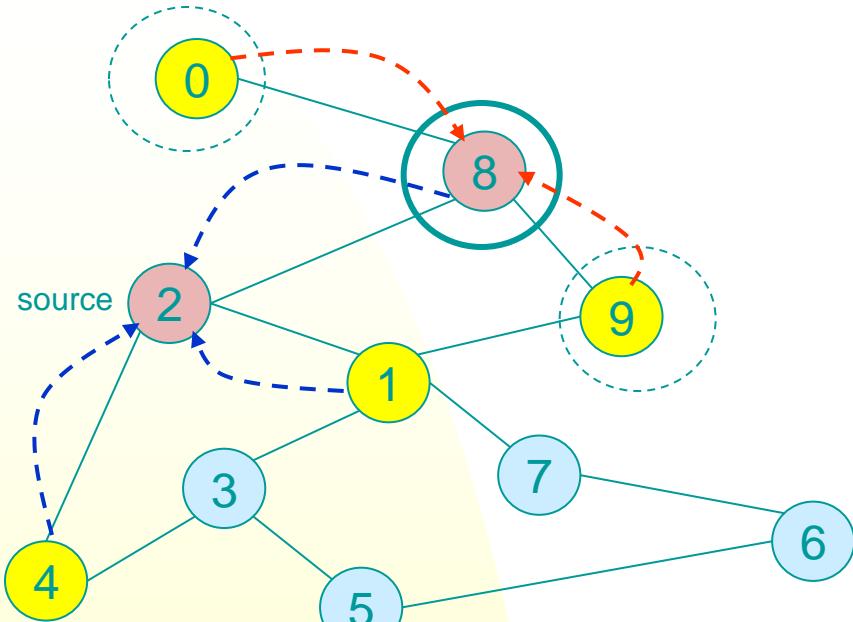
  

-1
2
-1
-1
2
-1
-1
-1
2
-1

Pred

Record in Pred  
who was marked  
(discovered)  
by 2.

# Example



Dequeue 8.

-- Place all unmarked neighbors of 8 on the queue.

-- Notice that 2 is not placed on the queue again, it has been visited!

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

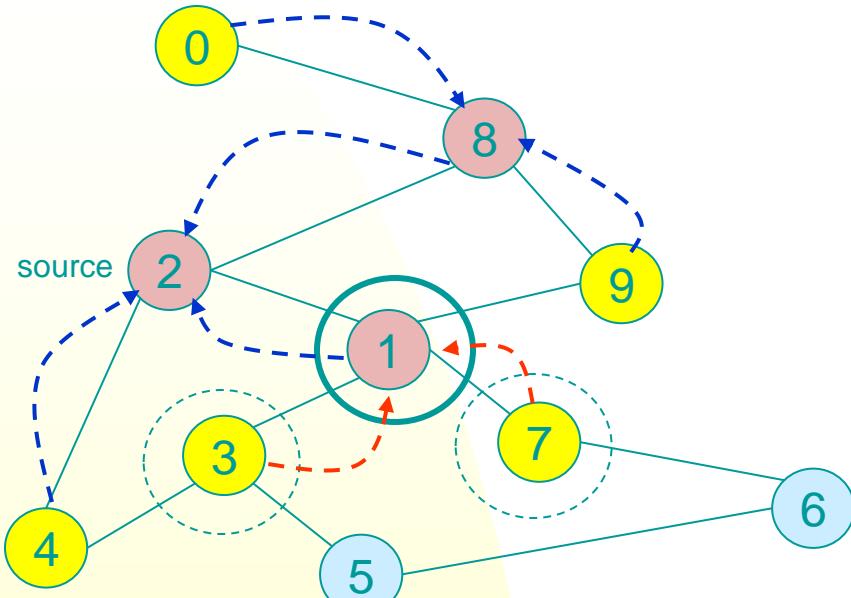
0	T	8
1	T	2
2	T	-1
3	F	-1
4	T	2
5	F	-1
6	F	-1
7	F	-1
8	T	2
9	T	8

Pred

Mark unmarked  
Neighbors.

Record in Pred  
who was marked  
by 8.

# Example



Dequeue 1.

- Place all unmarked neighbors of 1 on the queue.
- Only nodes 3 and 7 haven't been marked yet.

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

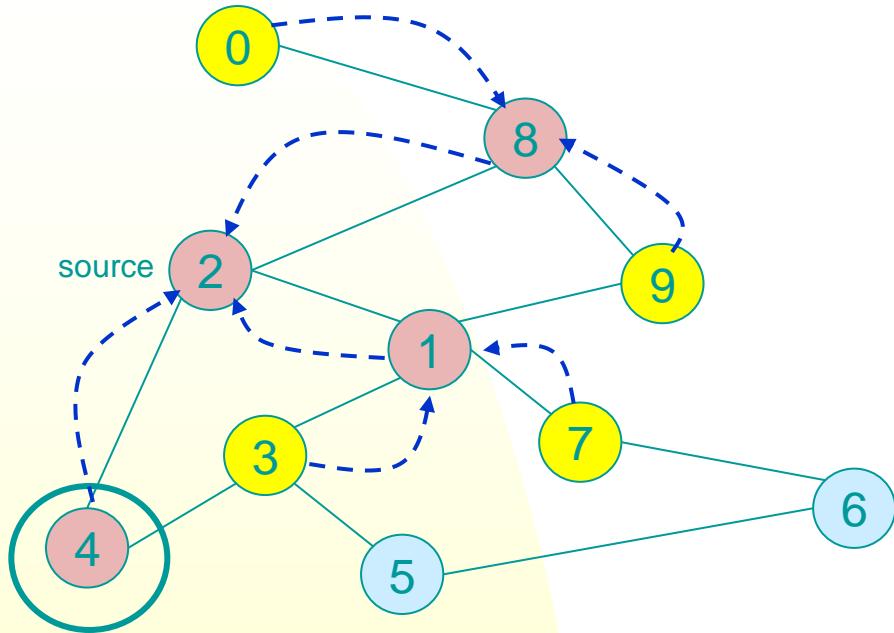
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	F	-1
6	F	-1
7	T	1
8	T	2
9	T	8

Pred

Mark unmarked  
Neighbors.

Record in Pred  
who was marked  
by 1.

# Example



$$Q = \{ 4, 0, 9, 3, 7 \} \rightarrow \{ 0, 9, 3, 7 \}$$

Dequeue 4.  
-- 4 has no unmarked neighbors!

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

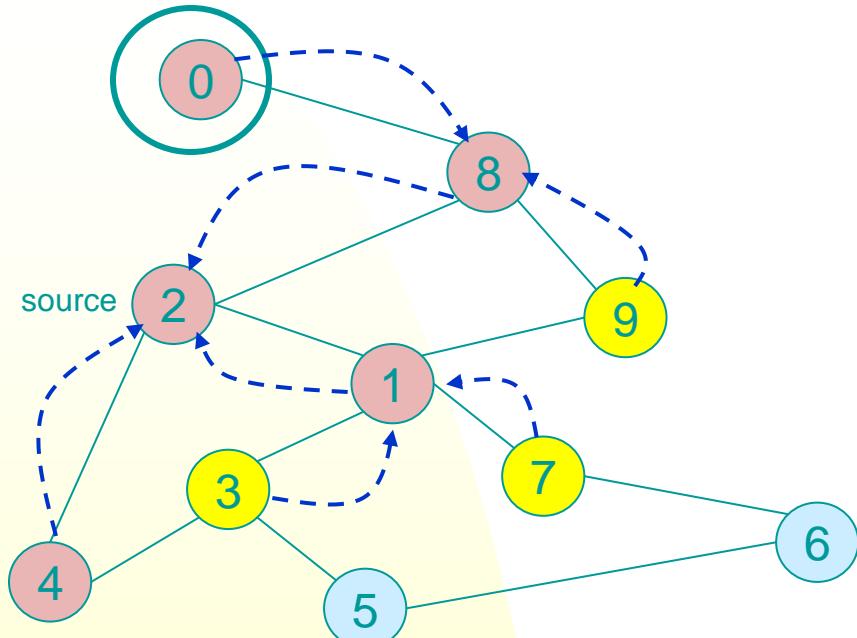
Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Pred

8
2
-1
1
2
-1
-1
1
2
8

# Example



$$Q = \{0, 9, 3, 7\} \rightarrow \{9, 3, 7\}$$

Dequeue 0.

-- 0 has no unmarked neighbors!

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

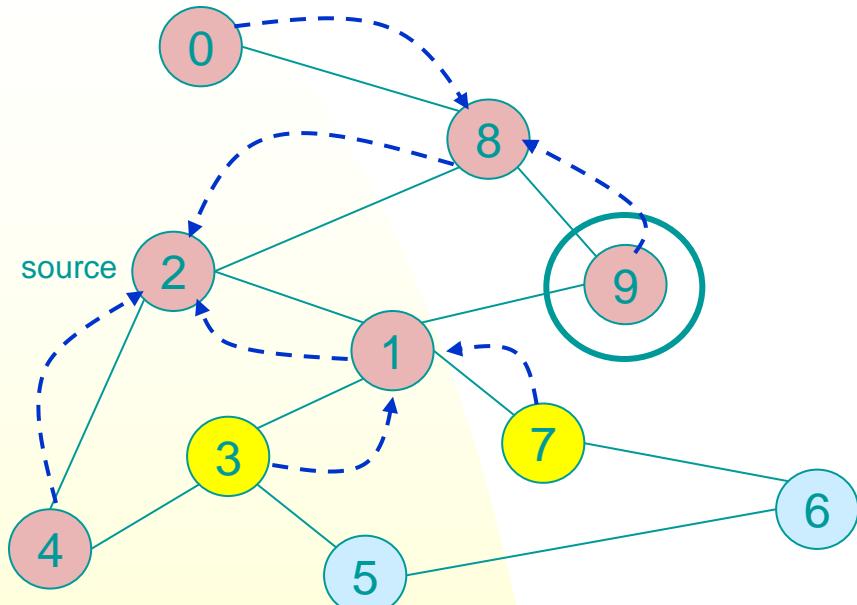
Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Pred

8
2
-1
1
2
-1
1
2
8

# Example



$$Q = \{ 9, 3, 7 \} \rightarrow \{ 3, 7 \}$$

Dequeue 9.  
-- 9 has no unmarked neighbors!

Adjacency List

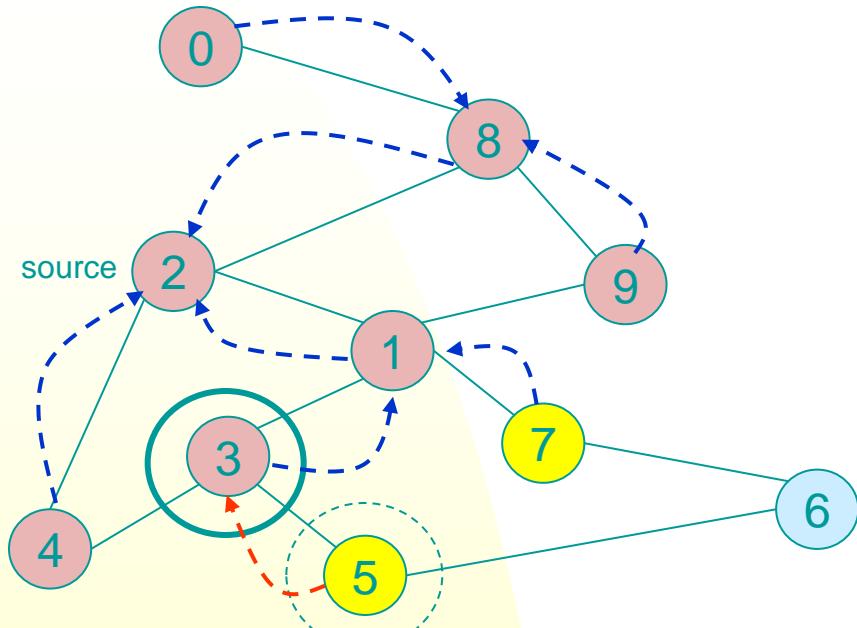
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Pred

# Example



$$Q = \{3, 7\} \rightarrow \{7, 5\}$$

Dequeue 3.  
-- place neighbor 5 on the queue.

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T

8
2
-1
1
2
3
-
1
2
8

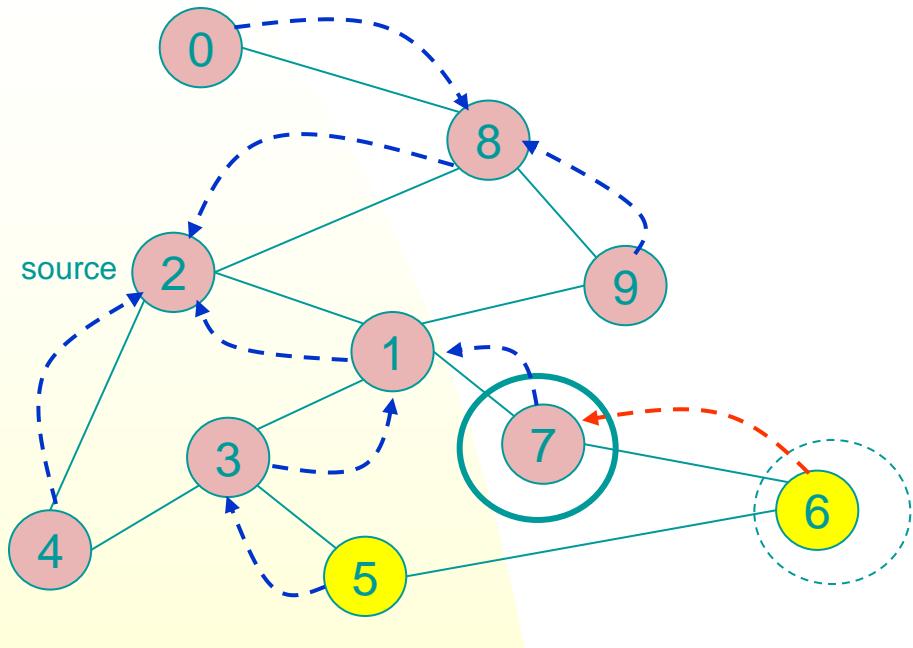
Pred

Mark unmarked  
Vertex 5.

Breadth-First Search

Record in Pred  
who was marked  
by 3.

# Example



$$Q = \{7, 5\} \rightarrow \{5, 6\}$$

Dequeue 7.  
-- place neighbor 6 on the queue.

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

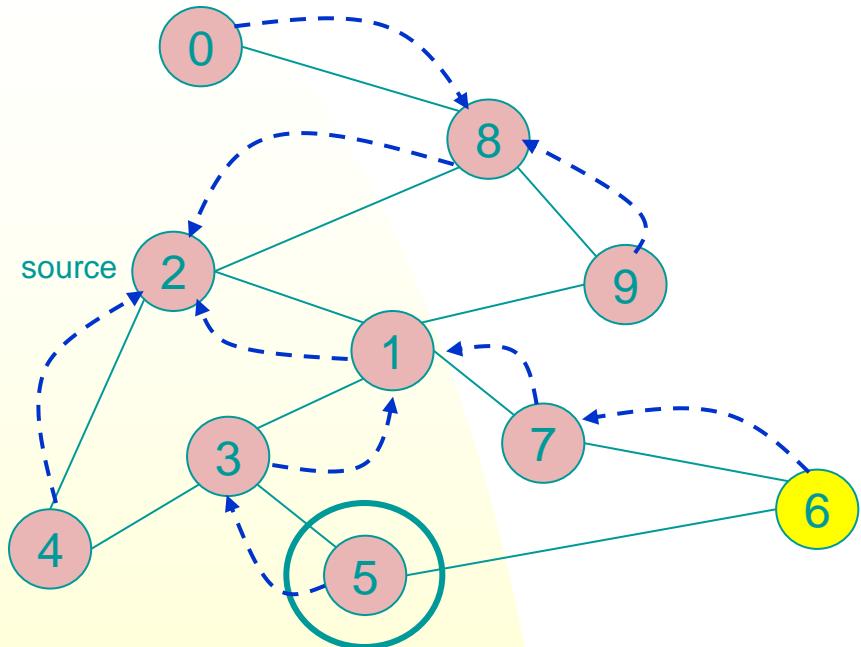
Pred

Mark unmarked  
Vertex 6.

Breadth-First Search

Record in Pred  
who was marked  
by 7.

# Example



$$Q = \{5, 6\} \rightarrow \{6\}$$

Dequeue 5.  
-- no unmarked neighbors of 5.

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

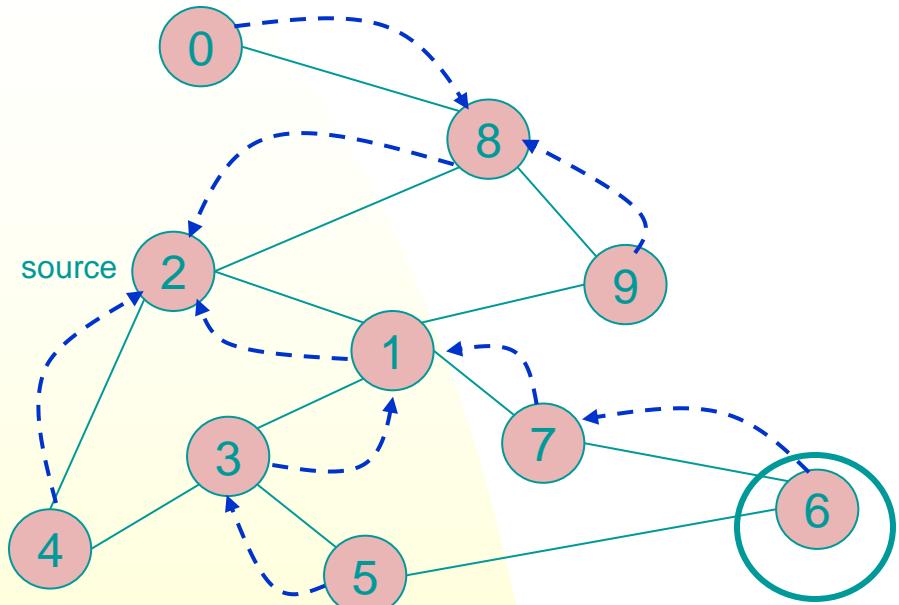
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

8
2
-1
1
2
3
7
1
2
8

Breadth-First Search

# Example



$$Q = \{ 6 \} \rightarrow \{ \}$$

Dequeue 6.

-- no unmarked neighbors of 6.

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

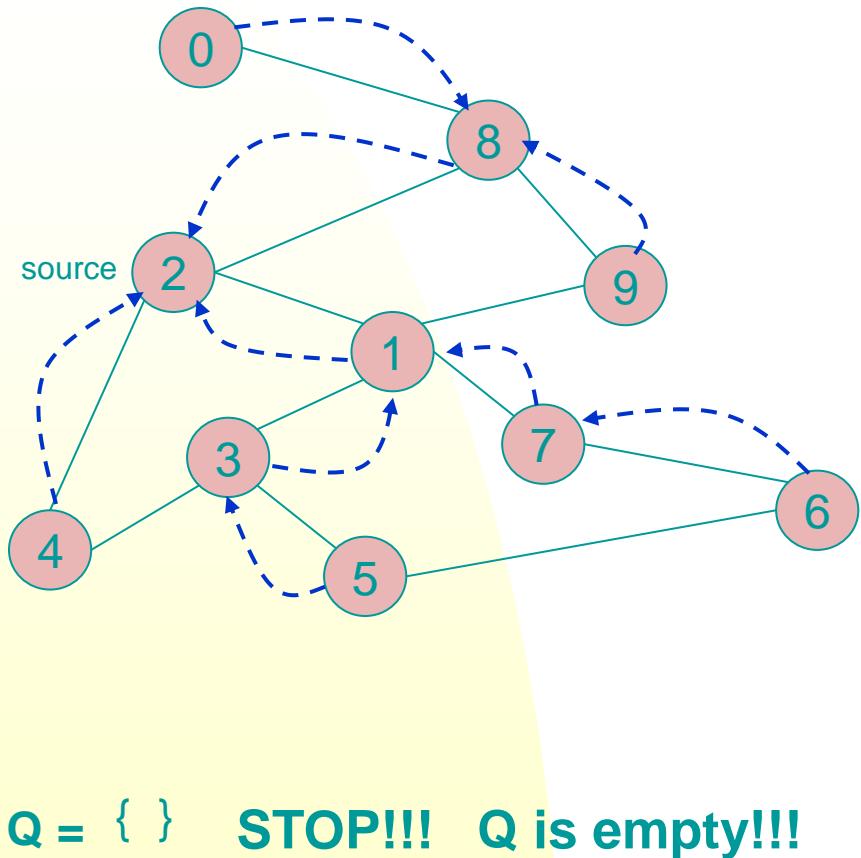
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

8
2
-1
1
2
3
7
1
2
8

Breadth-First Search

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

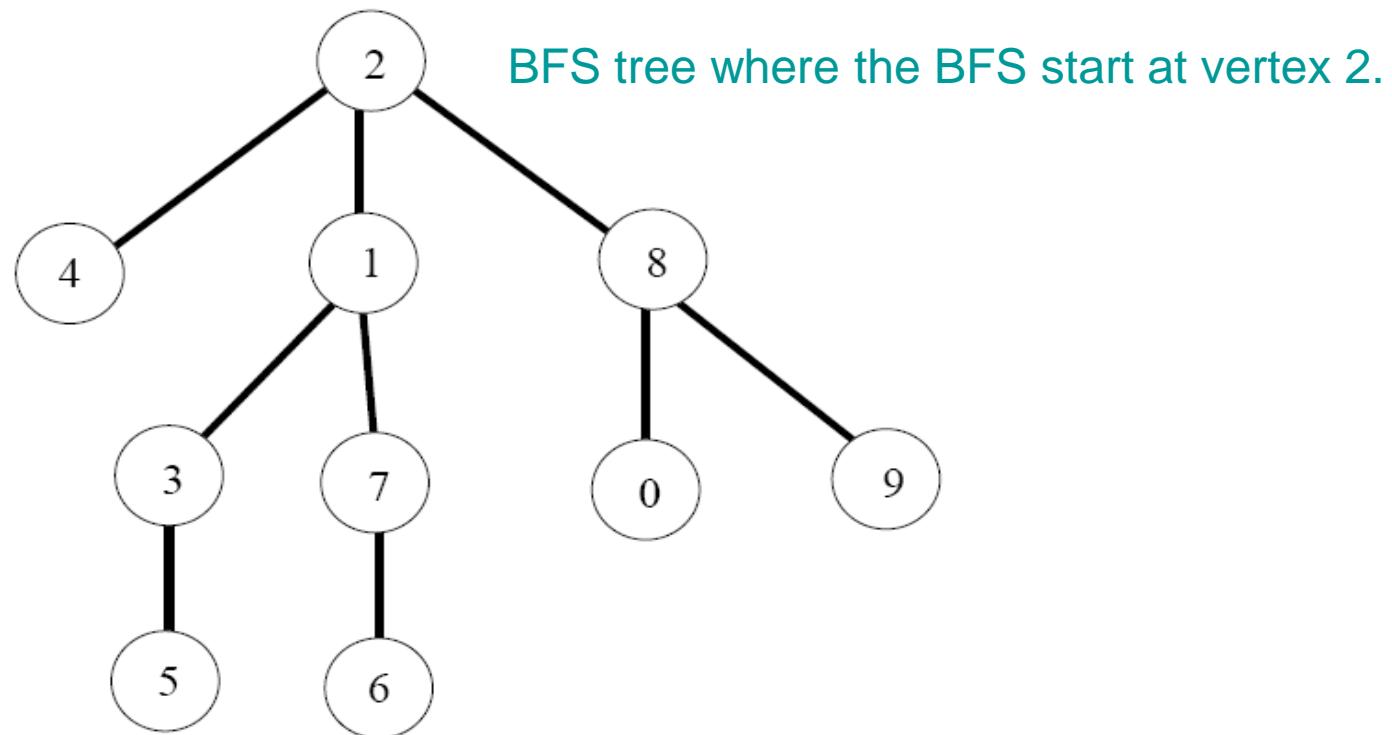
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Pred

Pred now stores all the paths!

# BFS tree

- We often draw the BFS paths as a tree, where  $s$  is the root.



The root ( $s$ ) to  $v$  path in the BFS tree represents the shortest from  $s$  to  $v$  in the original graph, and the level of  $v$  represents the length of such shortest path.

# DFS : Depth-First Search

---

- DFS is another popular search strategy.
- It can do certain things that BFS cannot do. We will discuss some of these algorithms in COMP 271 (so you cannot get rid of DFS after COMP171).
- DFS idea :
  - ◆ Whenever we visit a vertex  $v$  from another vertex  $u$ , we **recursively visit a neighbor of  $v$**  that has not been visited before until all neighbors of  $v$  have been visited. Then we **backtrack** (return) to  $u$ .

# Algorithm

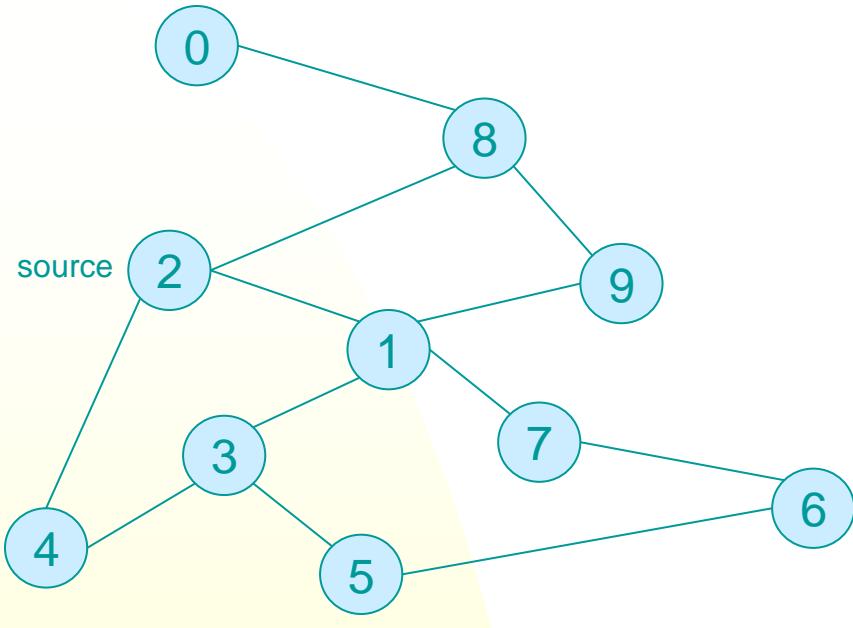
## Algorithm $DFS(s)$

1. **for** each vertex  $v$
2.     **do**  $flag[v] := \text{false};$  ← Flag all vertices as not visited
3.      $RDFS(s);$

## Algorithm $RDFS(v)$

1.  $flag[v] := \text{true};$  ← Visit  $v$ , and mark  $v$  as visited.
2. **for** each neighbor  $w$  of  $v$
3.     **do if**  $flag[w] = \text{false}$      For each unvisited neighbor.  
        **then**  $RDFS(w);$      make a recursive call  $RDFS(w).$
- 4.

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

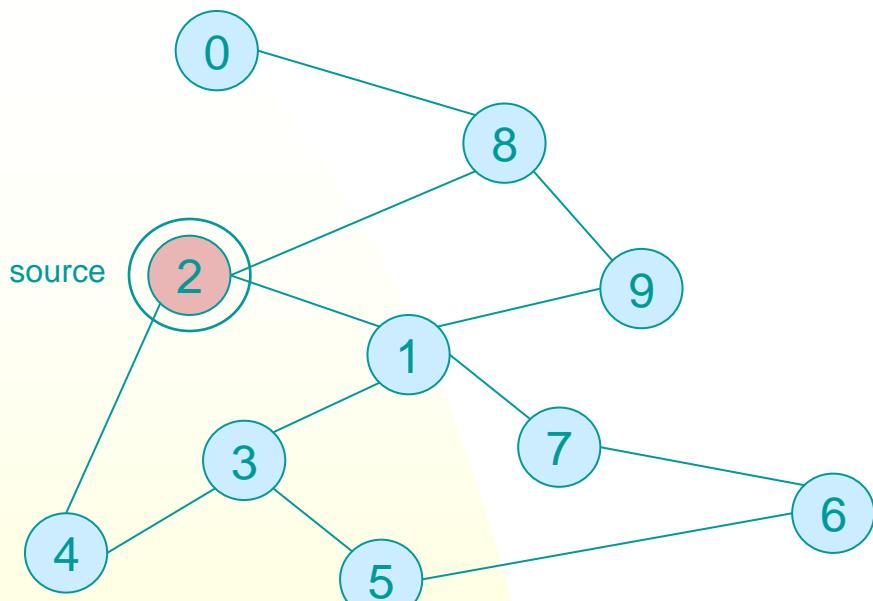
0	F	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

Pred

Initialize visited  
table (all empty F)

Initialize Pred to -1

# Example



RDFS( 2 )

recursive call → RDFS(8)

Adjacency List

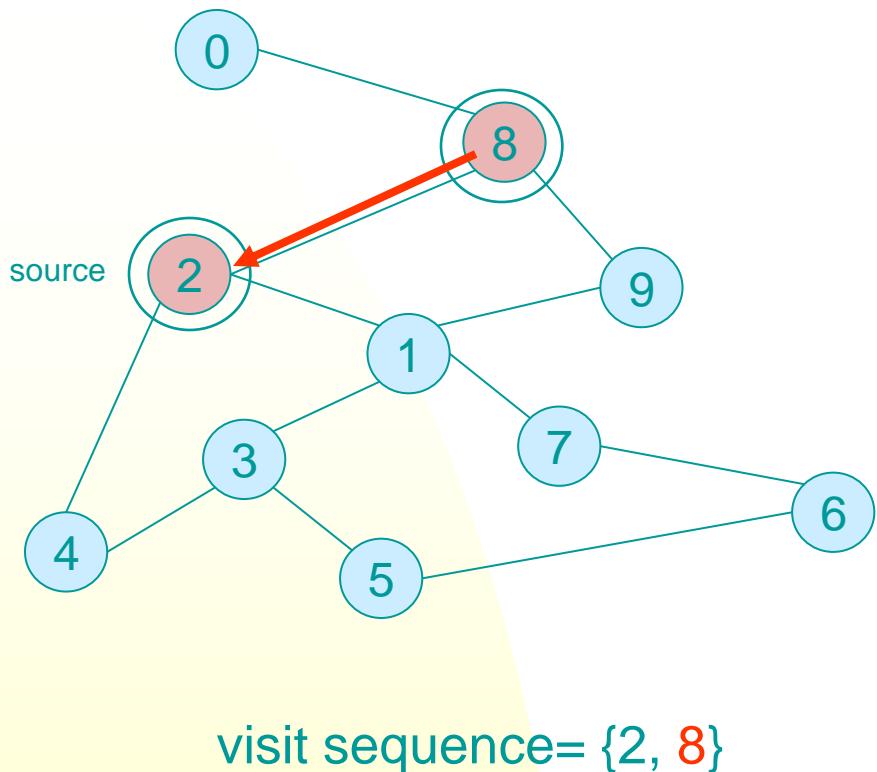
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

	Pred
0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Mark 2 as visited

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

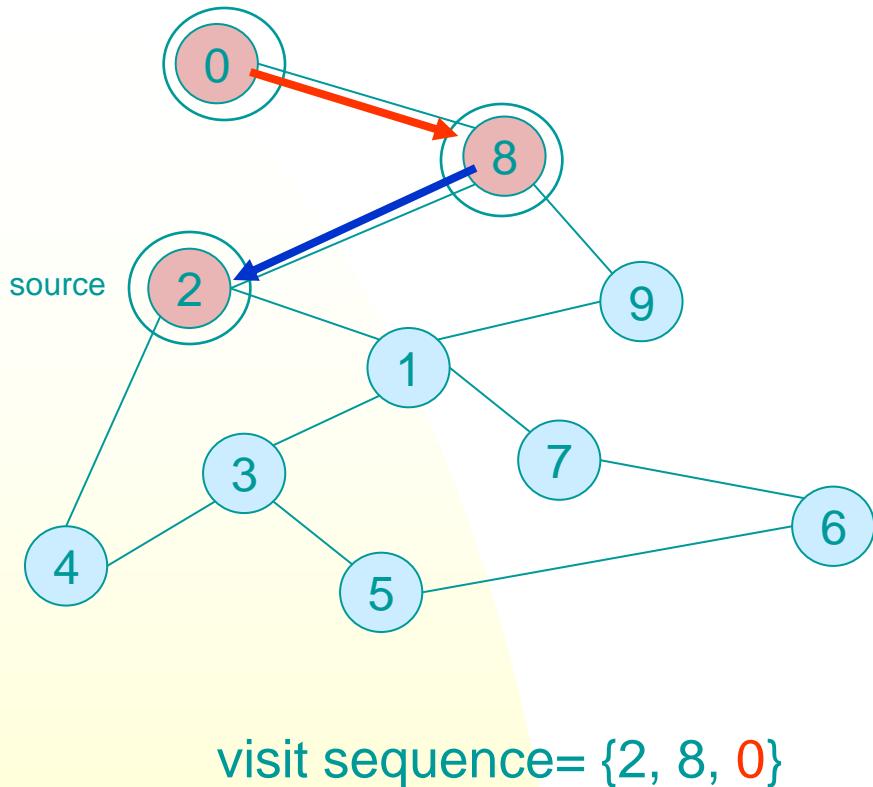
Visited Table (T/F)

	Pred
0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	F

Pred

Mark 8 as visited

# Example



Recursive calls

RDFS( 2 )

RDFS(8)

RDFS(**0**) -> no unvisited neighbor, return to (backtrack) RDFS(8)

Depth-First Search

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

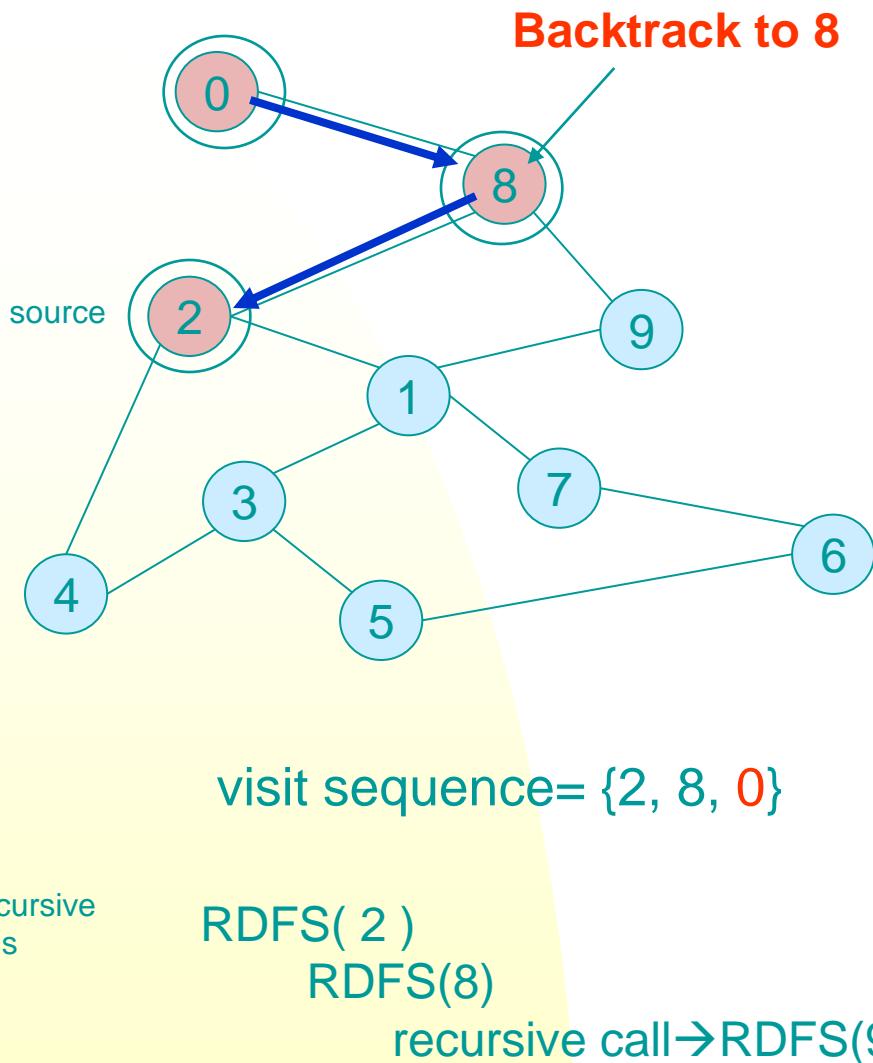
Visited Table (T/F)

	Pred
0	T
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	F

Pred

Mark 0 as visited

# Example



Adjacency List

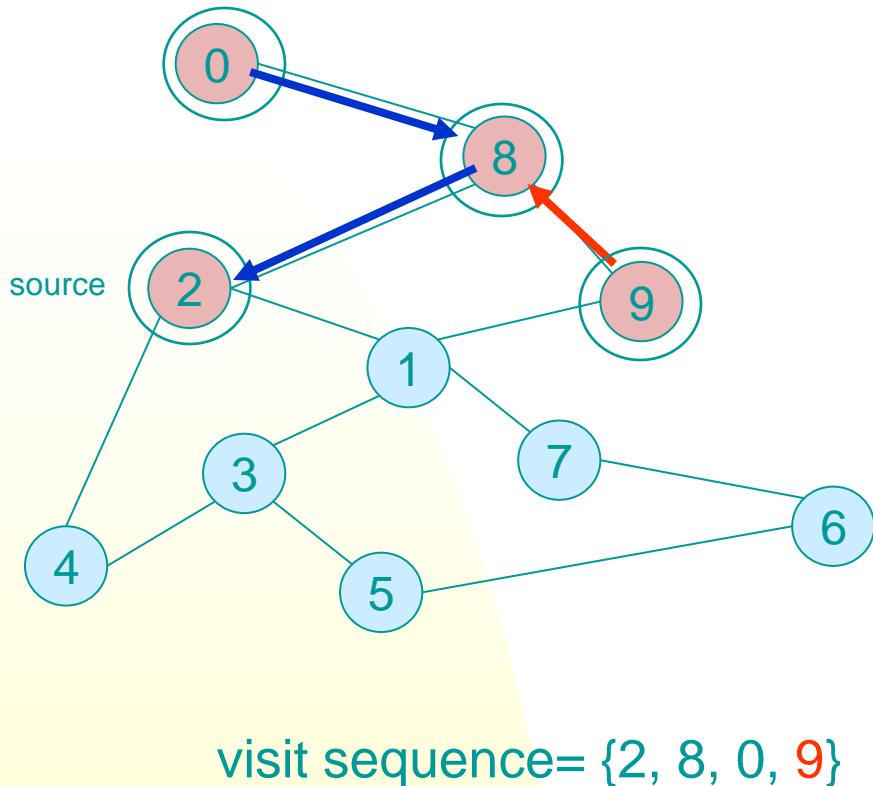
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	F

Pred

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

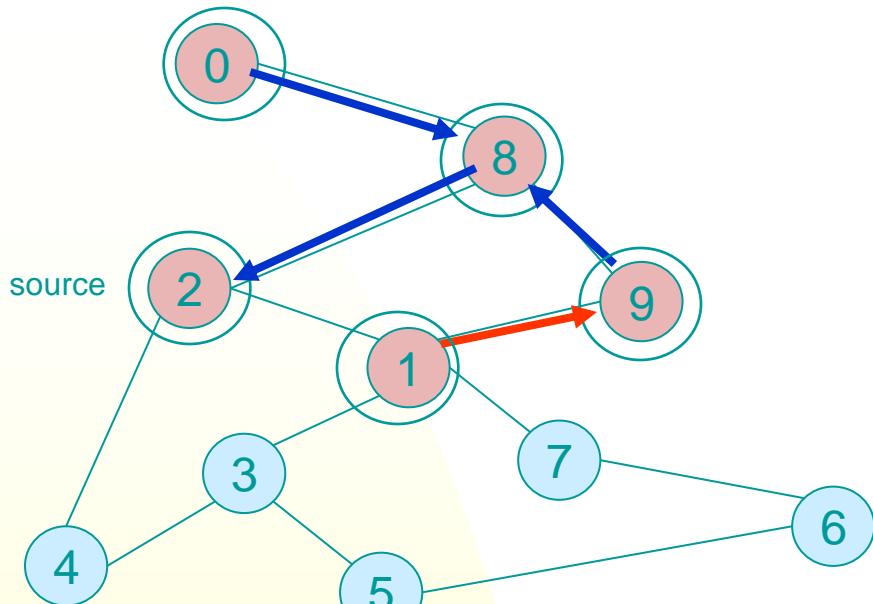
0	T
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	T

Pred

Mark 9 as visited

Depth-First Search

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

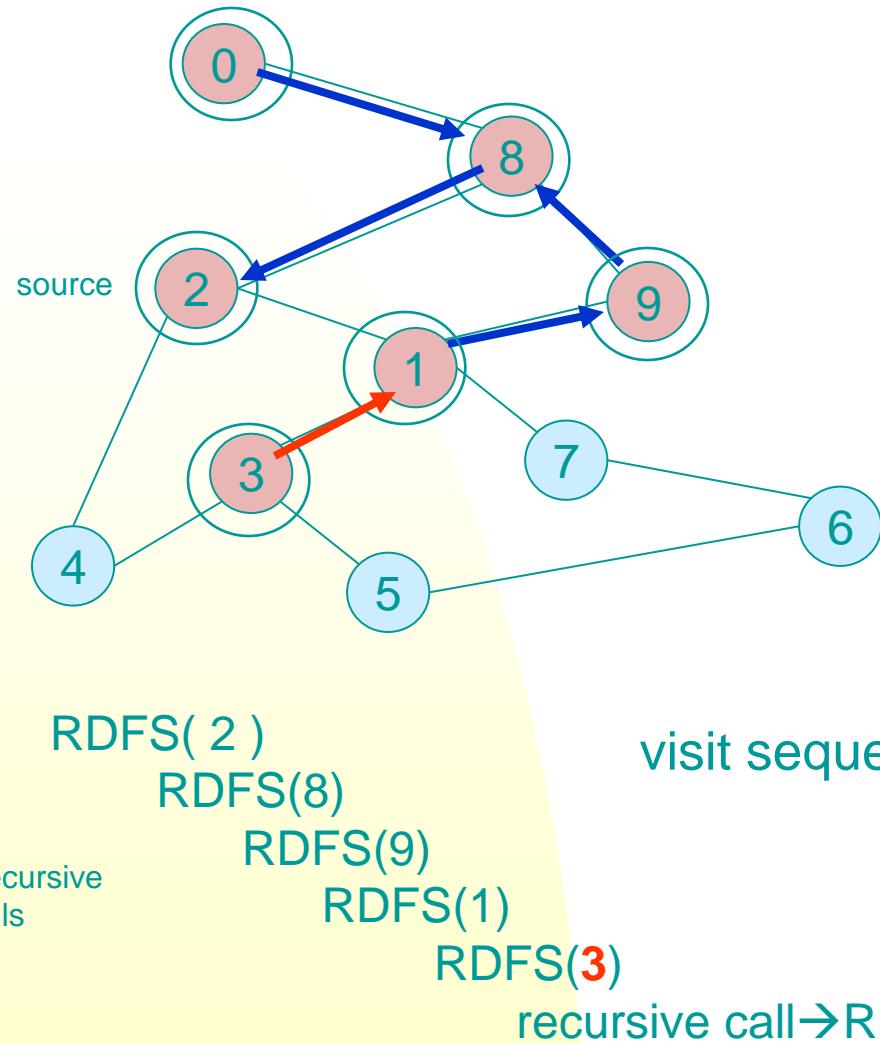
0	T
1	T
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	T

Pred

Mark 1 as visited

Depth-First Search

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

visit sequence= {2, 8, 0, 9, 1, 3}

Depth-First Search

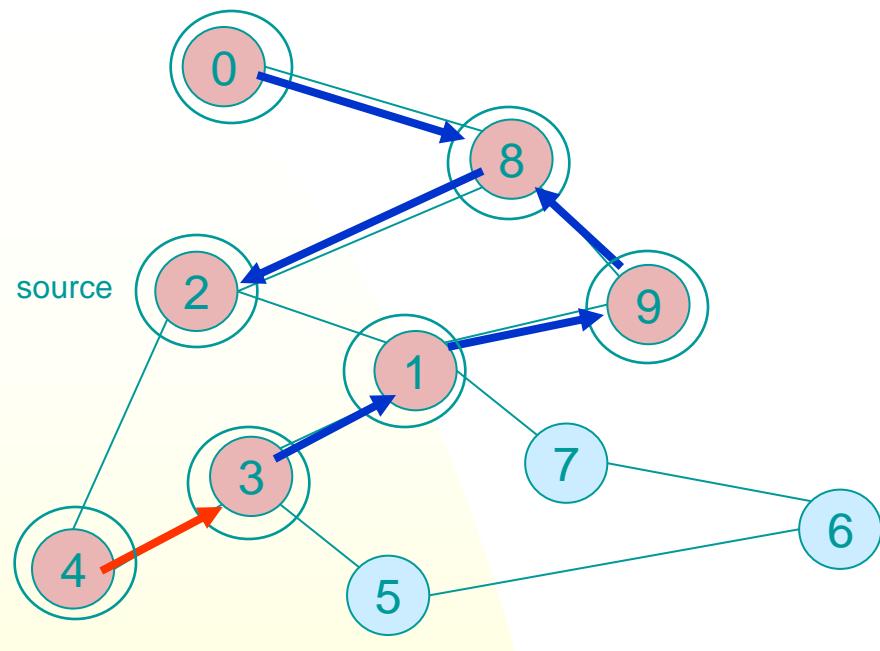
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	F
5	F
6	F
7	F
8	T
9	T

Pred

Mark 3 as visited

# Example



visit sequence= {2, 8, 0, 9, 1, 3, 4}

Recursive calls

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

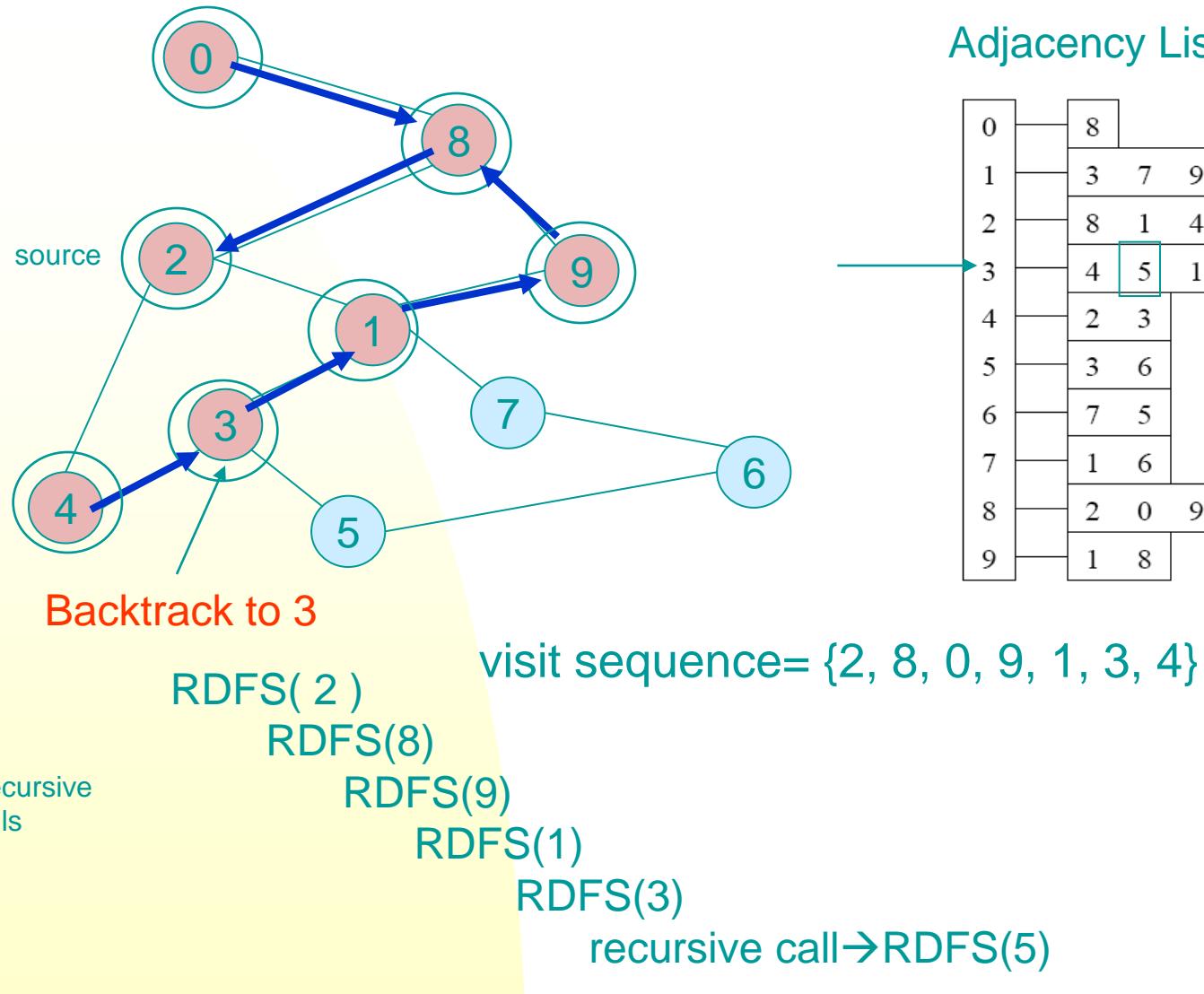
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	F
8	T
9	T

Pred

Mark 4 as visited

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

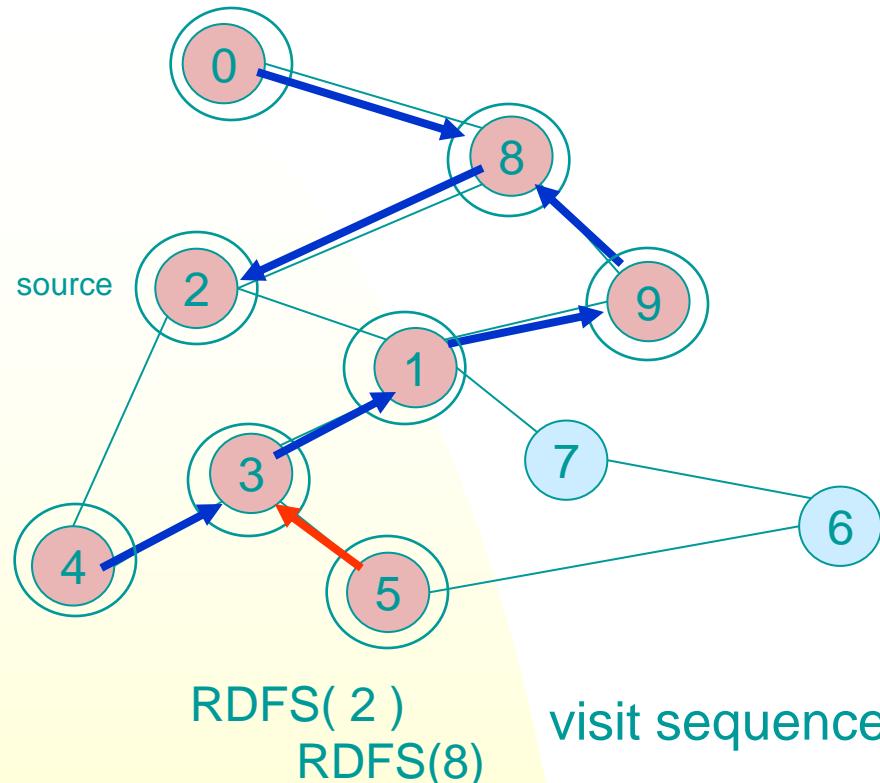
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	F
8	T
9	T

Pred

Depth-First Search

# Example



Recursive calls

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

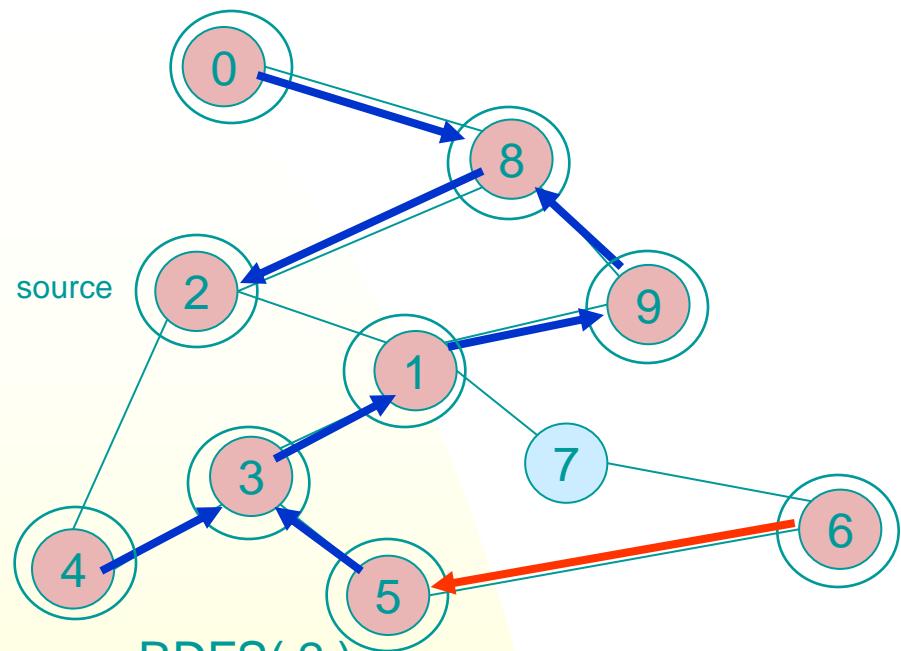
0	T
1	T
2	T
3	T
4	T
5	<b>T</b>
6	F
7	F
8	T
9	T

Pred

Mark 5 as visited

Depth-First Search

# Example



RDFS(2)

RDFS(8)

RDFS(9)

RDFS(1)

RDFS(3)

RDFS(5)

RDFS(6)

recursive call → RDFS(7)

Recursive calls

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

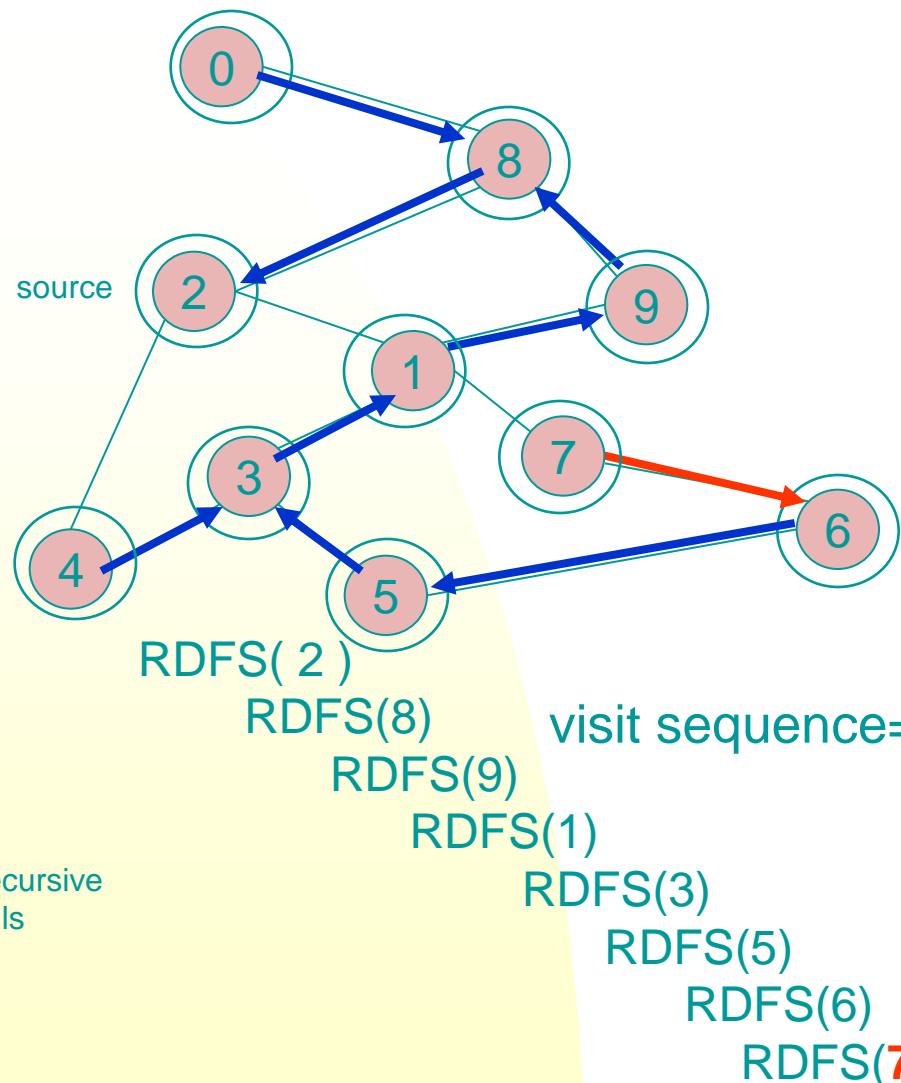
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	F
8	T
9	T

Pred

Mark 6 as visited

Depth-First Search

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

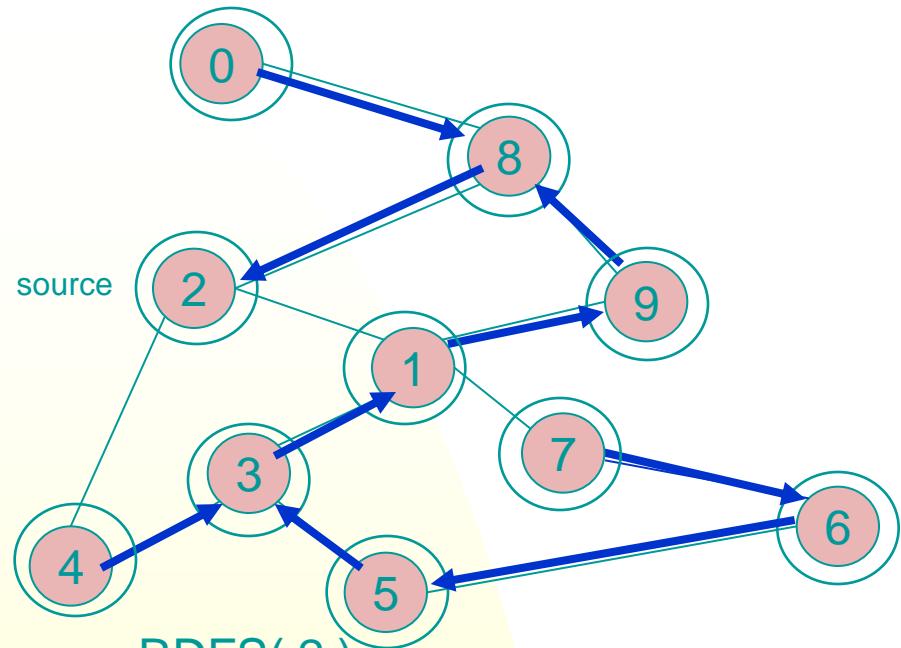
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

Mark 7 as visited

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

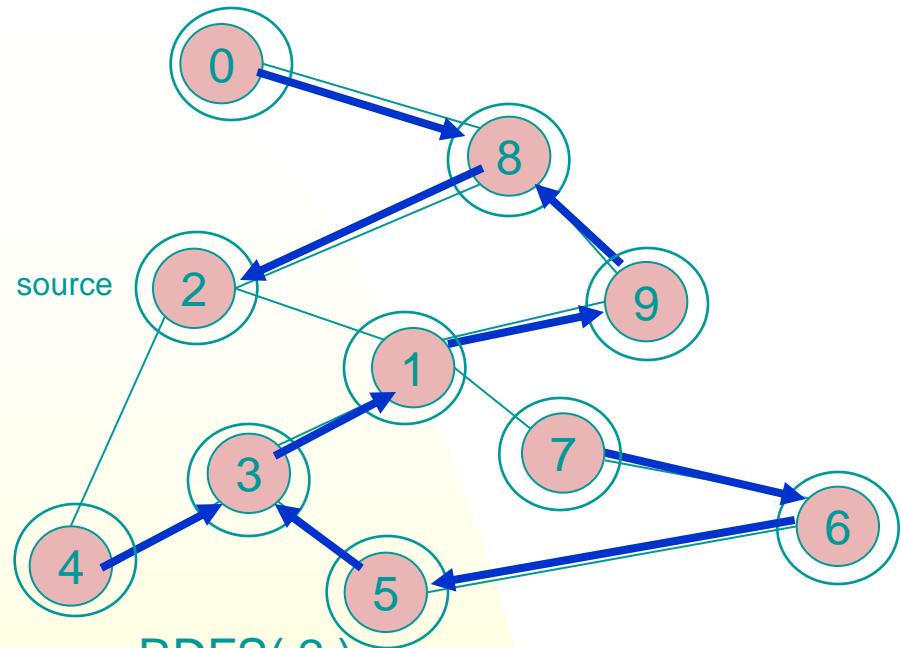
visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

# Example



visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Recursive calls

RDFS(2)  
RDFS(8)  
RDFS(9)  
RDFS(1)  
RDFS(3)  
RDFS(5)  
RDFS(6) → no recursive call

Adjacency List

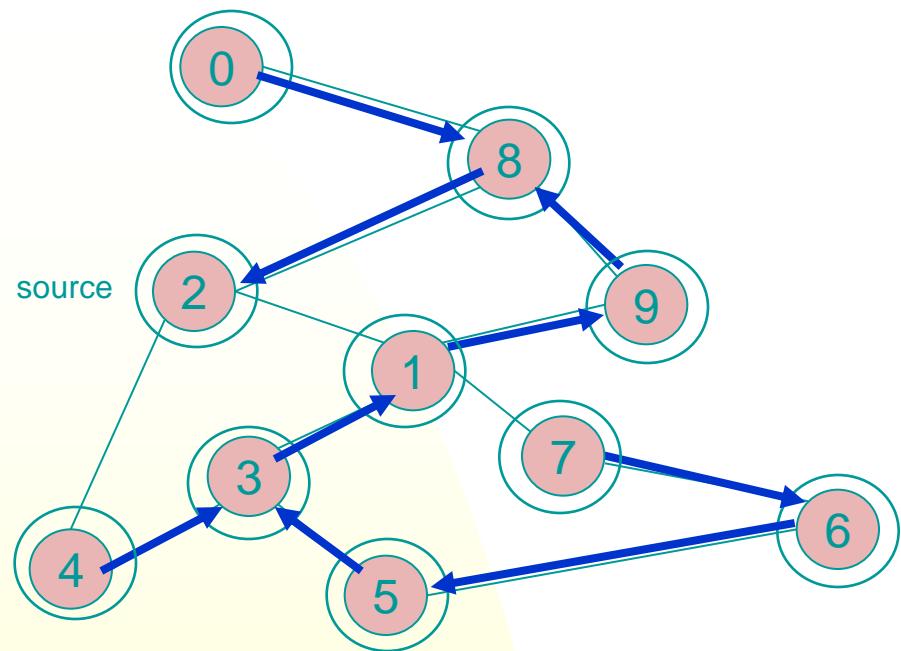
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

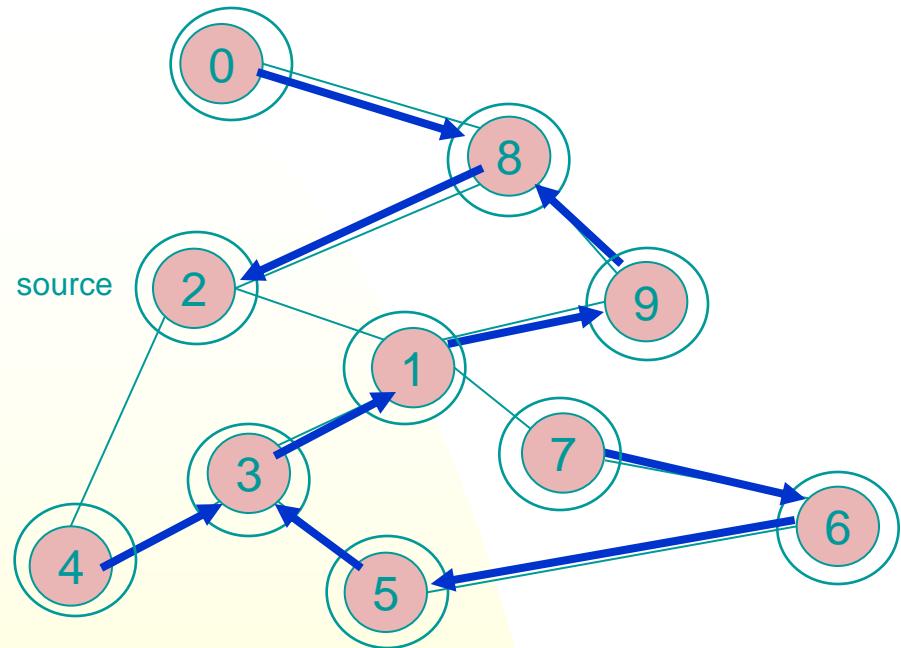
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

# Example



RDFS(2)

RDFS(8)

RDFS(9)

RDFS(1)

RDFS(3) → no recursive call

Recursive calls

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

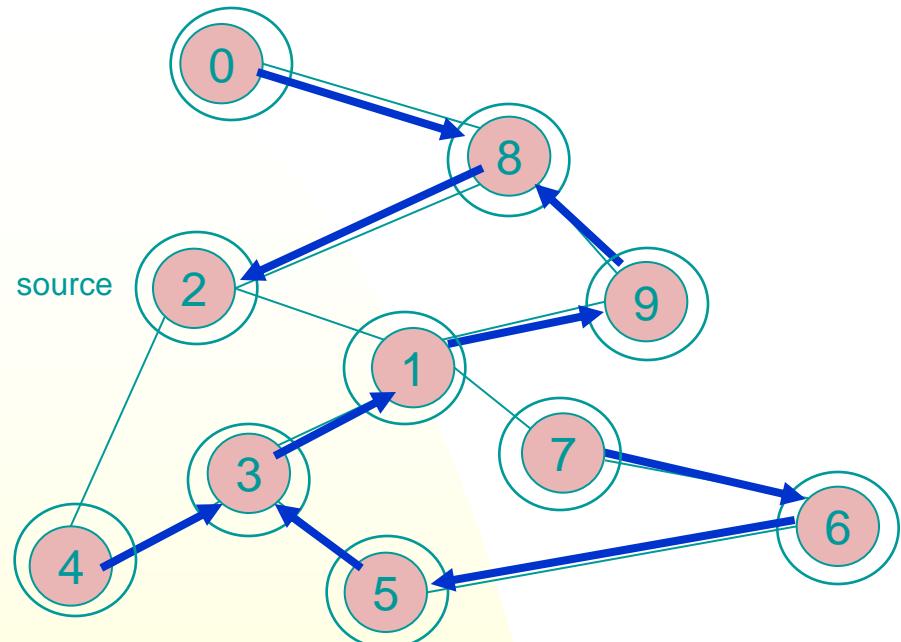
visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Visited Table (T/F)

0	T
1	T
2	-1
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

# Example



RDFS(2)

RDFS(8)

RDFS(9)

RDFS(1) → no recursive call

Recursive calls

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

Adjacency List

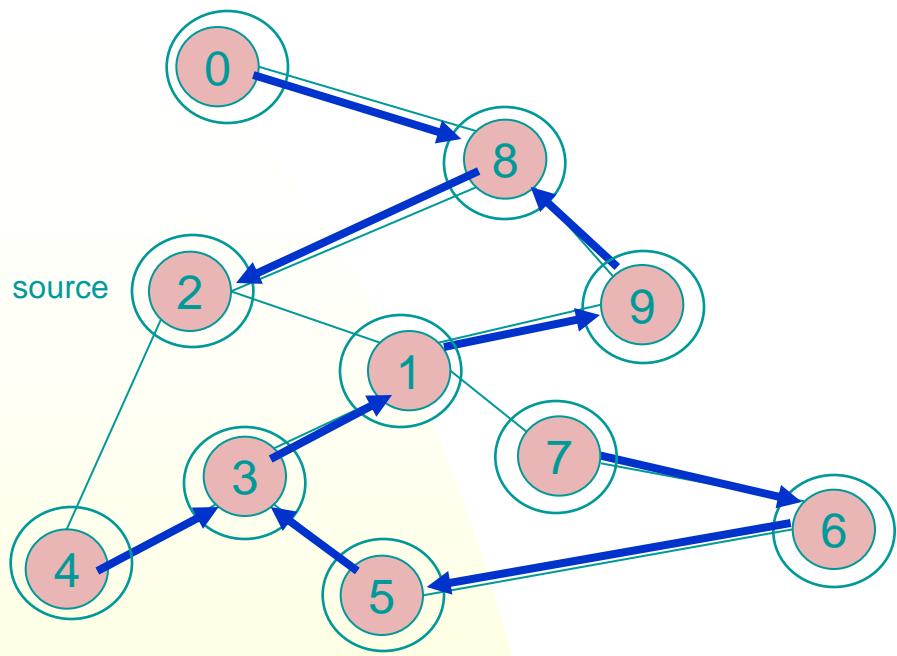
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

# Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T	8
1	T	9
2	T	-1
3	T	1
4	T	3
5	T	3
6	T	5
7	T	6
8	T	2
9	T	8

Pred

visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

RDFS( 2 )

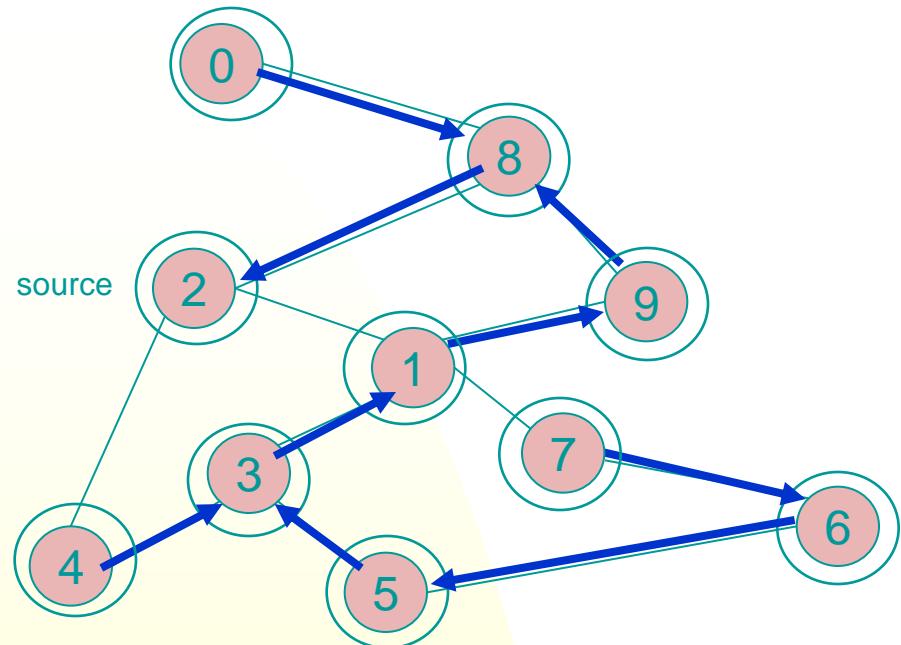
RDFS(8)

RDFS(9) → no recursive call

Recursive calls

Depth-First Search

# Example



visit sequence= {2, 8, 0, 9, 1, 3, 4, 5, 6, 7}

RDFS( 2 )

RDFS(8) → no recursive call

Recursive calls

Adjacency List

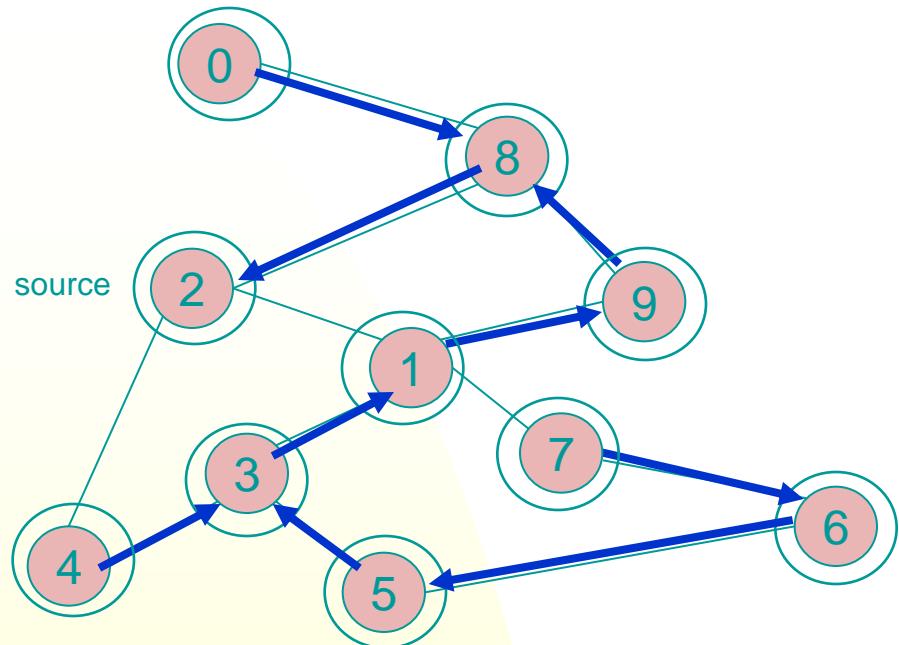
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

# Example



Recursive calls

RDFS( 2 ) → no recursive call

Adjacency List

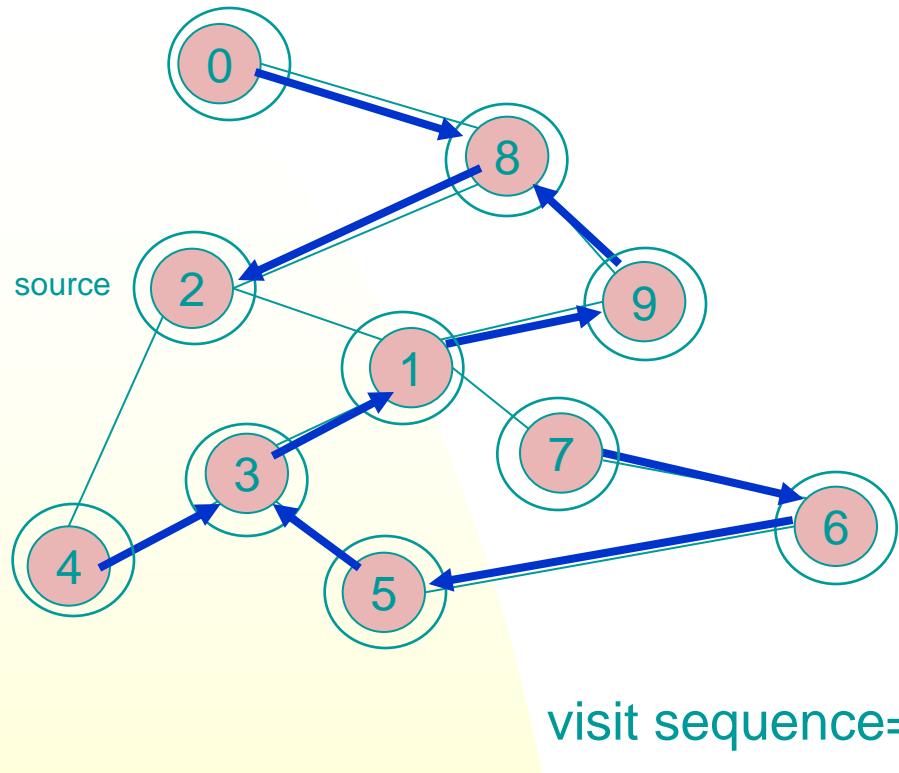
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

# Recover a path



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred

**Algorithm**  $\text{Path}(w)$

1. **if**  $\text{pred}[w] \neq -1$
2. **then**
3.      $\text{Path}(\text{pred}[w]);$
4. **output**  $w$

Try some examples.

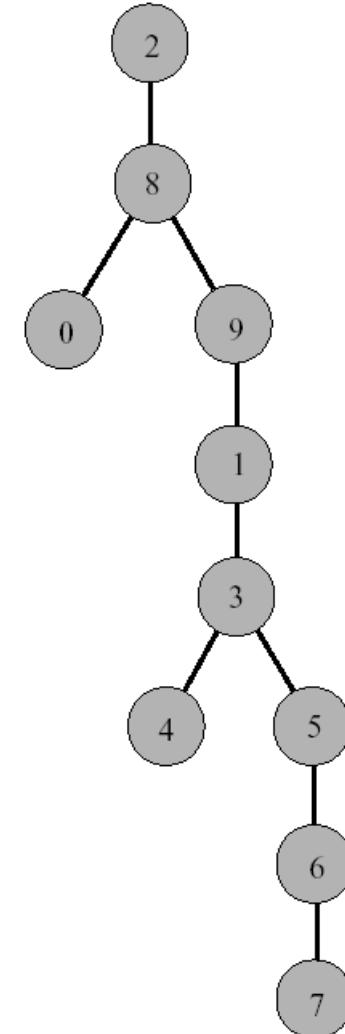
$\text{Path}(0) \rightarrow$

$\text{Path}(6) \rightarrow$

$\text{Path}(7) \rightarrow$

# DFS Tree

- The edges that we traverse during DFS (or the edges that we backtrack along) form a tree. We usually call the rooted version (rooted at the source) the DFS tree.



# Running time analysis

- The running time analysis is very similar to BFS. Let the graph be represented by an *adjacent list*, and  $n$  and  $m$  represent the number of vertices and edges in the graph respectively.

# Running time analysis

- Each (connected) vertex is visited EXACTLY one time → so there are  $O(n)$  recursive call.
- For a particular vertex  $v$ , (in the recursive call) we need to find all its neighbors. For *adjacent list* representation, it takes  $O(\text{degree}(v))$ .
  - ◆ Hence, the total number of time for all vertices is

$$\sum_{\text{vertex } v} O(\deg(v)) = O(2m) = O(m)$$

- For *adjacency list* representation, the running time for DFS is

$$O(n) + O(n) + O(2m) = O(n+m)$$

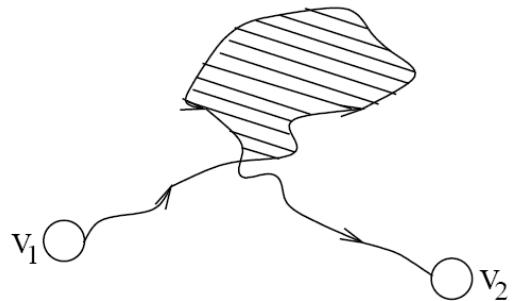
initialization

number of recursive call

find out all the neighbors

# Some applications of BFS/DFS -- Connectivity

- A graph is *connected* if and only if there exists a path between *every* pair of distinct vertices.
- If a path is not simple, then it contains cycles. Since any cycle can be bypassed, the non-simple path contains a simple path.



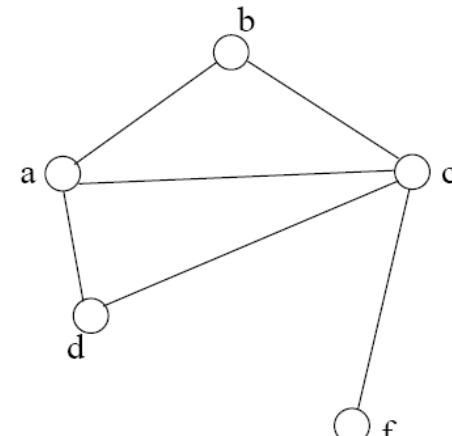
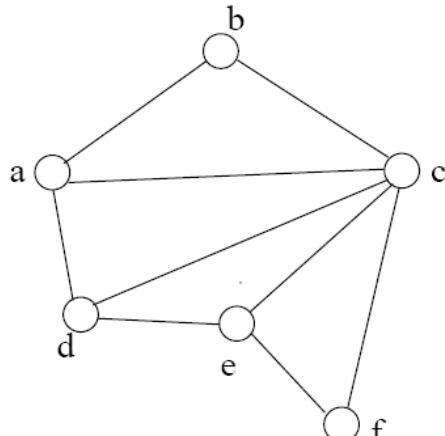
- Therefore, a graph is *connected* if and only if there exists a simple path between every pair of distinct vertices.

## Some applications of BFS/DFS -- Connectivity

- One can use BFS or DFS to decide if a graph is connected. Run BFS or DFS using an arbitrary vertex as the source. At the end, if all vertices have been visited (all flags are marked ' $T$ '), then the graph is connected. Otherwise, the graph is disconnected.
- The running time is  $O(n+m)$ .

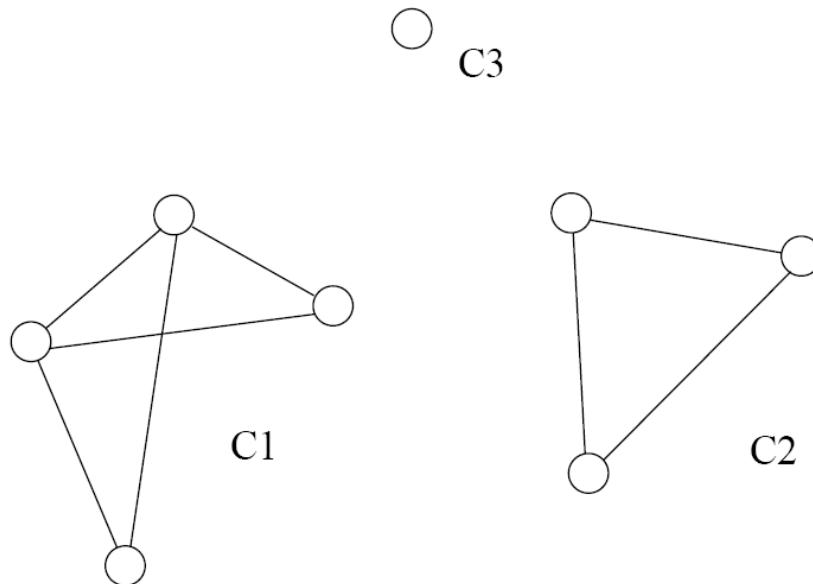
# Some applications of BFS/DFS – Connected Components

A graph  $H(V_H, E_H)$  is a *subgraph* of  $G(V_G, E_G)$  if and only if  $V_H \subset V_G$  and  $E_H \subset E_G$ .



# Some applications of BFS/DFS – Connected Components

- A connected component is a maximal connected subgraph of a graph.
- The set of connected components is unique for a given graph.



3 components: C1, C2, and C3

# Some applications of BFS/DFS – Connected Components

**Algorithm**  $DFSConn(G)$

**Input:** a graph  $G$

**Output:** the connected components

1. **for** each vertex  $v$
2.     **do**  $flag[v] := \text{false};$
3. **for** each vertex  $v$  ← For each vertex
4.     **do if**  $flag[v] = \text{false}$  ← If not visited
5.         **then** output "A new connected component:";
6.          $RDFS(v);$  ← Call  $RDFS(v)$

**Algorithm**  $RDFS(v)$

1.  $flag[v] := \text{true};$
2. output  $v;$
3. **for** each neighbor  $w$  of  $v$
4.     **do if**  $flag[w] = \text{false}$
5.         **then**  $RDFS(w);$

Basic DFS algorithm.

This will find all connected vertices to "v"

# Some applications of BFS/DFS – Connected Components

## Running time analysis

- Running time for each  $i$  connected-component

$$O(n_i + m_i)$$

- Question:
  - ◆ Can two connected components have the same edge?
  - ◆ Can two connected components have the same vertex?
- It follows

$$\sum_i O(n_i + m_i) = O\left(\sum_i n_i + \sum_i m_i\right) = O(n + m)$$

# IKI10400 • Struktur Data & Algoritma: Hashtables

**Fakultas Ilmu Komputer • Universitas Indonesia**

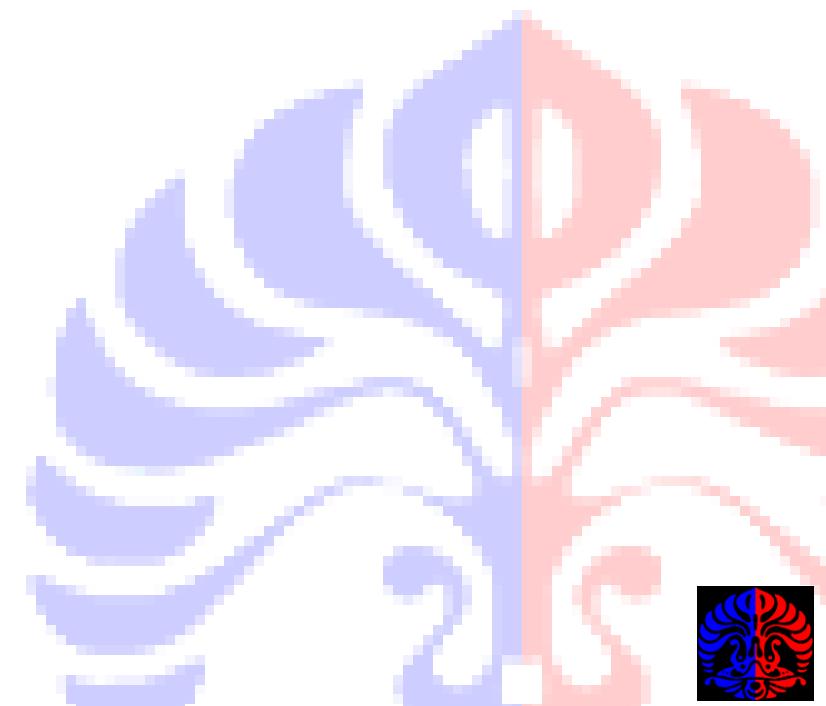
*Slide acknowledgments:*

Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung



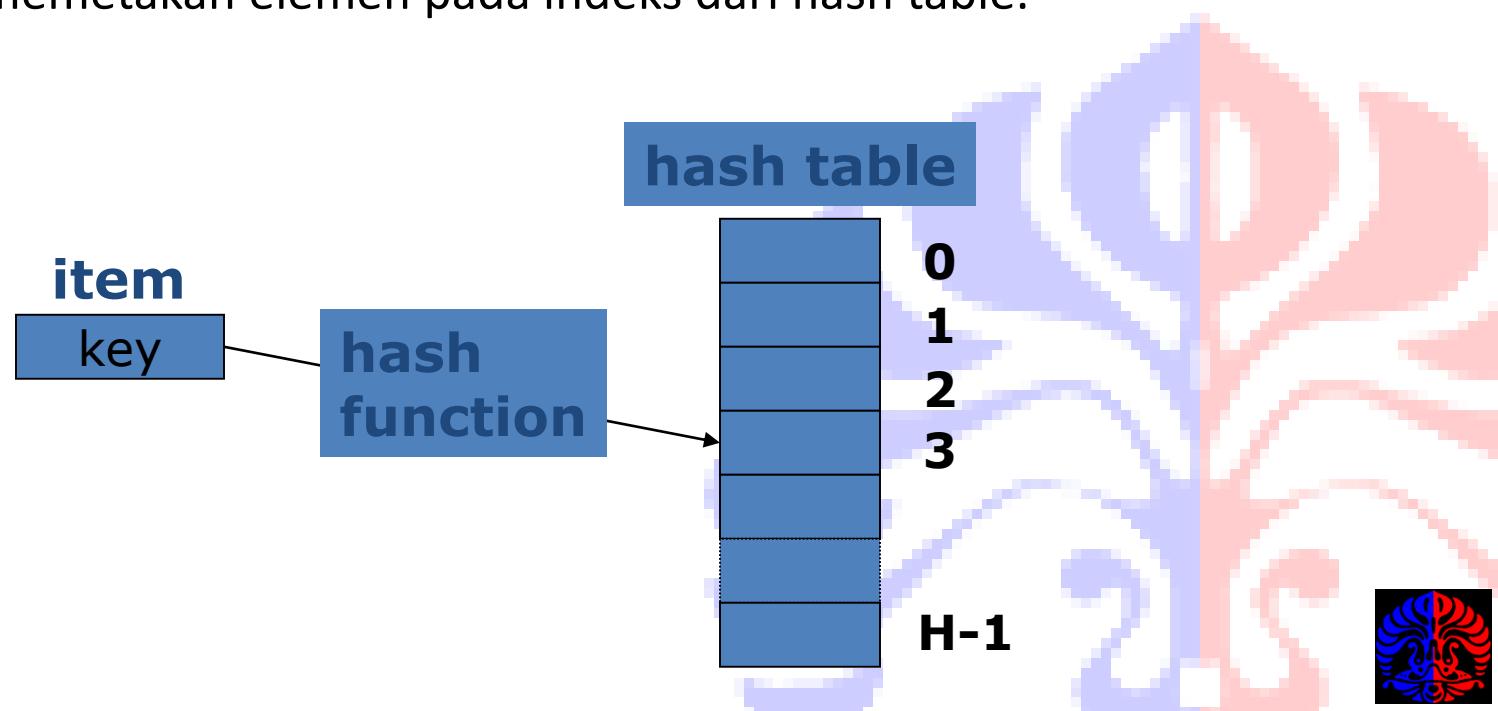
# Outline

- Hashing
  - Definition
  - Hash function
  - Collision resolution
    - Open hashing
      - Separate chaining
    - Closed hashing (Open addressing)
      - Linear probing
      - Quadratic probing
      - Double hashing
    - Primary Clustering, Secondary Clustering
  - Access: insert, find, delete



# Hash Tables

- Hashing digunakan untuk menyimpan data yang cukup besar pada ADT yang disebut hash table.
- Ukuran Hash table ( $H$ -size), biasanya lebih besar dari jumlah data yang hendak disimpan.
- **load factor** ( $\lambda$ ) adalah perbandingan antara data yang disimpan dengan ukuran hash table.
- **Fungsi Hash** memetakan elemen pada indeks dari hash table.



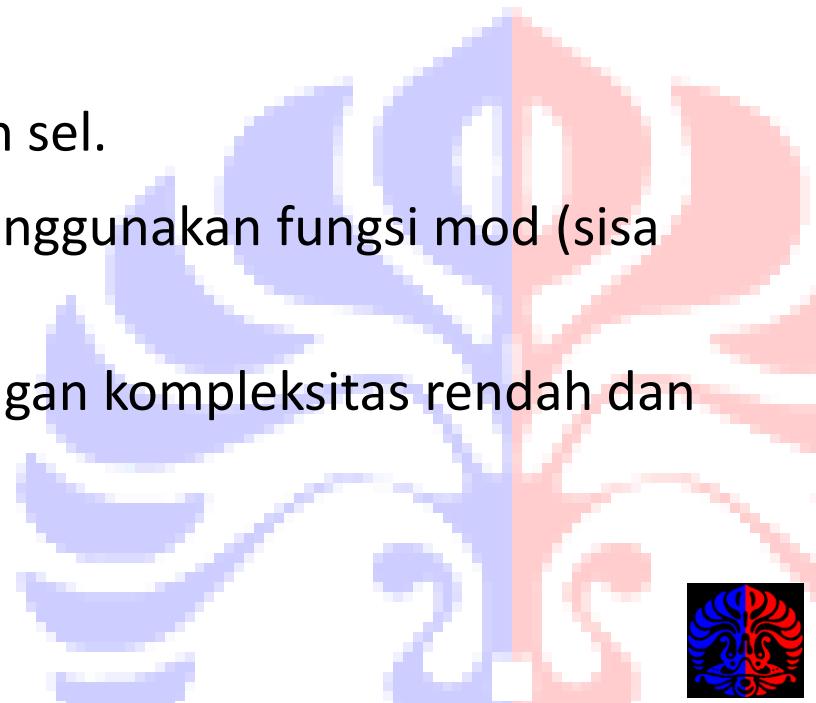
# Hash Tables (2)

- Hashing adalah teknik untuk melakukan penambahan, penghapusan dan pencarian dengan **constant average time**.
- Untuk menambahkan data atau pencarian, ditentukan key dari data tersebut dan digunakan sebuah fungsi hash untuk menetapkan lokasi untuk key tersebut.
- Hash tables adalah arrays dengan sel-sel yang ukurannya telah ditentukan dan dapat berisi data atau key yang berkesesuaian dengan data.
- Untuk setiap key, digunakan fungsi hash untuk memetakan key pada bilangan dalam rentang 0 hingga  $H\text{-size}-1$ .



# Fungsi Hash

- Fungsi hash harus memiliki sifat berikut:
  - mudah dihitung.
  - dua key yang berbeda akan dipetakan pada dua sel yang berbeda pada array. (secara umum tidak bisa berlaku, mengapa? ).
    - dapat dicapai dengan menggunakan direct-address table dimana semesta dari key relatif kecil.
  - membagi key secara rata pada seluruh sel.
- Sebuah fungsi hash sederhana adalah menggunakan fungsi mod (sisa bagi) dengan bilangan prima.
- Dapat menggunakan manipulasi digit dengan kompleksitas rendah dan distribusi key yang rata.



# Fungsi Hash: Truncation

- Sebagian dari key dapat dibuang/diabaikan, bagian key sisanya digabungkan untuk membentuk index.
- contoh:

<i>Phone no:</i>	<i>index</i>
731-3018	338
539-2309	329
428-1397	217

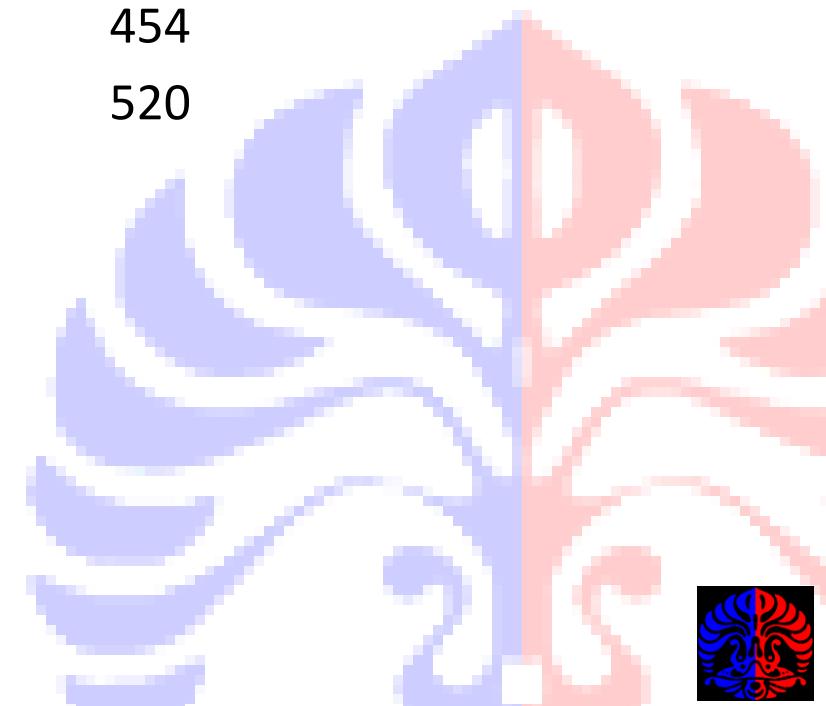


# Fungsi Hash: Folding

- Data dipecah menjadi beberapa bagian, kemudian tiap bagian tersebut digabungkan lagi dalam bentuk lain.
- contoh.

*Phone no:*    *3-group*

<i>Phone no:</i>	<i>3-group</i>	<i>index</i>
7313018	73+13+018	104
5392309	53+92+309	454
4281397	42+81+397	520



# Fungsi Hash: Modular arithmetic

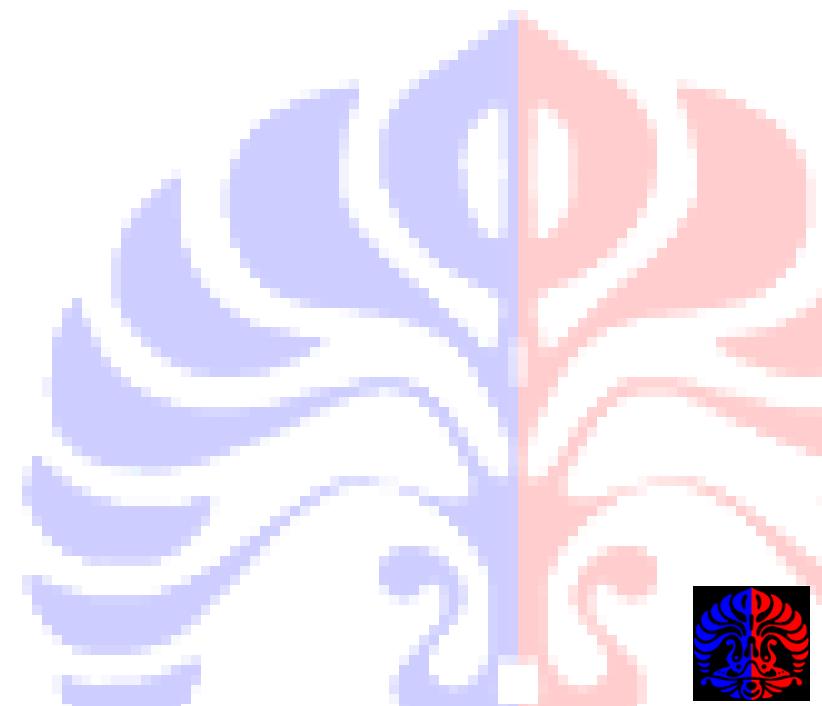
- Melakukan konversi data ke bentuk bilangan bulat, dibagi dengan ukuran hash table, dan mengambil hasil sisa baginya sebagai indeks.
- contoh:

<i>Phone no:</i>	<i>2-group</i>	<i>index</i>
7313018	731+3018	$3749 \% 100 = 49$
5392309	539+2309	$2848 \% 100 = 48$
4281397	428+1397	$1825 \% 100 = 25$



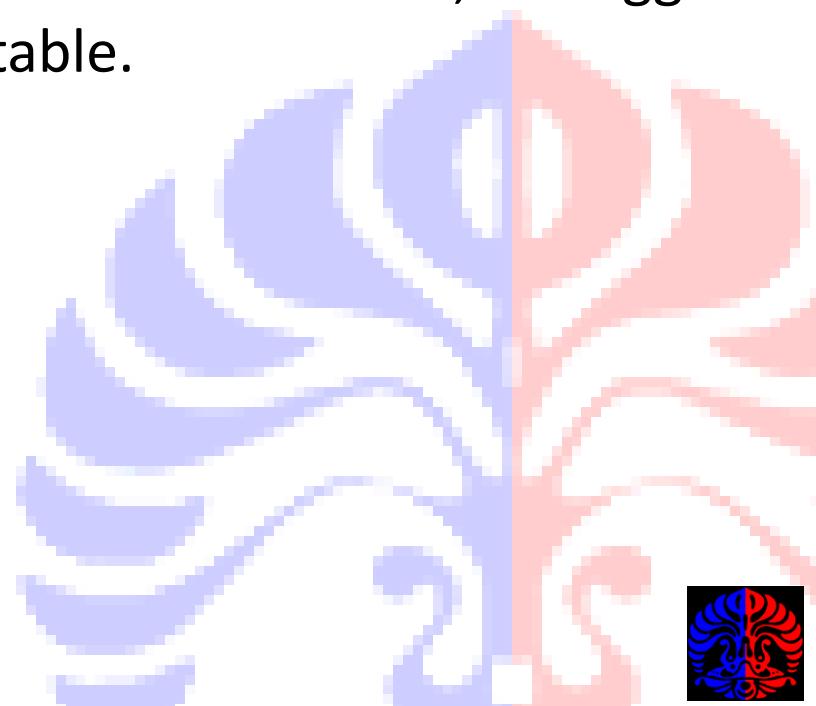
# Memilih Fungsi Hash

- Sebuah fungsi hash yang bagus memiliki dua kriteria:
  1. Harus dapat cepat dihitung.
  2. Harus meminimalkan juga collisions yang terjadi.



# Contoh Fungsi Hash

- Fungsi Hash untuk string
  - $X = 128$
  - $A_3 X^3 + A_2 X^2 + A_1 X^1 + A_0 X^0$
  - $((A_3 X + A_2) X + A_1) X + A_0$
- Hasil dari fungsi hash jauh lebih besar dari ukuran table, sehingga perlu di modulo dengan ukuran hash table.



# Contoh Fungsi Hash

```
int hash(String key, int tableSize) {  
    int hashVal = 0;  
    for (int i=0; i < key.length(); i++) {  
        hashVal = (hashVal * 128  
                  + key.charAt(i)) % tableSize;  
    }  
    return hashVal % tableSize;  
}
```

- Modulo
  - $(A + B) \% C = (A \% C + B \% C) \% C$
  - $(A * B) \% C = (A \% C * B \% C) \% C$



# Contoh Fungsi Hash

```
int hash(String key, int tableSize) {  
    int hashVal = 0;  
    for (int i=0; i < key.length(); i++) {  
        hashVal = (hashVal * 37  
                  + key.charAt(i));  
    }  
    hashVal %= tableSize;  
    if (hashVal < 0) {  
        hashVal += tableSize;  
    }  
    return hashVal;  
}
```



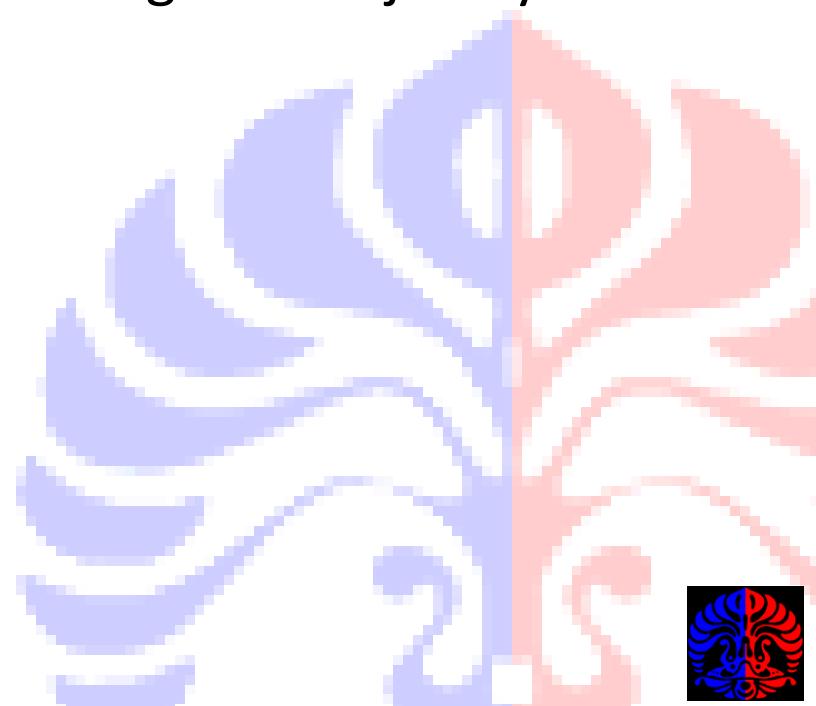
# Contoh Fungsi Hash

```
int hash(String key, int tableSize) {  
    int hashVal = 0;  
    for (int i=0; i < key.length(); i++) {  
        hashVal += key.charAt(i)  
    }  
    return hashVal % tableSize;  
}
```



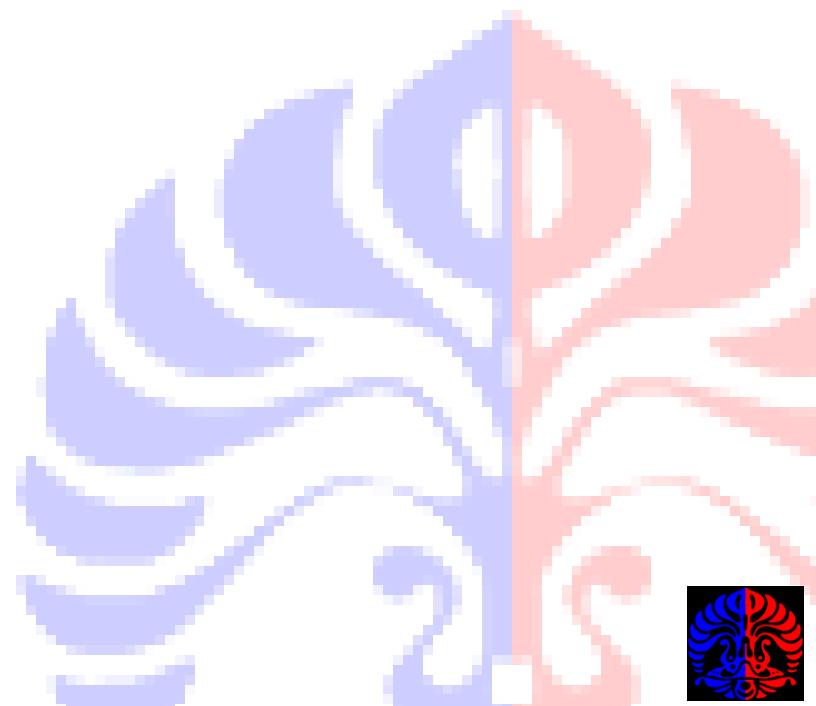
# Collision Resolution

- Collision Resolution: Penyelesaian bila terjadi ***collision (tabrakan)***.
- Dikatakan terjadi ***collision*** jika dua buah *keys* dipetakan pada sebuah sel.
- ***Collision*** bisa terjadi saat melakukan ***insertion***.
- Dibutuhkan prosedur tambahan untuk mengatasi terjadinya ***collision***.
- Ada dua strategi umum:
  - Closed Hashing (Open Addressing)
  - Open Hashing (Chaining)



# *Closed Hashing*

- Ide: mencari alternatif sel lain pada tabel.
  - Pada proses insertion, coba sel lain sesuai urutan dengan menggunakan fungsi pencari urutan seperti berikut:
    - $h_i(x) = (\text{hash}(x) + f(i)) \bmod H\text{-size}$        $f(0) = 0$
  - Fungsi  $f$  digunakan sebagai pengatur strategy *collision resolution*.
  - Bagaimana bentuk fungsi  $f$  ?



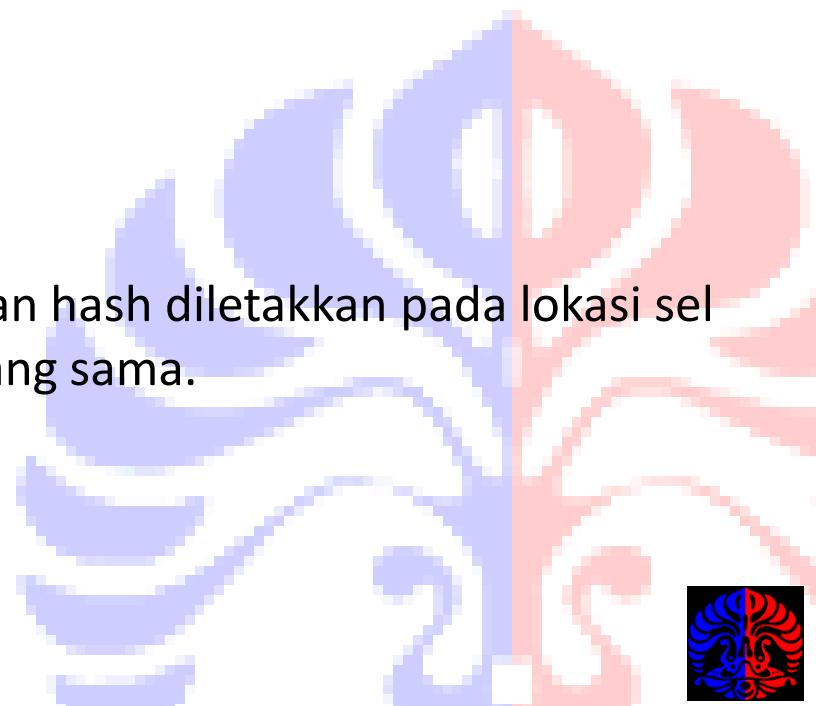
# Closed Hashing

- Beberapa strategy/alternatif untuk menentukan bentuk fungsi  $f$ , yaitu:
  - *Linear probing*
  - *Quadratic probing*
  - *Double hashing*



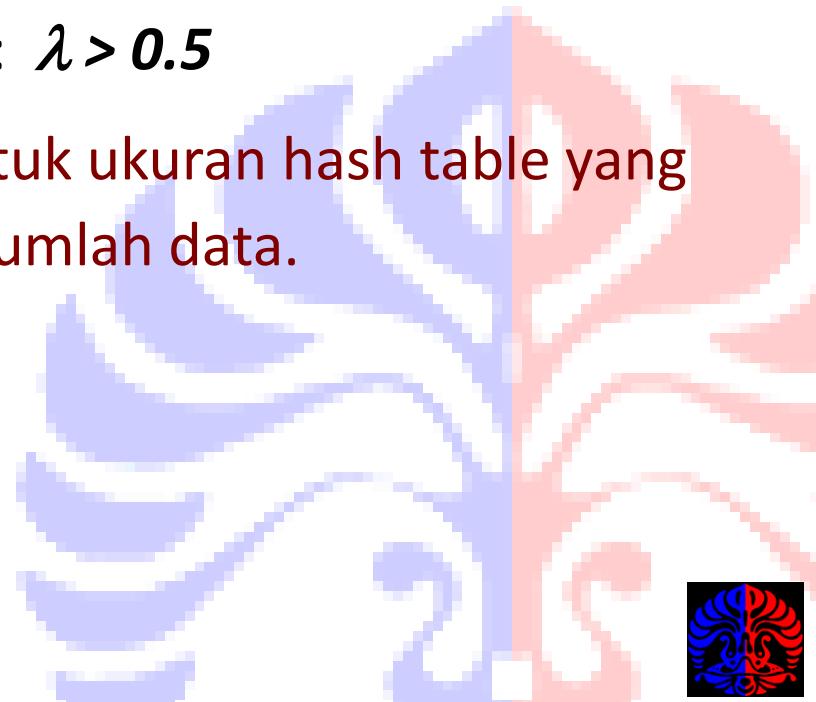
# Linear Probing

- Gunakan fungsi linear  
$$f(i) = i$$
- Bila terjadi collision, cari posisi pertama pada tabel yang terdekat dengan posisi yang seharusnya.
- fungsi linear relatif paling sederhana.
  - Mudah diimplementasikan.
- Dapat menimbulkan masalah:  
***primary clustering***
  - Elemen-elemen yang menurut perhitungan hash diletakkan pada lokasi sel berbeda, diarahkan pada sel pengganti yang sama.

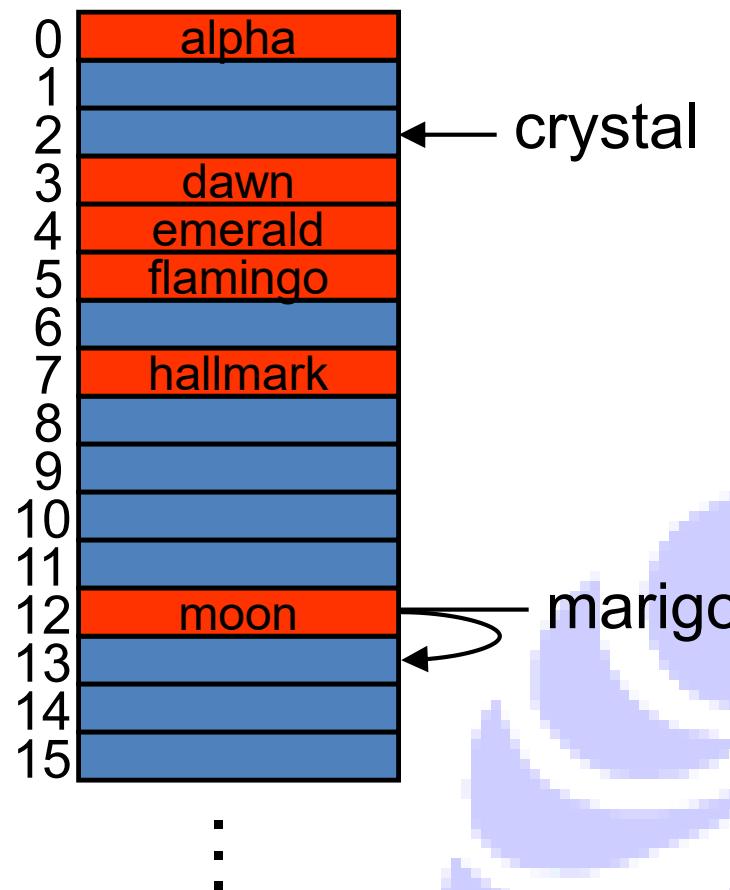


# Linear Probing: Load factor

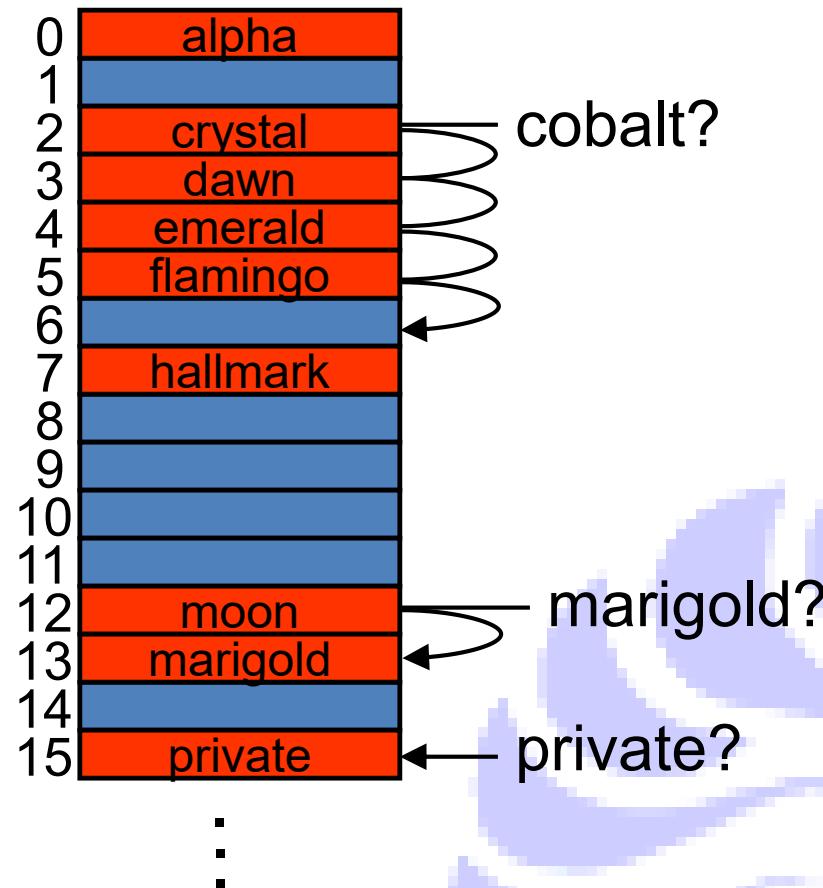
- Kompleksitas dari teknik ini bergantung pada nilai  $\lambda$  (**load factor**).
- **Definisi  $\lambda$  (load factor):**
  - Untuk hash table  $T$  dengan ukuran  $m$ , yang menyimpan  $n$  data.
  - $\lambda$  (**Load factor**) dari  $T$  adalah  $n/m$
- **Linear Probing** tidak disarankan bila:  $\lambda > 0.5$
- **Linear Probing** hanya disarankan untuk ukuran hash table yang ukurannya lebih besar dua kali dari jumlah data.



# Hashing - insert

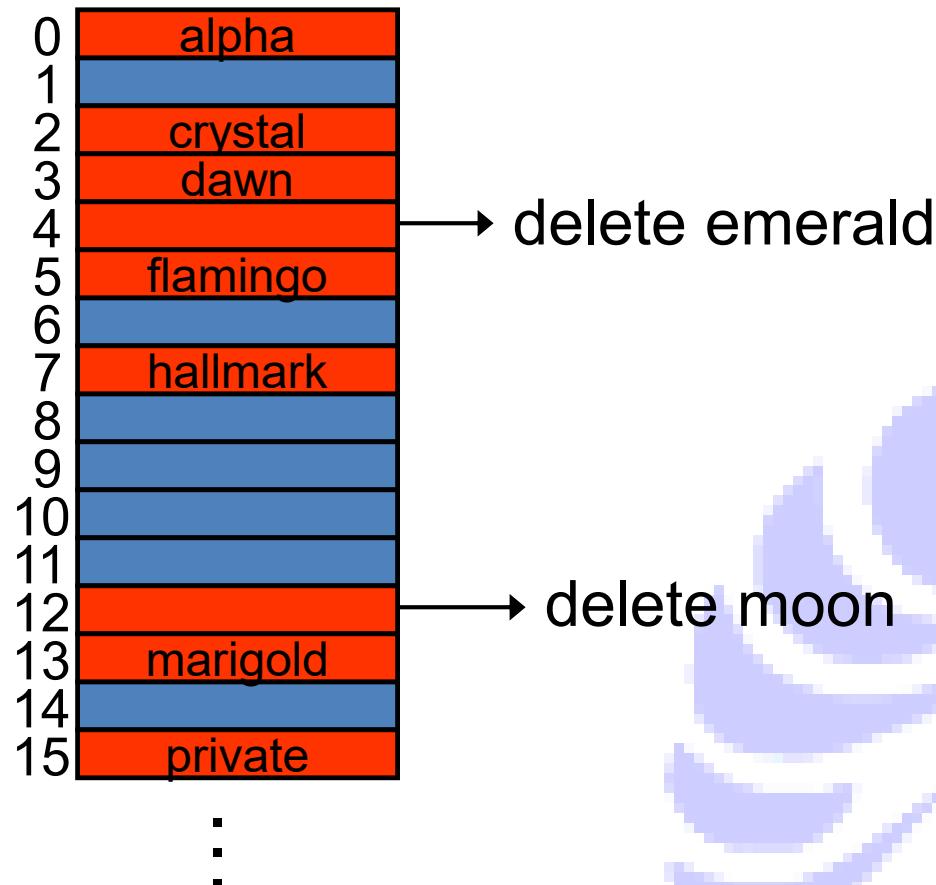


# Hashing - lookup

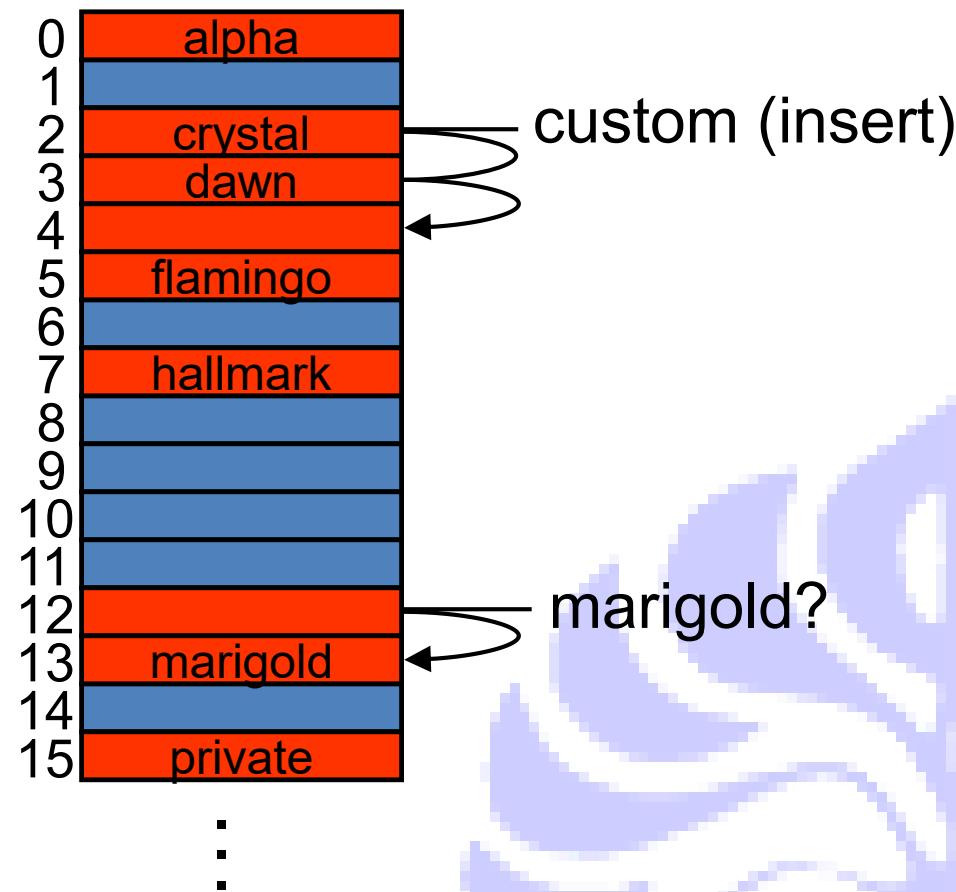


# Hashing - delete

- lazy deletion - mengapa?

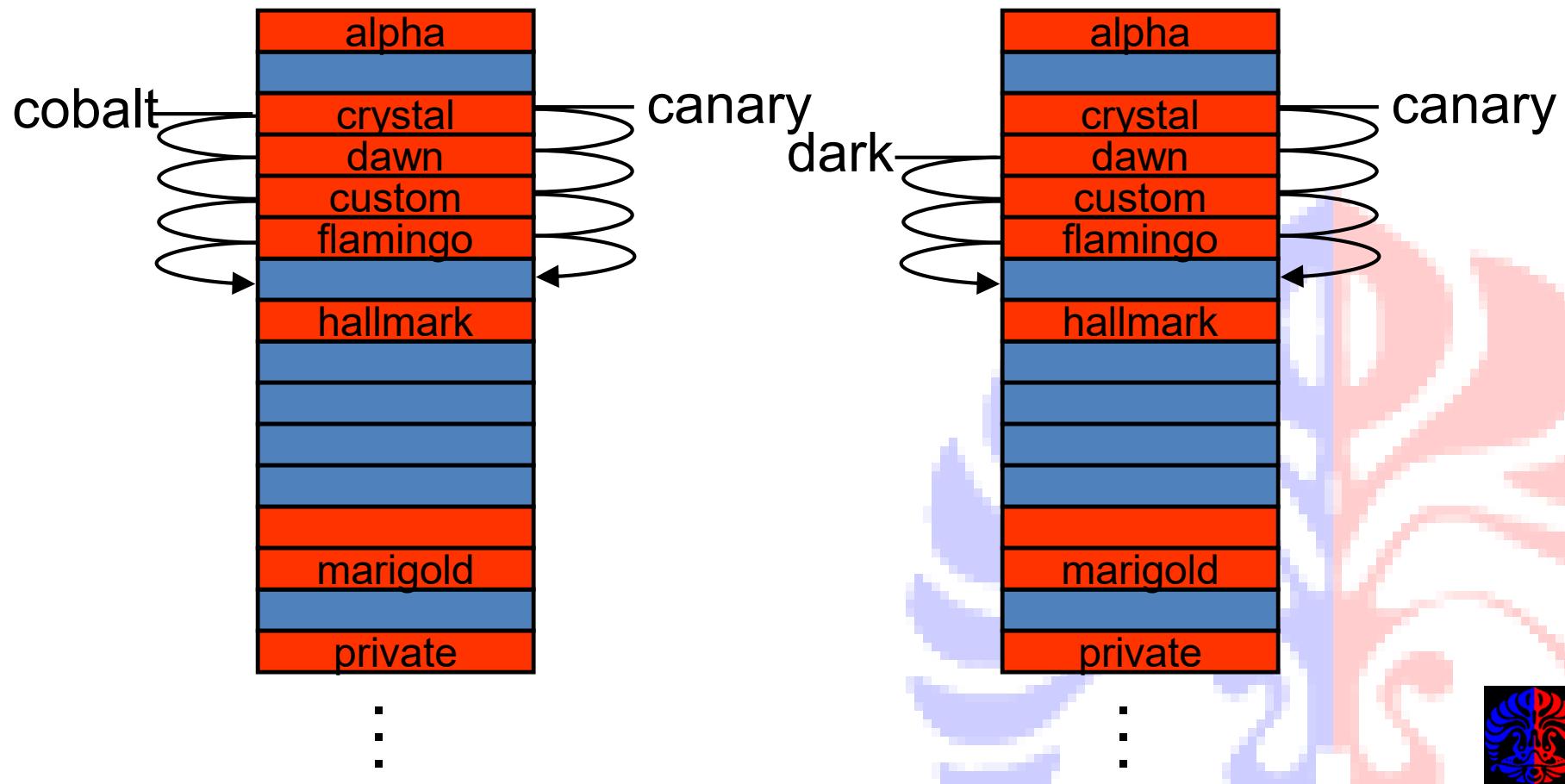


# Hashing - operation after delete



# Primary Clustering

- Elemen-elemen yang menurut perhitungan hash diletakkan pada lokasi sel berbeda, diarahkan (probe) pada sel pengganti yang sama.

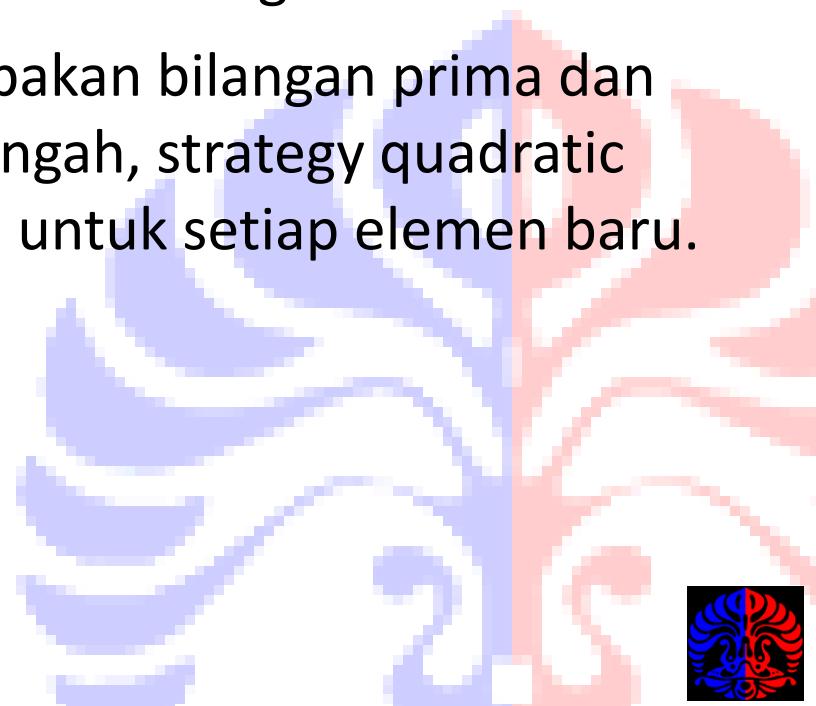


# Quadratic Probing

- Menghindari *primary clustering* dengan menggunakan fungsi:

$$f(i) = i^2$$

- Menimbulkan banyak permasalahan bila hash table telah terisi lebih dari setengah.
- Perlu dipilih ukuran hash table yang bukan bilangan kuadrat.
- Dengan ukuran hash table yang merupakan bilangan prima dan hash table yang terisi kurang dari setengah, strategy quadratic probe dapat selalu menemukan lokasi untuk setiap elemen baru.



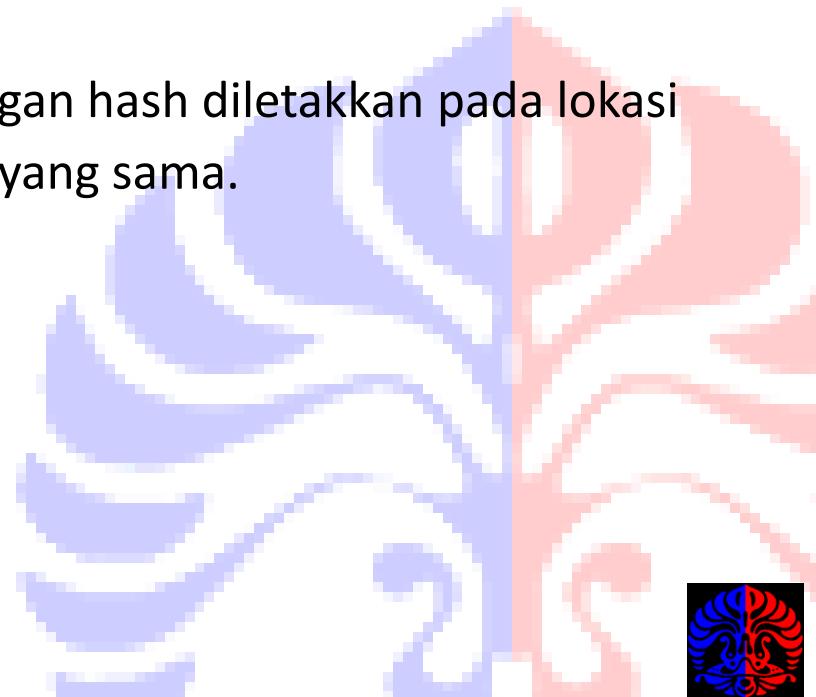
# Quadratic Probing

- Dapat melakukan *increment* bila terjadi *collision*
- Perhatikan bahwa fungsi *quadratic* dapat dijabarkan sebagai berikut:

$$f(i) = i^2 = f(i-1) + 2i - 1.$$

- Menimbulkan ***second clustering:***

- Elemen-elemen yang menurut perhitungan hash diletakkan pada lokasi sel sama, diarahkan pada sel pengganti yang sama.

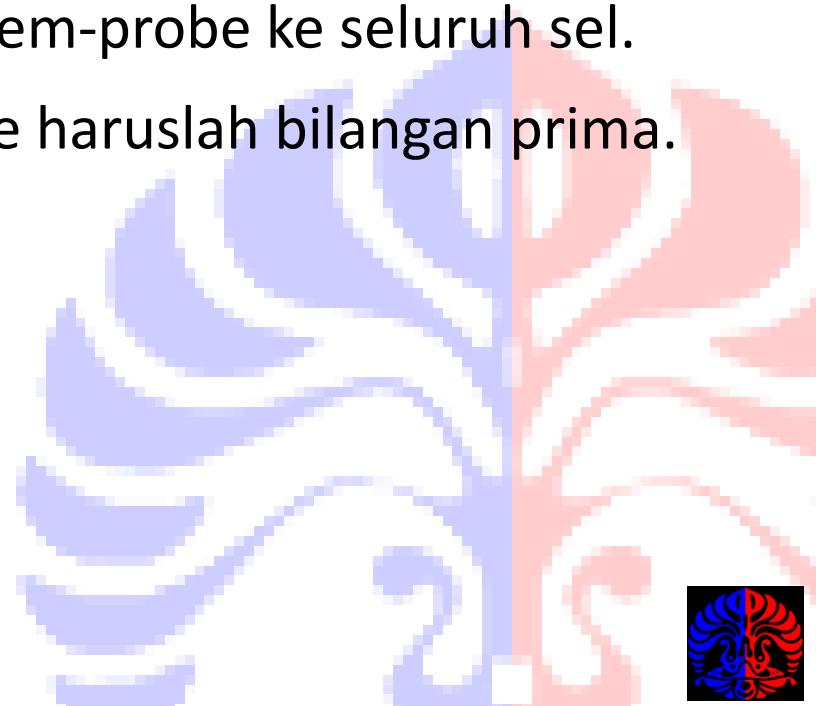


# Double hashing

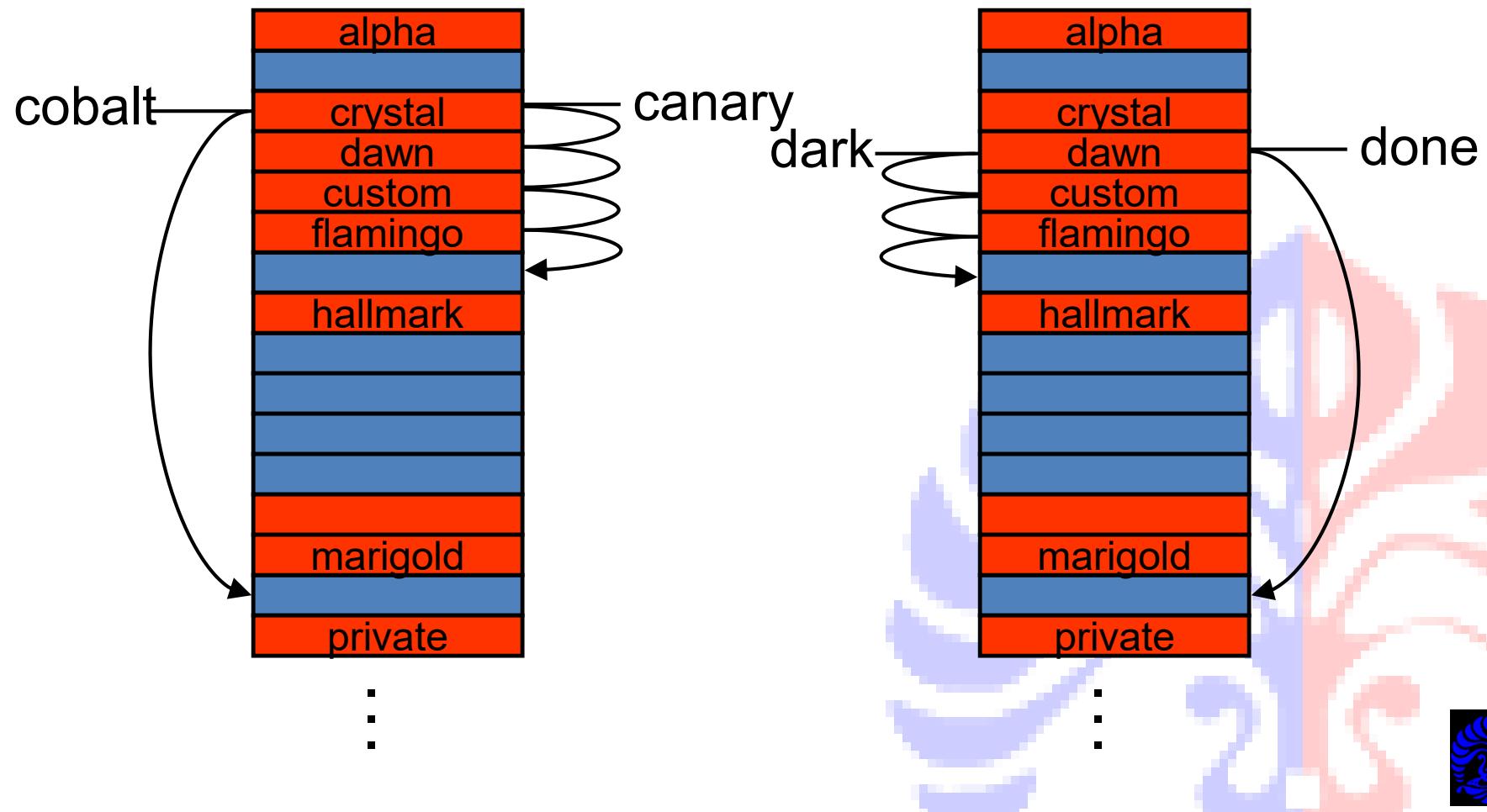
- fungsi untuk *collision resolution* disusun dengan fungsi hash seperti :

$$f(i) = i * \text{hash2}(x)$$

- Setiap saat faktor *hash2(x)* ditambahkan pada *probe*.
- Harus hati-hati dalam memilih fungsi hash kedua untuk menjamin agar tidak menghasilkan nilai 0 dan mem-probe ke seluruh sel.
- Salah satu syaratnya ukuran hash table haruslah bilangan prima.

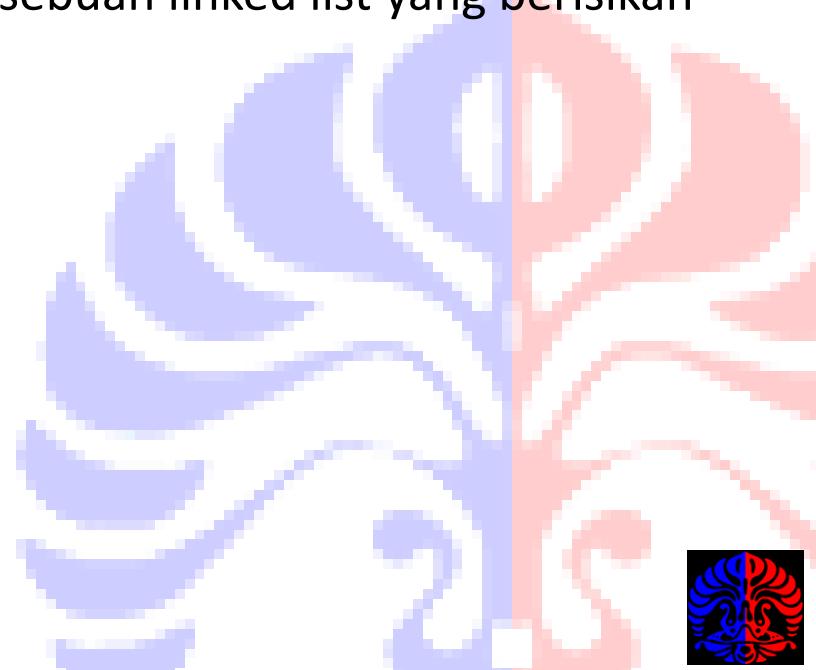


# Double Hashing



# Open Hashing

- Permasalahan *Collision* diselesaikan dengan menambahkan seluruh elemen yang memiliki nilai hash sama pada sebuah set.
- *Open Hashing:*
  - Menyediakan sebuah linked list untuk setiap elemen yang memiliki nilai hash sama.
  - Tiap sel pada hash table berisi pointer ke sebuah linked list yang berisikan data/element.

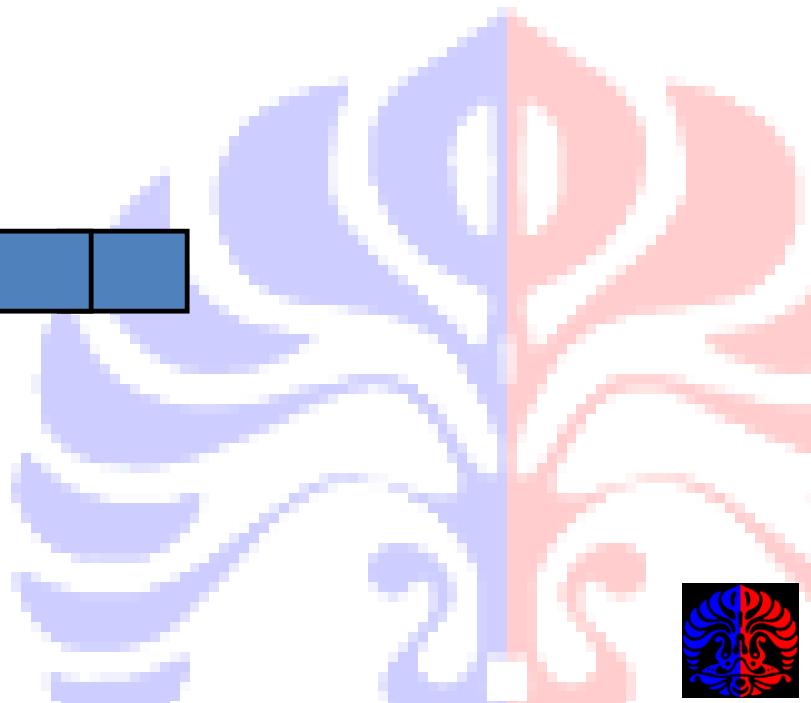
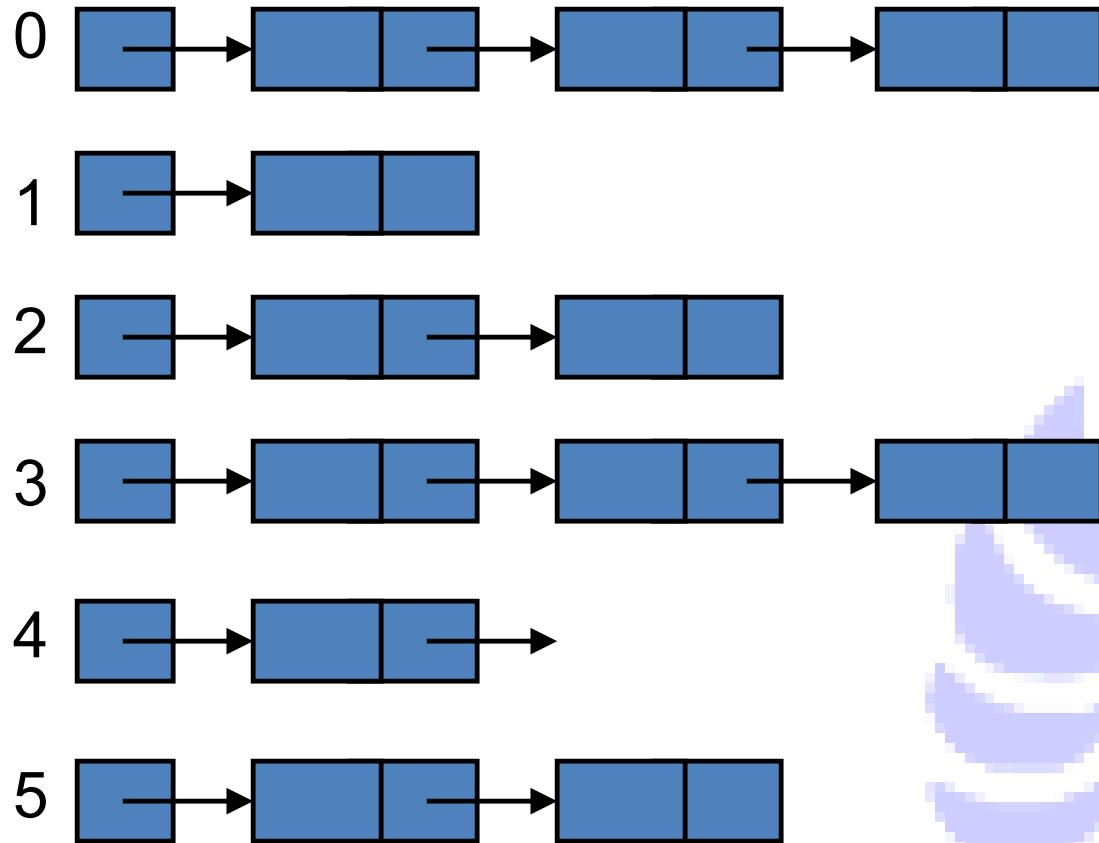


# *Open Hashing*

- Fungsi dan analisa *Open Hashing*:
  - Menambahkan sebuah elemen ke dalam tabel:
    - Dilakukan dengan menambahkan elemen pada akhir atau awal linked-list yang sesuai dengan nilai hash.
    - Bergantung apakah perlu ada pengujian nilai duplikasi atau tidak.
    - Dipengaruhi berapa sering elemen terakhir akan diakses.



# Open Hashing



# Open Hashing

- Untuk pencarian, gunakan fungsi hash untuk menentukan linked list mana yang memiliki elemen yang dicari, kemudian lakukan pembacaan terhadap linked list tersebut.
- Penghapusan dilakukan pada linked list setelah pencarian elemen dilakukan.
- Dapat saja digunakan struktur data lain selain linked list untuk menyimpan elemen yang memiliki fungsi hash yang sama tersebut.
- Kelebihan utama dari metode ini adalah dapat menyimpan data yang tak terbatas. (**dynamic expansion**).
- Kekurangan utama adalah penggunaan memory pada tiap sel.



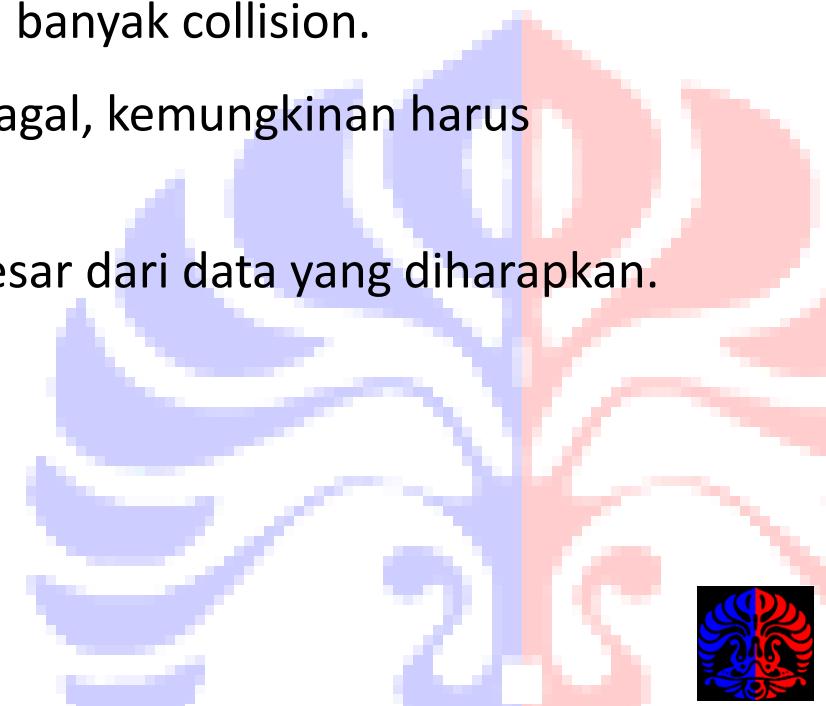
# Analisa *Open Hash*

- Secara umum panjang dari linked list yang dihasilkan sejalan dengan nilai  $\lambda$ .
- Kompleksitas insertion bergantung pada fungsi hash dan insertion pada linked-list.
- Untuk pencarian, kompleksitasnya adalah waktu konstan dalam mengevaluasi fungsi hash + pembacaan list.
- *Worst case*  $O(n)$  untuk pencarian.
- *Average case* bergantung pada  $\lambda$ .
- Aturan umum untuk *open hashing* adalah untuk menjaga agar:  $\lambda \approx 1$ .
- Digunakan untuk data yang ukuran-nya dinamic.



# Isu-isu lain

- Hal-hal lain yang umum dan perlu diperhatikan pada metode *closed hashing resolutions*:
  - Proses menghapus agak membingungkan karena tidak benar-benar dihapus.
  - Secara umum lebih sederhana dari pada open hashing.
  - Bagus bila diperkirakan tidak akan terjadi banyak collision.
  - Jika pencarian berdasarkan fungsi hash gagal, kemungkinan harus mencari/membaca seluruh tabel.
  - Menggunakan ukuran table yang lebih besar dari data yang diharapkan.



# HASHSET

Contoh Implementasi Hash Table  
Quadratic Probing

```
public class HashSet<AnyType> extends AbstractCollection<AnyType>
implements Set<AnyType>{
    /**
     * Construct an empty HashSet.
     */
    public HashSet( ) {
        allocateArray( DEFAULT_TABLE_SIZE );
        clear( );
    }

    /**
     * Construct a HashSet from any collection.
     */
    public HashSet( Collection<? extends AnyType> other ) {
        allocateArray( nextPrime( other.size( ) * 2 ) );
        clear( );

        for( AnyType val : other )
            add( val );
    }
}
```



```
/**  
 * This method is not part of standard Java.  
 * Like contains, it checks if x is in the set.  
 * If it is, it returns the reference to the matching  
 * object; otherwise it returns null.  
 * @param x the object to search for.  
 * @return if contains(x) is false, the return value is null;  
 *         otherwise, the return value is the object that causes  
 *         contains(x) to return true.  
 */  
public AnyType getMatch( AnyType x )  
{  
    int currentPos = findPos( x );  
  
    if( isActive( array, currentPos ) )  
        return (AnyType) array[ currentPos ].element;  
    return null;  
}
```

```

/**
 * Tests if some item is in this collection.
 * @param x any object.
 * @return true if this collection contains an item equal to x.
 */
public boolean contains( Object x )
{
    return isActive( array, findPos( x ) );
}

/**
 * Tests if item in pos is active.
 * @param pos a position in the hash table.
 * @param arr the HashEntry array (can be oldArray during rehash).
 * @return true if this position is active.
 */
private static boolean isActive( HashEntry [ ] arr, int pos )
{
    return arr[ pos ] != null && arr[ pos ].isActive;
}

```

```
/**  
 * Adds an item to this collection.  
 * @param x any object.  
 * @return true if this item was added to the collection.  
 */  
public boolean add( AnyType x )  
{  
    int currentPos = findPos( x );  
    if( isActive( array, currentPos ) )  
        return false;  
  
    if( array[ currentPos ] == null )  
        occupied++;  
    array[ currentPos ] = new HashEntry( x, true );  
    currentSize++;  
    modCount++;  
  
    if( occupied > array.length / 2 )  
        rehash();  
  
    return true;  
}
```

```
/**  
 * this inner class is needed to encapsulate the element  
 * and provide the flag field required by the Hash Table  
 */  
  
private static class HashEntry  
{  
    public Object element; // the element  
    public boolean isActive; // false if marked deleted  
  
    public HashEntry( Object e )  
    {  
        this( e, true );  
    }  
  
    public HashEntry( Object e, boolean i )  
    {  
        element = e;  
        isActive = i;  
    }  
}
```

```

/**
 * Private routine to perform rehashing.
 * Can be called by both add and remove.
 */
private void rehash( )      {
    HashEntry [ ] oldArray = array;

        // Create a new, empty table
    allocateArray( nextPrime( 4 * size( ) ) );
    currentSize = 0;
    occupied = 0;

        // Copy table over
    for( int i = 0; i < oldArray.length; i++ )
        if( isActive( oldArray, i ) )
            add( (AnyType) oldArray[ i ].element );
}

/**
 * Internal method to allocate array.
 * @param arraySize the size of the array.
 */
private void allocateArray( int arraySize )      {
    array = new HashEntry[ nextPrime( arraySize ) ];
}

```

```
/***
 * Removes an item from this collection.
 * @param x any object.
 * @return true if this item was removed from the collection.
 */
public boolean remove( Object x )
{
    int currentPos = findPos( x );
    if( !isActive( array, currentPos ) )
        return false;

    array[ currentPos ].isActive = false;
    currentSize--;
    modCount++;

    if( currentSize < array.length / 8 )
        rehash( );

    return true;
}
```

```

/**
 * Method that performs quadratic probing resolution.
 * @param x the item to search for.
 * @return the position where the search terminates.
 */
private int findPos( Object x )    {
    int offset = 1;
    int currentPos = ( x == null ) ? 0 :
                           Math.abs( x.hashCode( ) % array.length );

    while( array[ currentPos ] != null )          {
        if( x == null )                      {
            if( array[ currentPos ].element == null )
                break;
        }
        else if( x.equals( array[ currentPos ].element ) )
            break;

        currentPos += offset;                  // Compute ith probe
        offset += 2;
        if( currentPos >= array.length )      // Implement the mod
            currentPos -= array.length;
    }
    return currentPos;
}

```

# OPEN HASHING (CHAINING)

```
/**  
 * this inner class is needed to encapsulate the element  
 * and provide the next field to implement the linked-list chaining  
 */  
  
private static class HashEntry {  
    public Object element; // the element  
    public HashEntry next; // linked list chaining.  
  
public HashEntry( Object e ) {  
    this( e, null );  
}  
  
public HashEntry( Object e, HashEntry n ) {  
    element = e;  
    next = n;  
}  
}
```

```
/**  
 * Adds an item to this collection.  
 * @param x any object.  
 * @return true if this item was added to the collection.  
 */  
public boolean add( AnyType x )  
{  
  
    if( getMatch( x ) )  
        return false;  
  
    int currentPos = x.hashCode();  
  
    array[ currentPos ] = new HashEntry( x, array[currentPost]);  
    currentSize++;  
    return true;  
}
```

# HASHMAP

```
/**  
 * Hash table implementation of the Map.  
 */  
public class HashMap<KeyType,valueType>  
    extends MapImpl<KeyType,valueType>{  
  
    /**  
     * Construct an empty HashMap.  
     */  
    public HashMap( ) {  
        super( new HashSet<Map.Entry<KeyType,valueType>>( ) );  
    }  
  
    /**  
     * Construct a HashMap with same key/value pairs as another map.  
     * @param other the other map.  
     */  
    public HashMap( Map<KeyType,valueType> other ) {  
        super( other );  
    }  
}
```

```

public int hashCode( )
{
    KeyType k = getKey( );
    return k == null ? 0 : k.hashCode( );
}

 /**
 * Computes the hashCode for this String.
 * A String is represented by an array of Character.
 * This is done with int arithmetic,
 * where ** represents exponentiation, by this formula:<br>
 * <code>s[0]*31**(n-1) + s[1]*31**(n-2) + ... + s[n-1]</code>.
 *
 * @return hashCode value of this String
 */

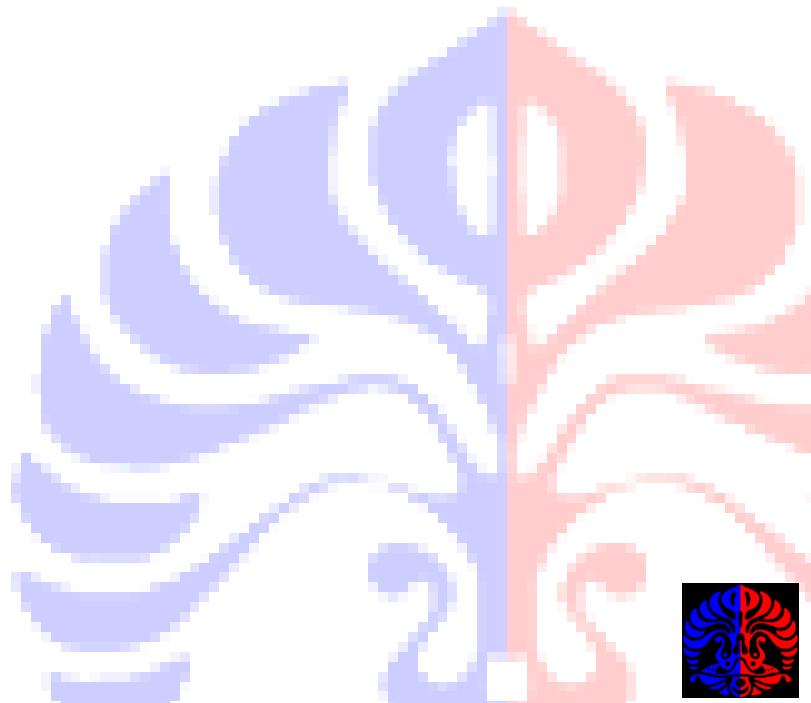
public int hashCode() {

    int hashCode = 0;
    int limit = count + offset;
    for (int i = offset; i < limit; i++)
        hashCode = hashCode * 31 + value[i];

    return hashCode;
}

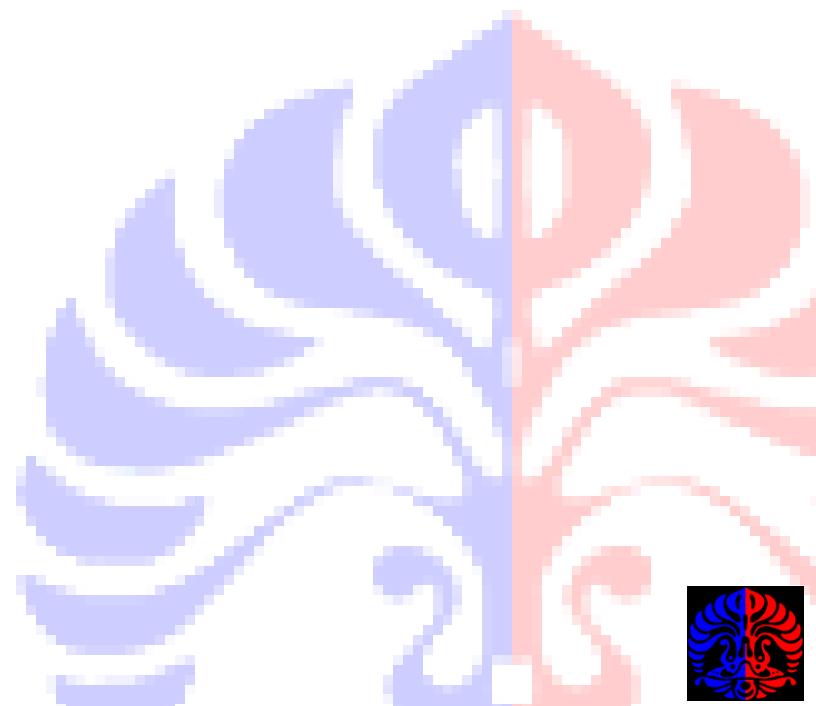
```

- Source Code lengkap bisa dilihat di:
- [http://telaga.cs.ui.ac.id/WebKuliah/IKI20100/resources/weiss.code/weiss/util/HashSet.java](http://telaga.cs.ui.ac.id/WebKuliah/IKI20100/resources/weiss/code/weiss/util/HashSet.java)
- [http://telaga.cs.ui.ac.id/WebKuliah/IKI20100/resources/weiss.code/weiss/util/HashMap.java](http://telaga.cs.ui.ac.id/WebKuliah/IKI20100/resources/weiss/code/weiss/util/HashMap.java)



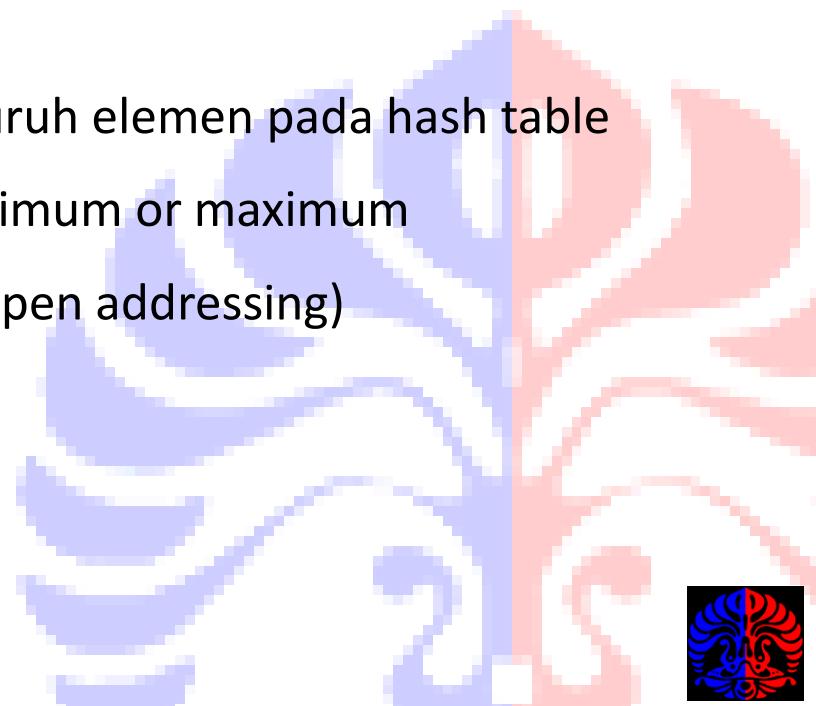
# Rangkuman

- Hash tables: array
- Hash function: Fungsi yang memetakan keys menjadi bilangan [0 ⇒ ukuran dari hash table)
- Collision resolution
  - Open hashing
    - Separate chaining
  - Closed hashing (Open addressing)
    - Linear probing
    - Quadratic probing
    - Double hashing
  - Primary Clustering, Secondary Clustering



# Rangkuman

- Advantage
  - running time
    - $O(1) + O(\text{collision resolution})$
  - Cocok untuk merepresentasikan data dengan frekuensi insert, delete dan search yang tinggi.
- Disadvantage
  - Sulit (tidak efficient) untuk mencetak seluruh elemen pada hash table
  - tidak efficient untuk mencari elemen minimum or maximum
  - tidak bisa di expand (untuk closed hash/open addressing)
  - ada pemborosan memory/space



# Referensi

- Bab 19 pada buku teks
- [http://www.cs.auckland.ac.nz/software/AlgAnim/hash\\_tables.htm](http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.htm)!
- <http://www.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm>

