

# IKI10400 • Struktur Data & Algoritma: Binary Search Tree

**Fakultas Ilmu Komputer • Universitas Indonesia**

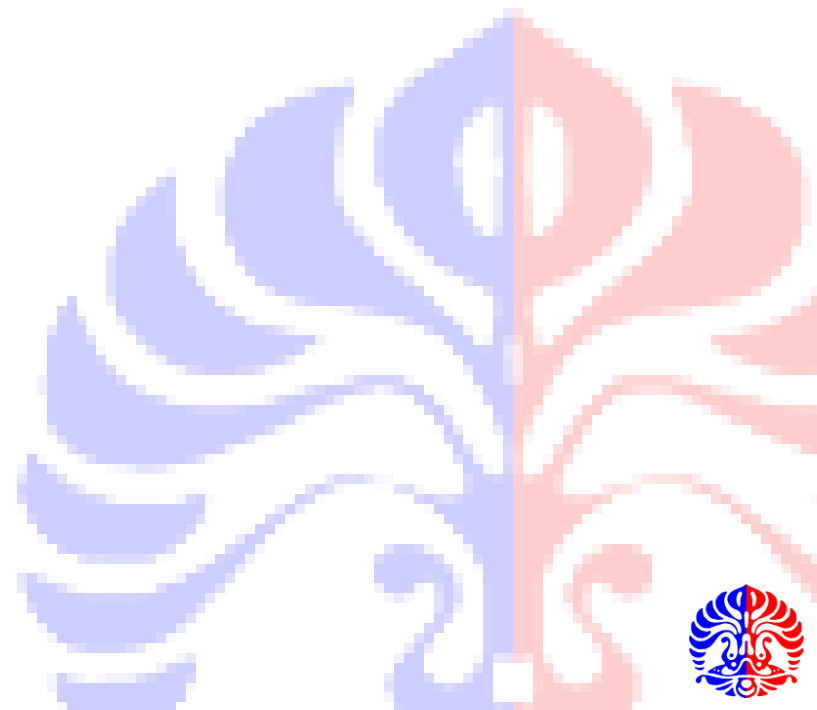
*Slide acknowledgments:*

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



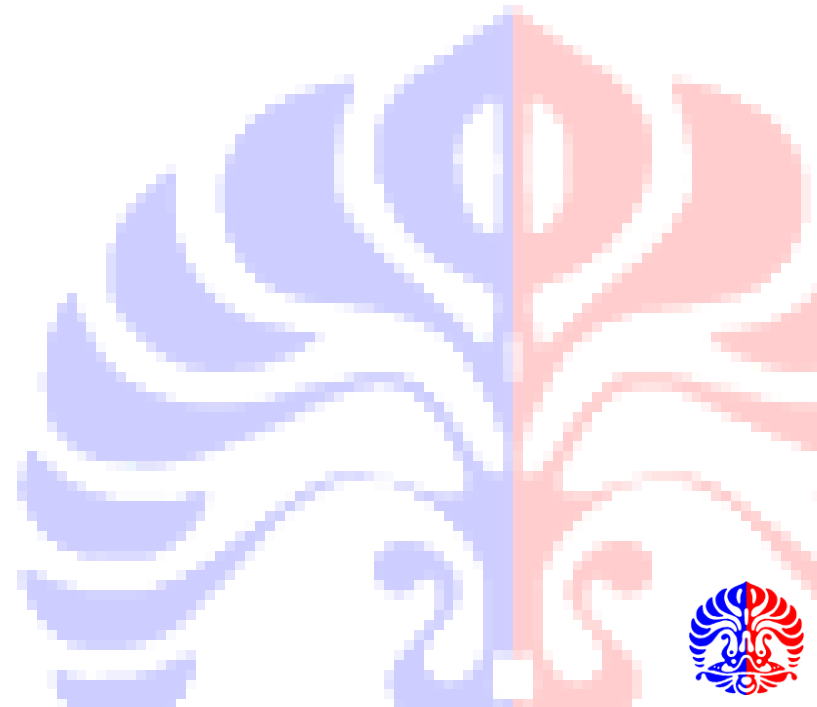
# Tujuan

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



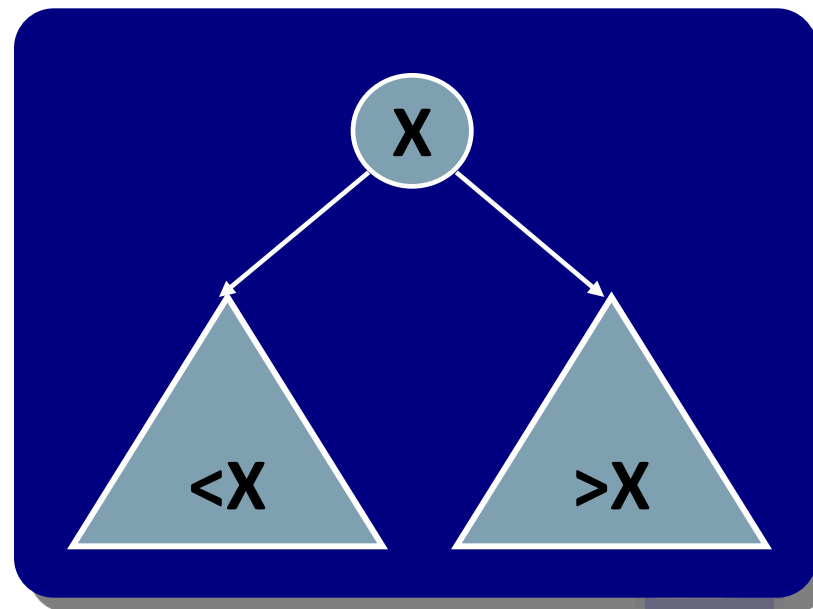
# Outline

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

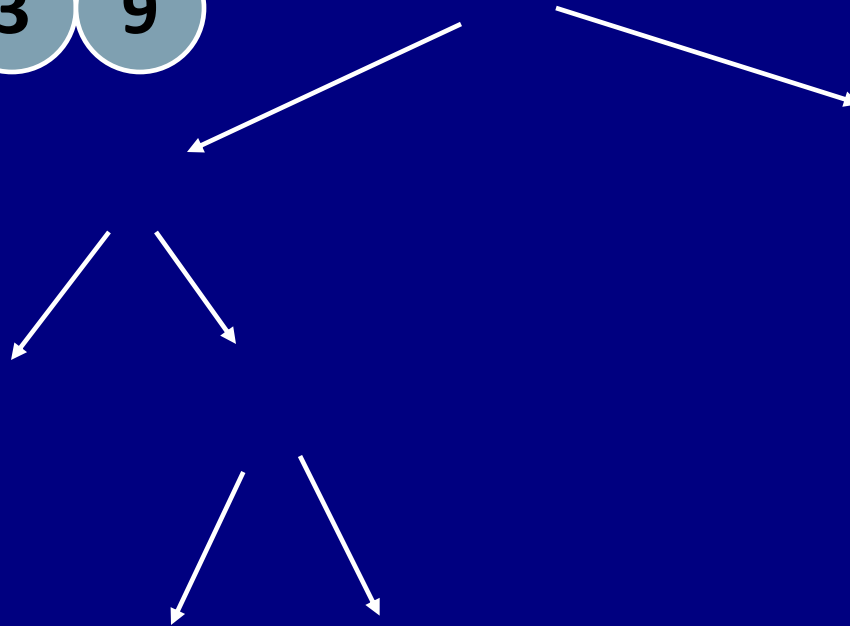
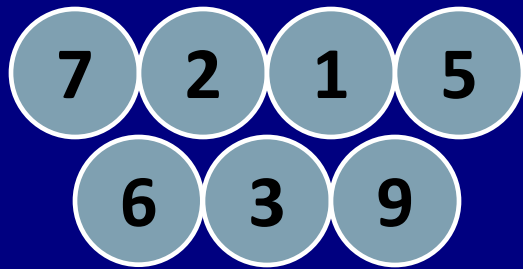


# Properties of Binary Search Tree

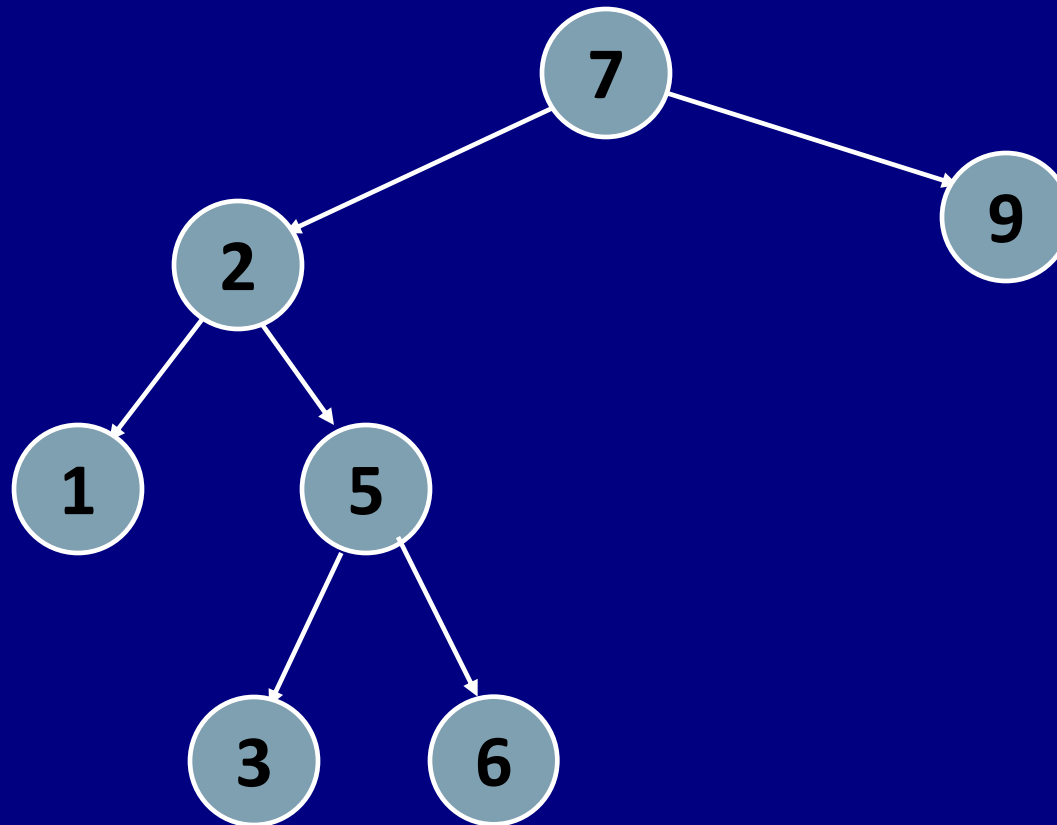
- Untuk setiap **node X** pada tree, nilai elemen pada subtree **sebelah kiri** selalu lebih kecil dari elemen **node X** dan nilai elemen pada subtree sebelah kanan selalu lebih besar dari elemen node X.
- Jadi object *elemen* harus **comparable**.



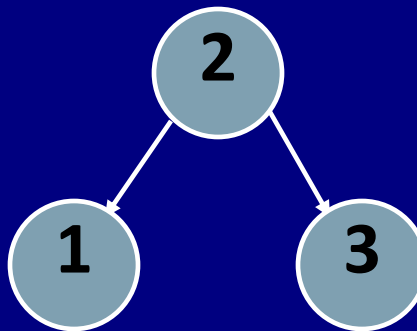
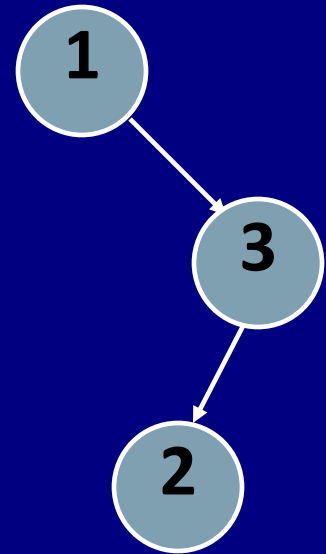
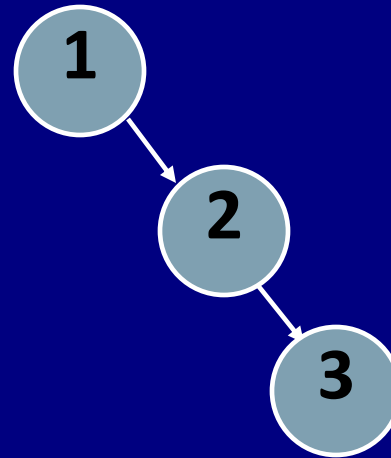
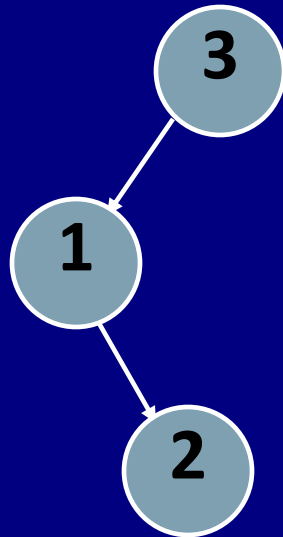
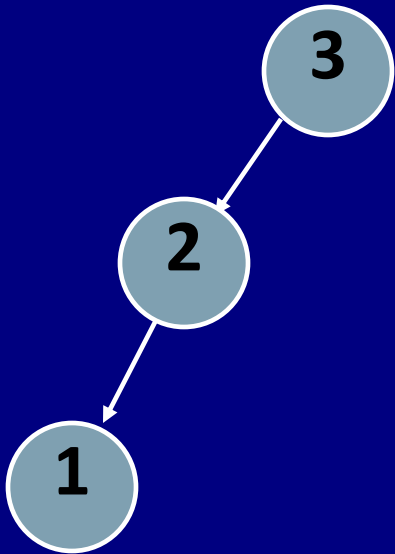
# Binary Search Tree



# Binary Search Tree

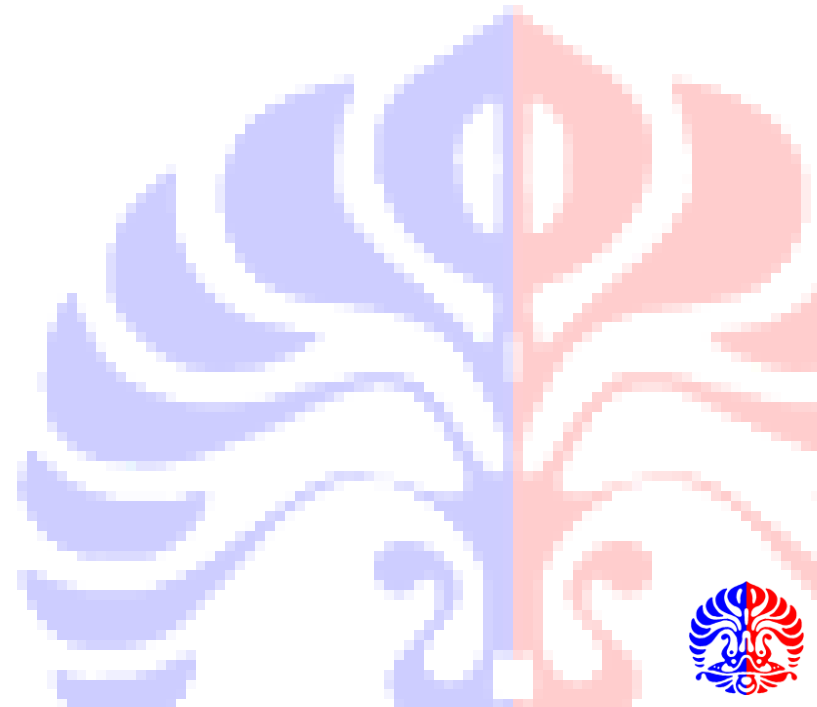


# Binary Search Tree



# Basic Operations

- insert
- findMin and findMax
- remove
- cetak terurut





# Print InOrder

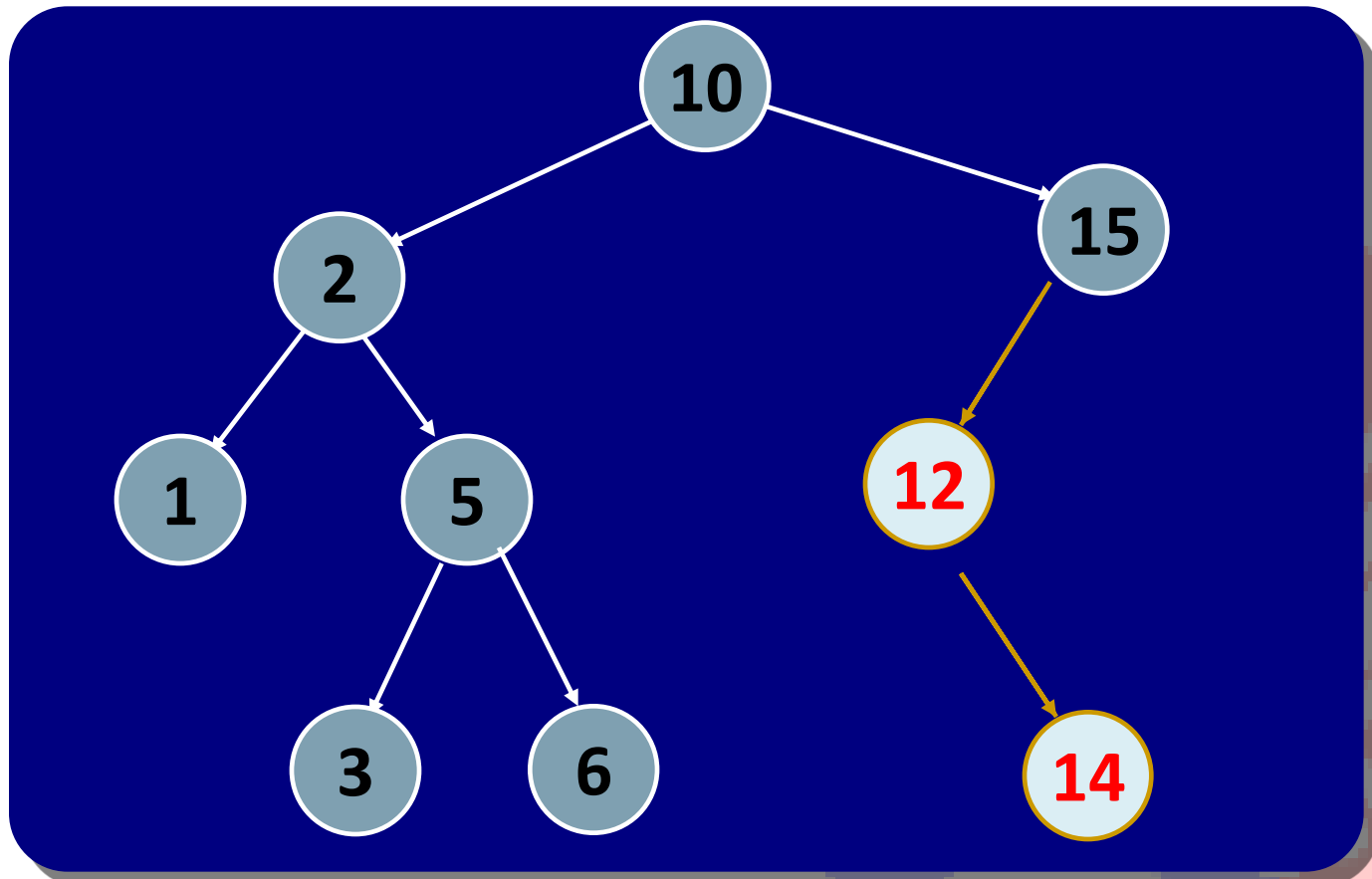
```
class BinaryNode {
    void printInOrder( )
    {
        if( left != null )
            left.printInOrder( );           // Left
        System.out.println( element );      // Node
        if( right != null )
            right.printInOrder( );          // Right
    }
}

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:

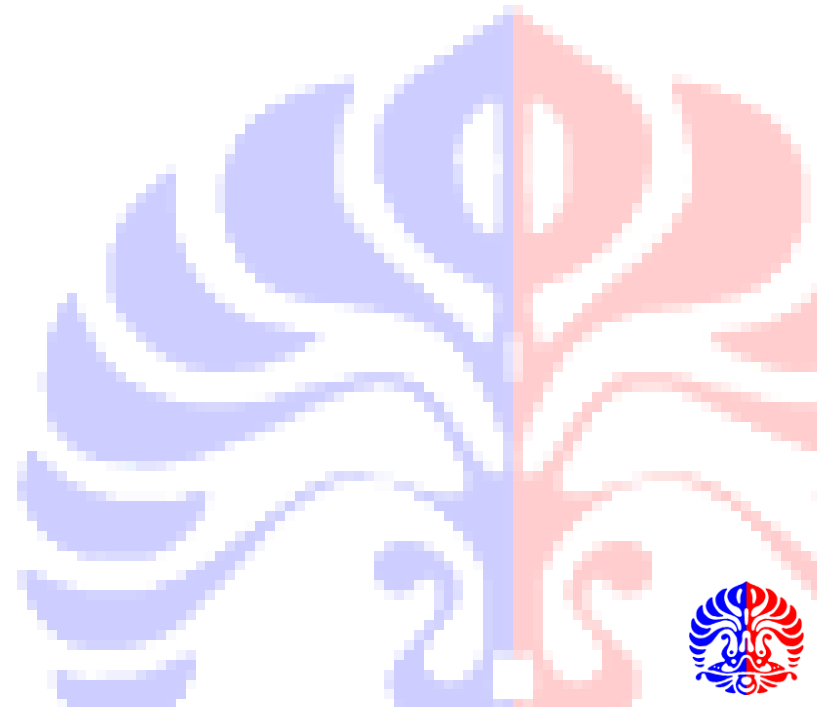
```
BinaryNode findMin (BinaryNode t)
{
    if (t == null) throw exception;

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



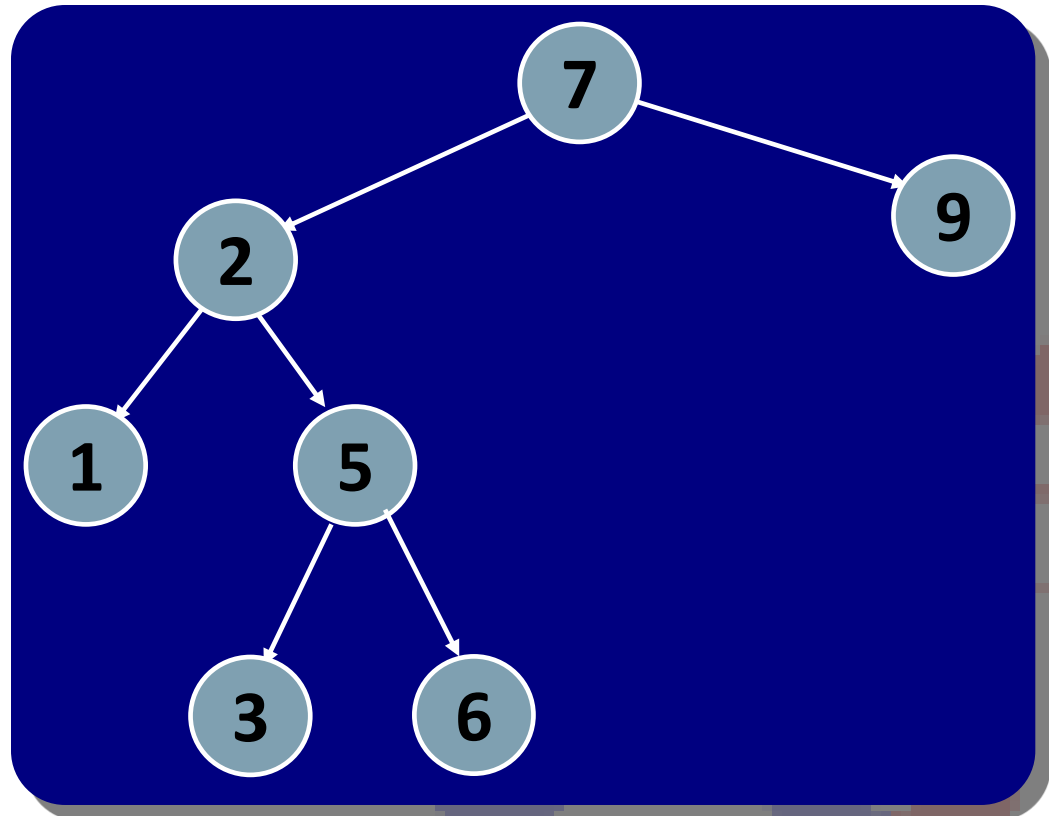
# FindMax

- 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?



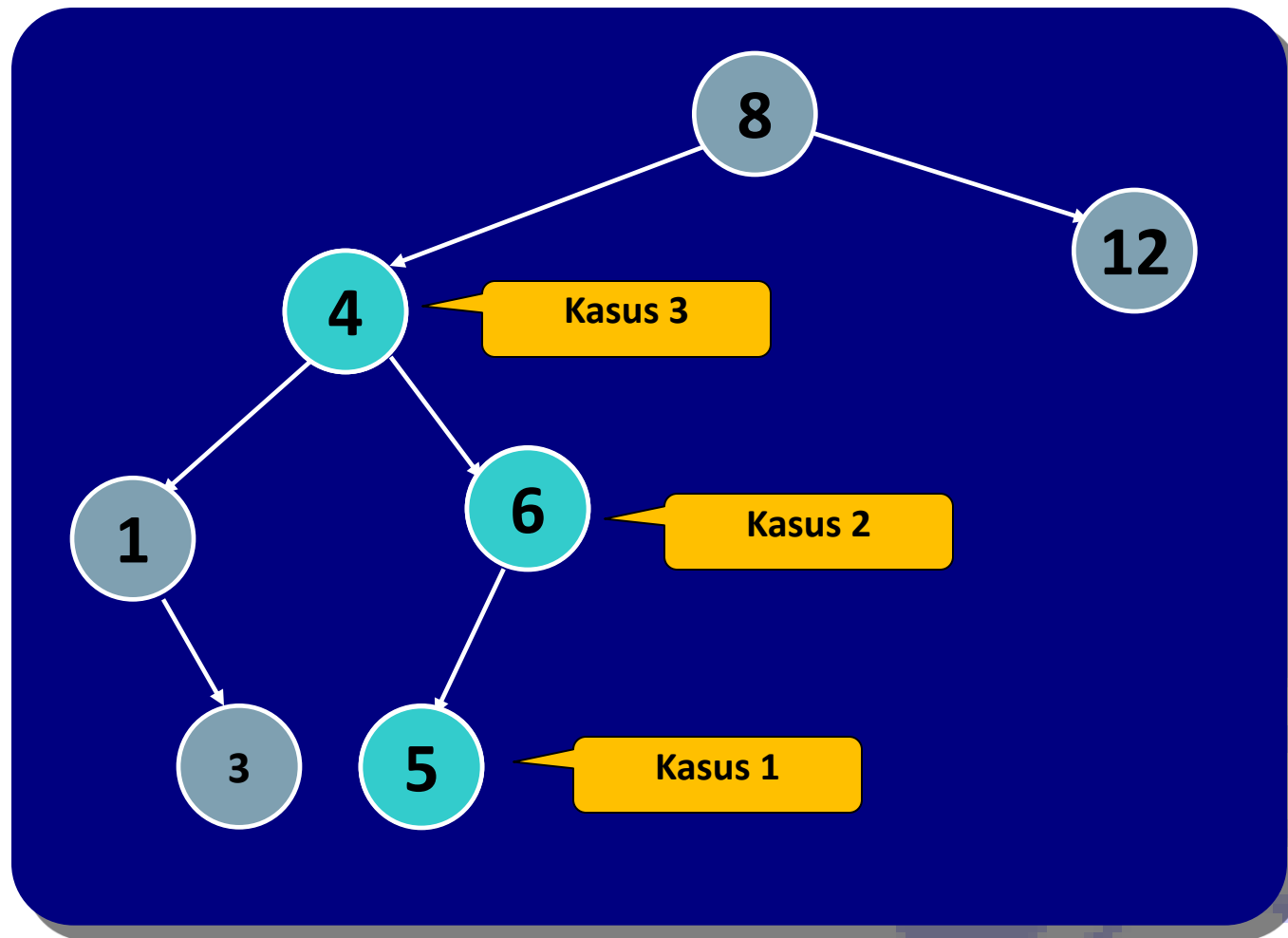
# Remove

- **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.....

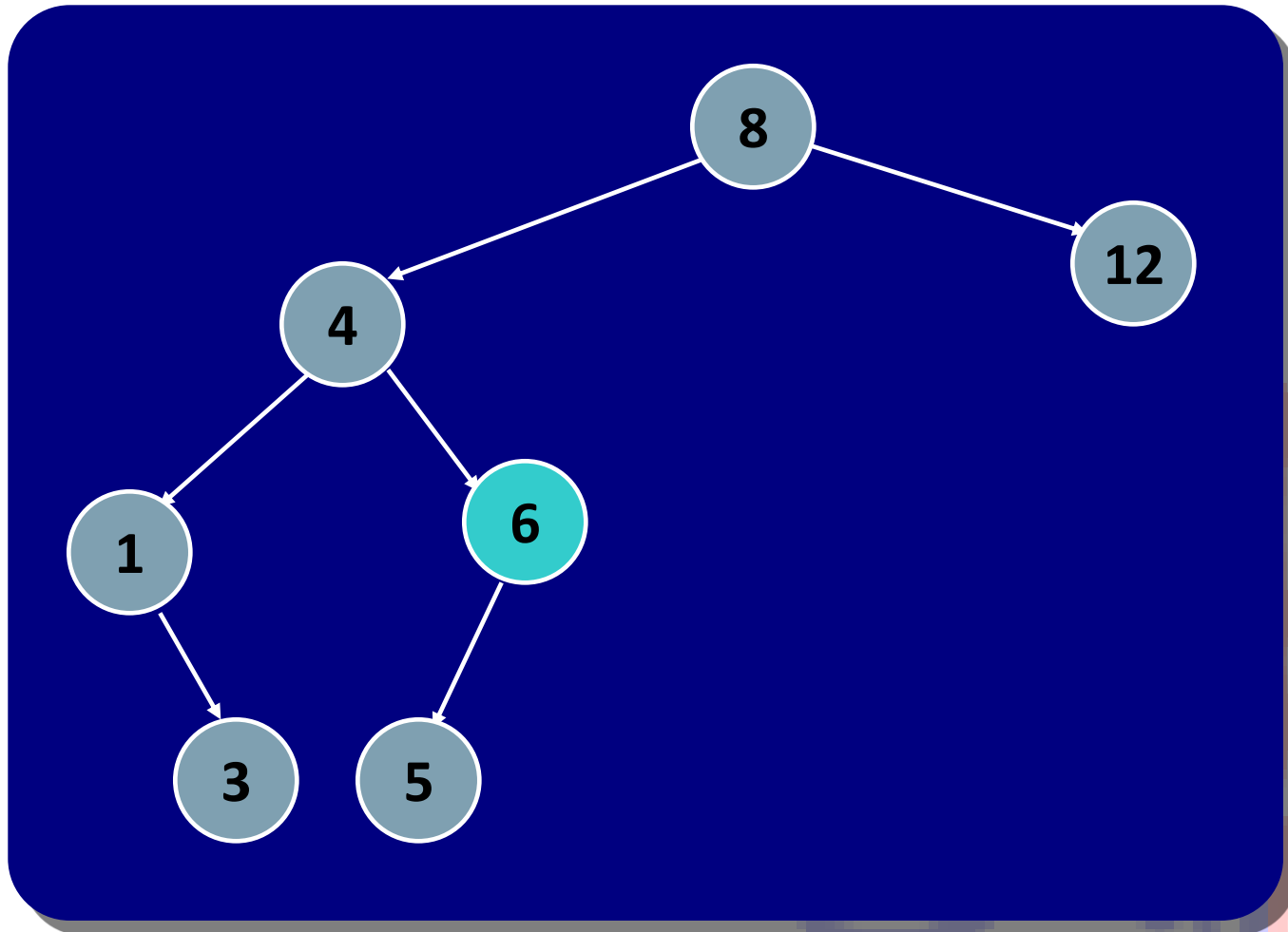




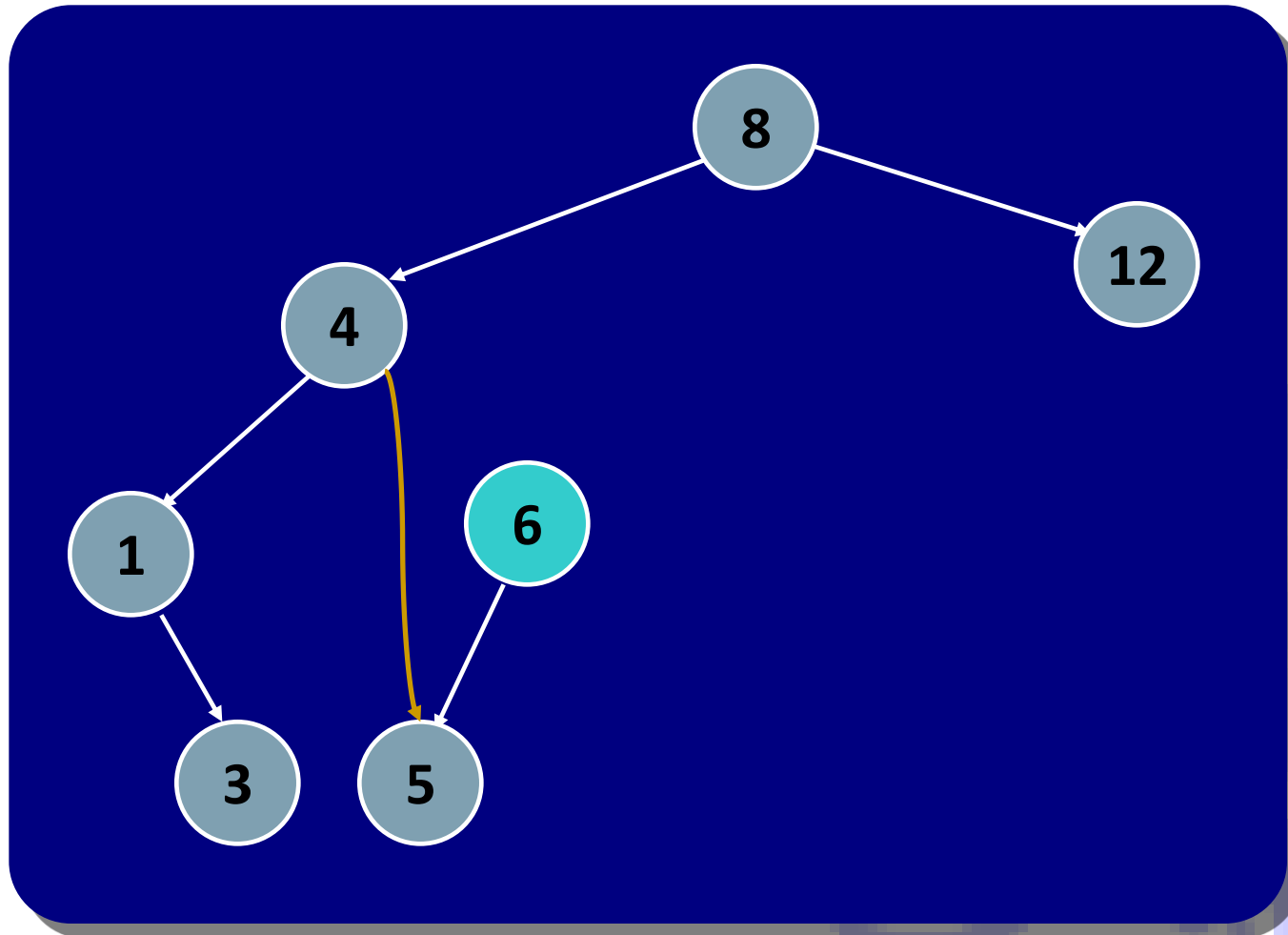
# Remove



# 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.

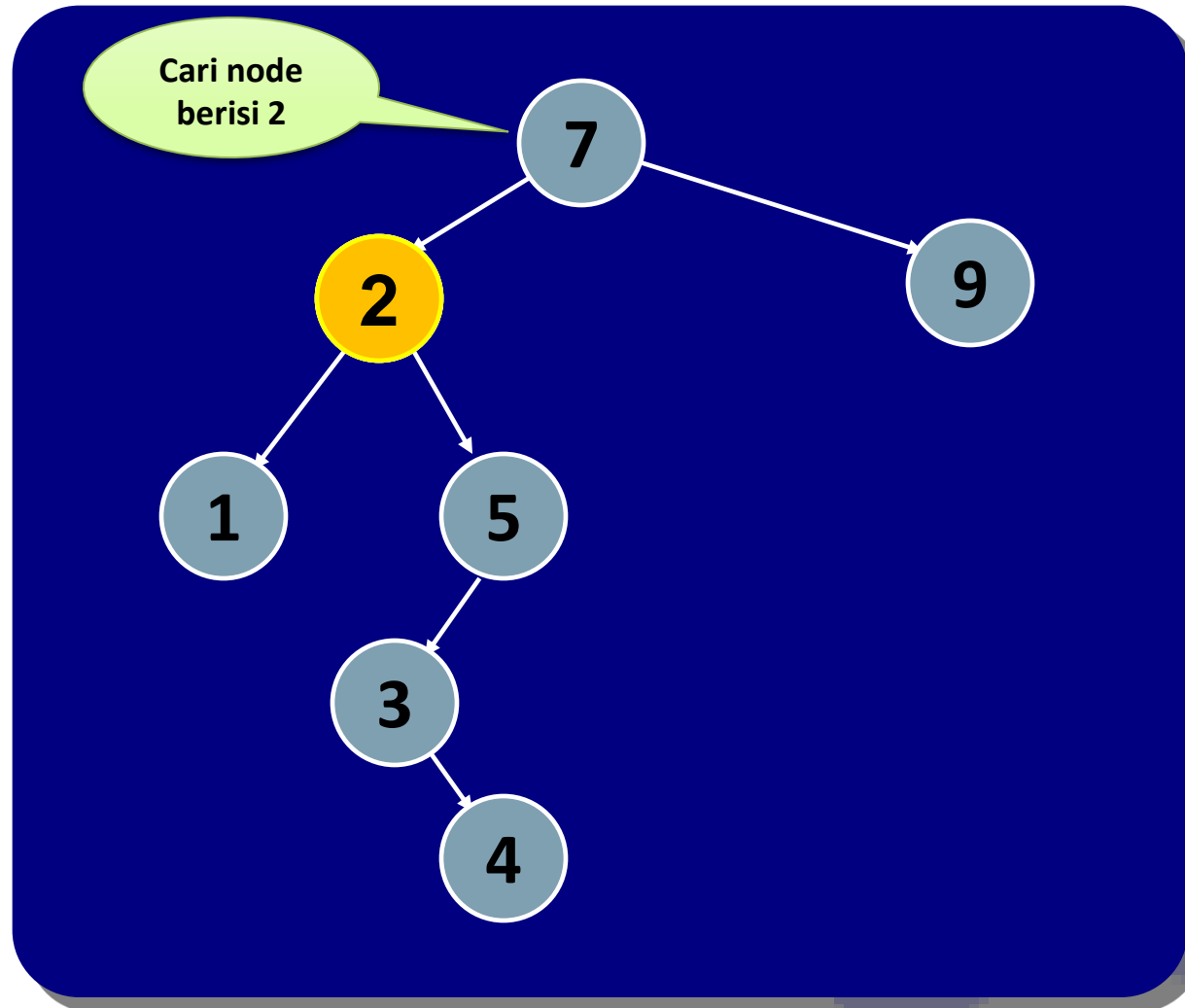
[Alternatif: dengan kaidah **Predecesor Inorder**,

2. Gantikan posisinya dengan:

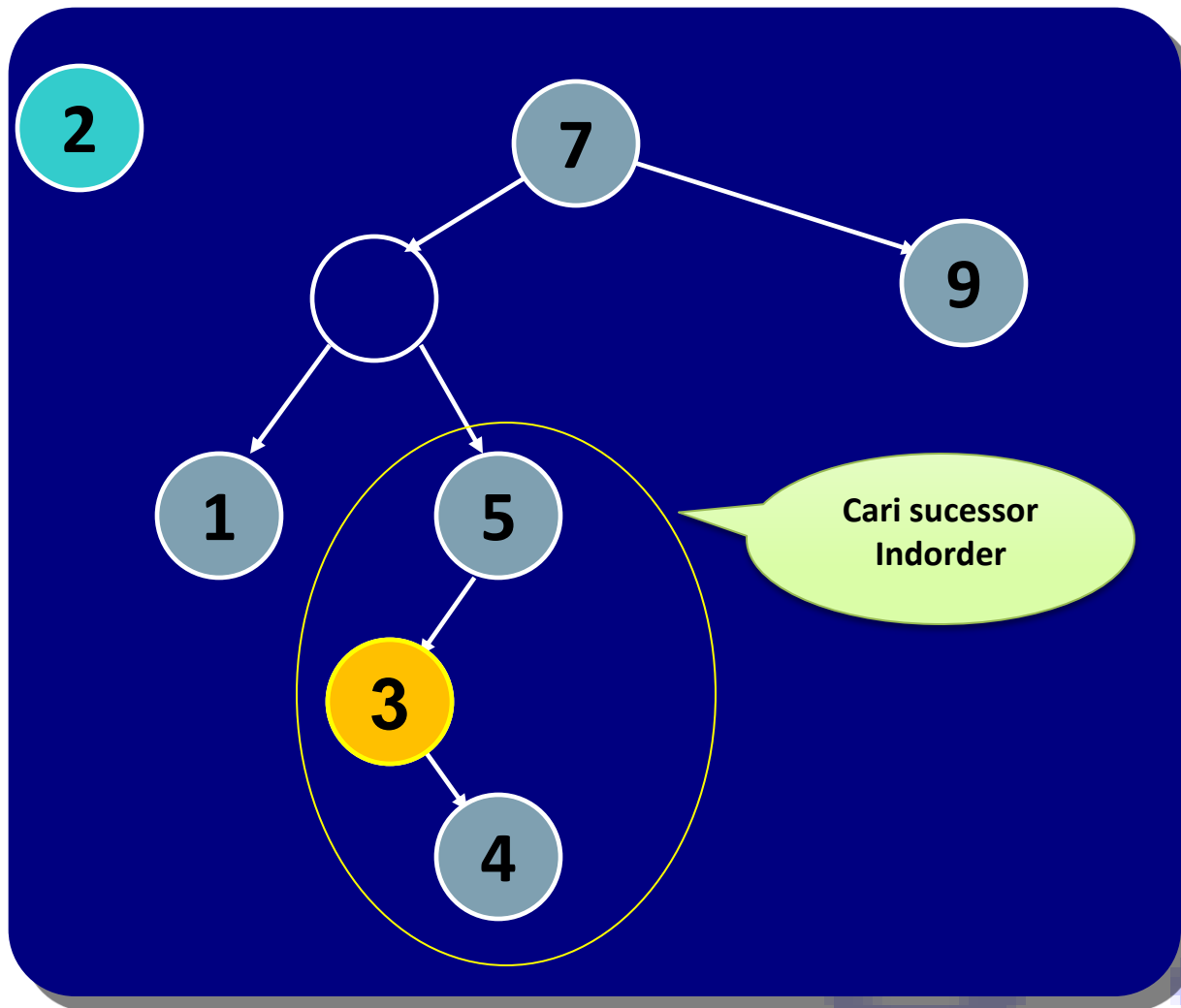
- **Predecesor Inorder**, node terbesar dari sub tree kiri, dilanjutkan dengan melakukan removeMax di subtree kiri.]



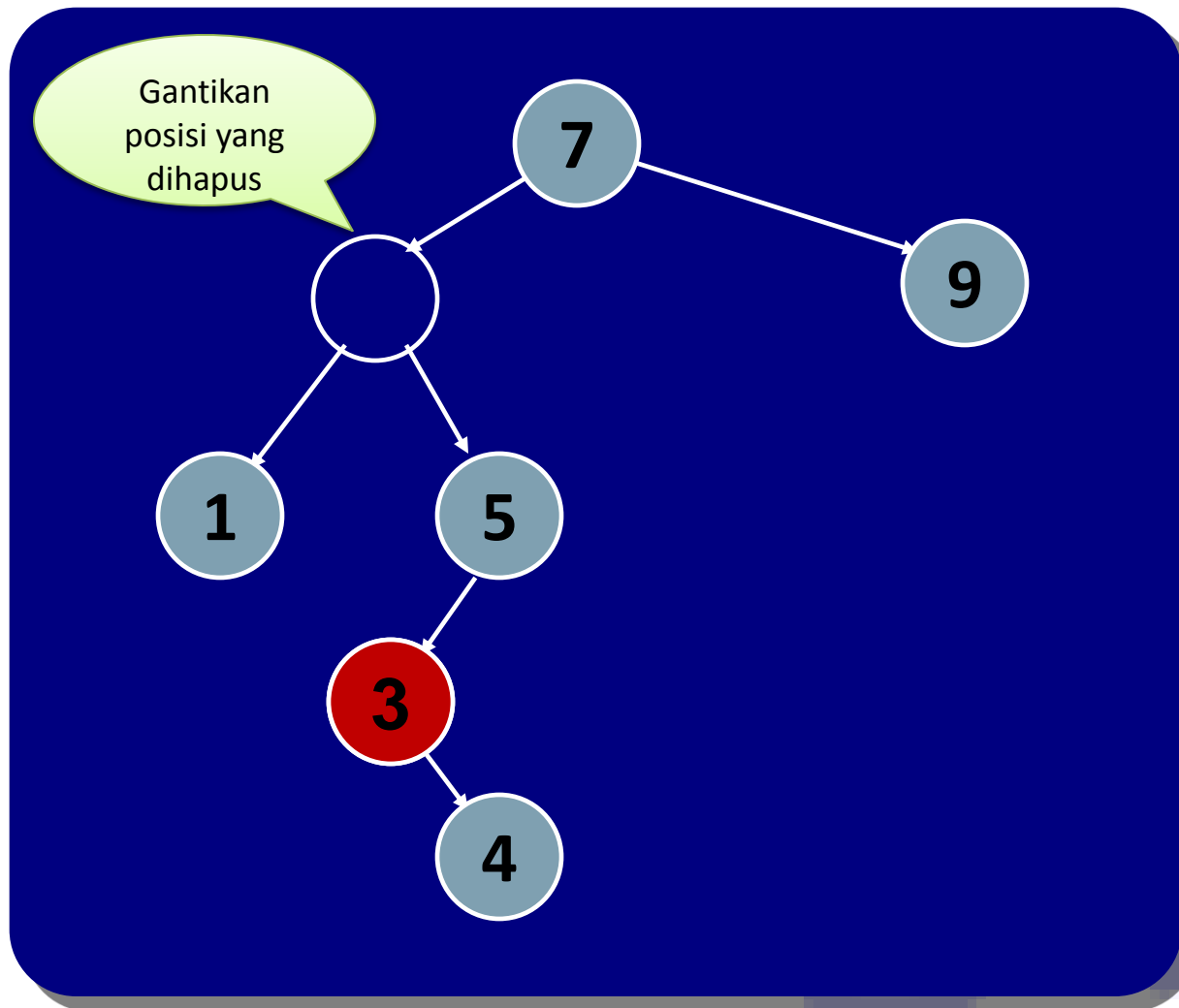
# Removing 2 (Sucessor Inorder)



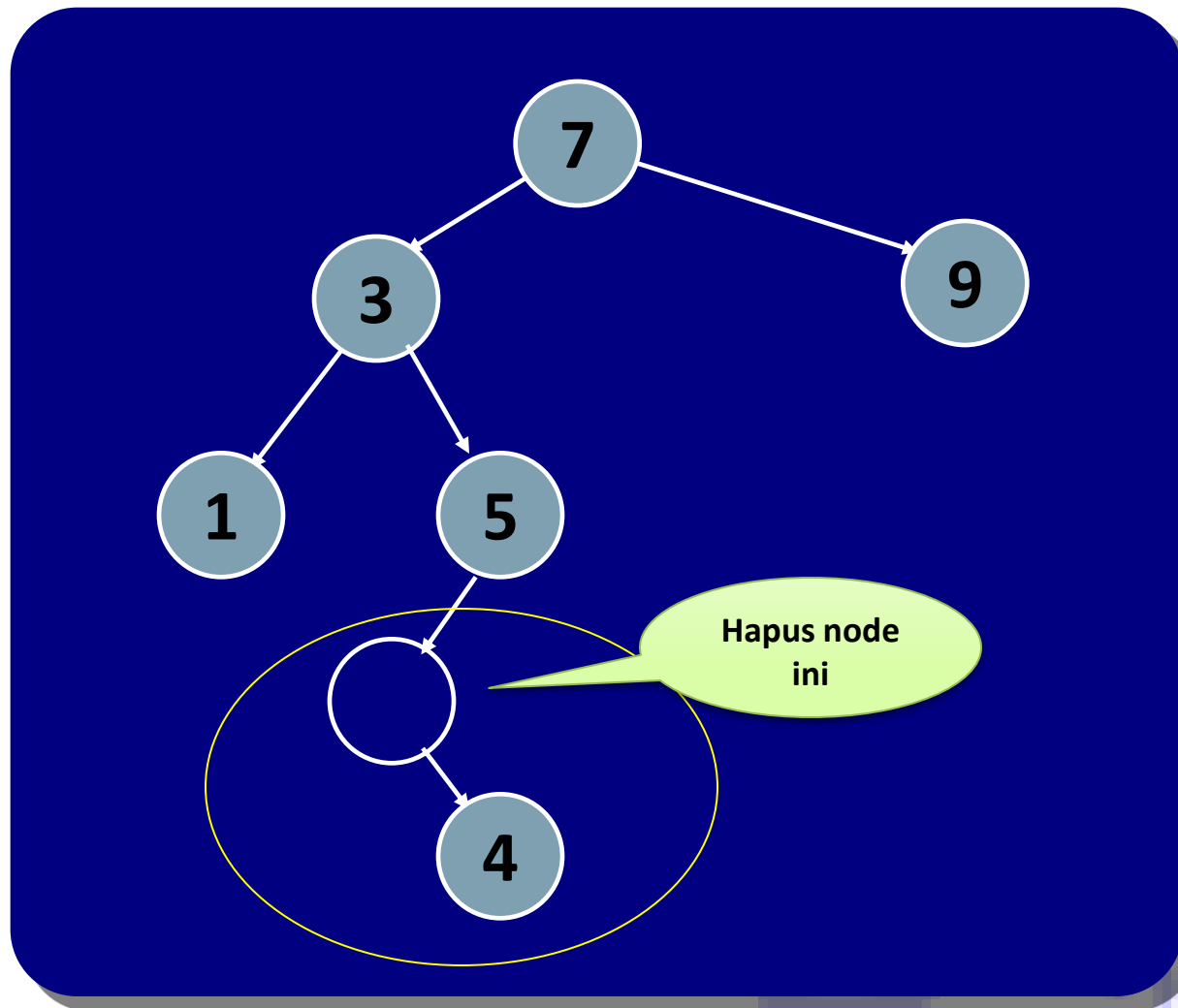
# Removing 2 (Sucessor Inorder)



# Removing 2 (Sucessor Inorder)

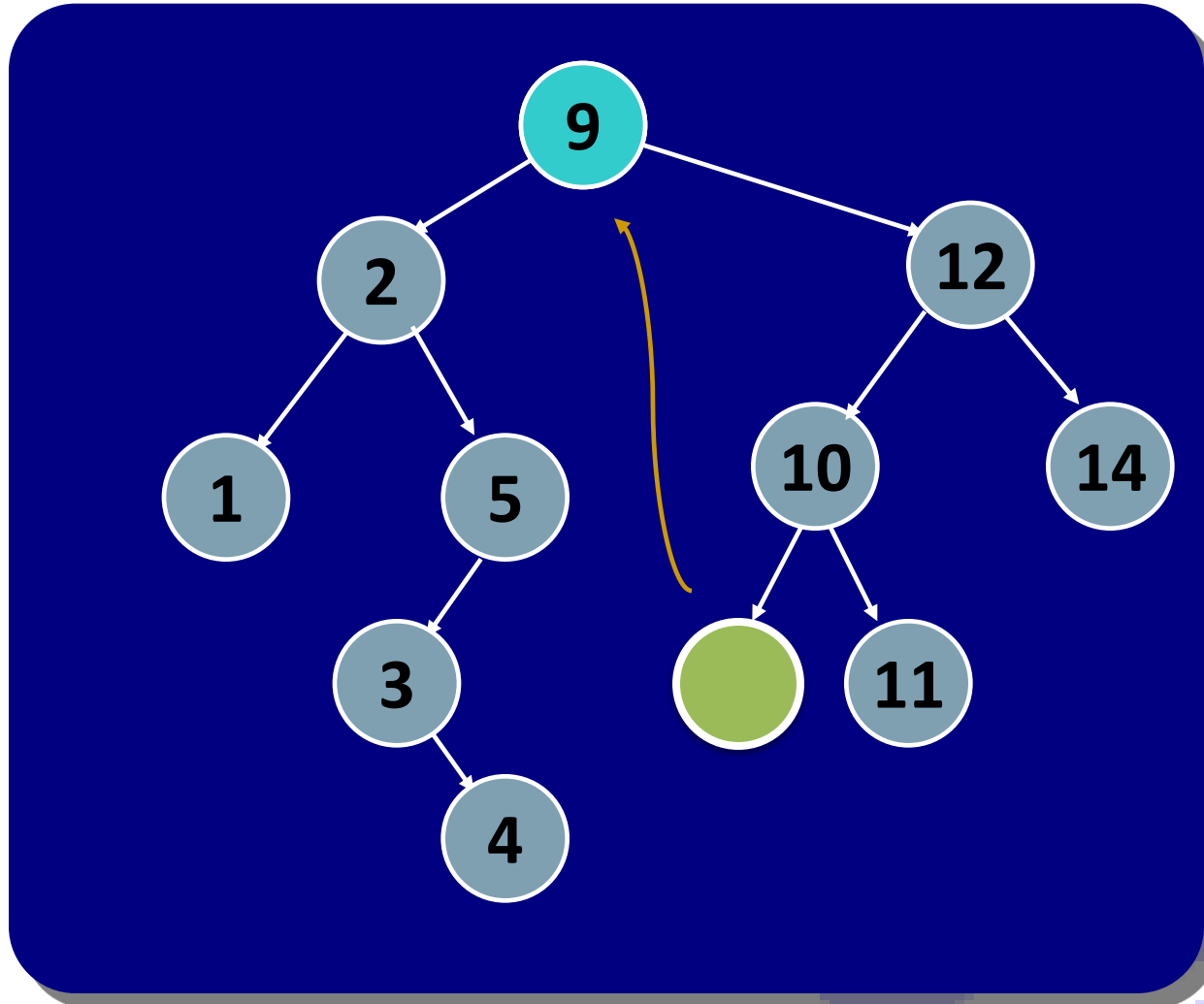


# After 2 deleted





# 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;
    }
}
```



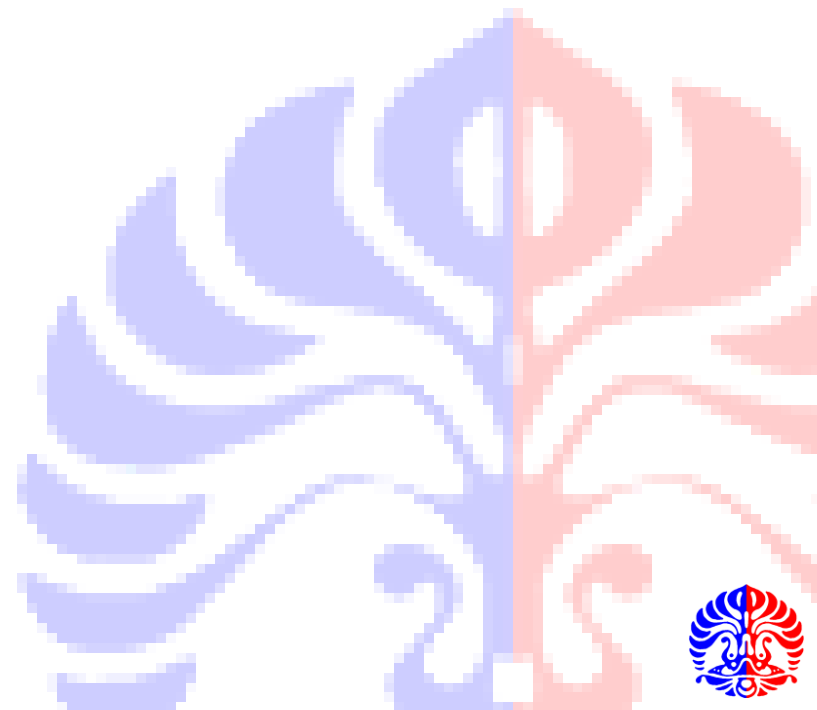
# Remove

```
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;  
}
```

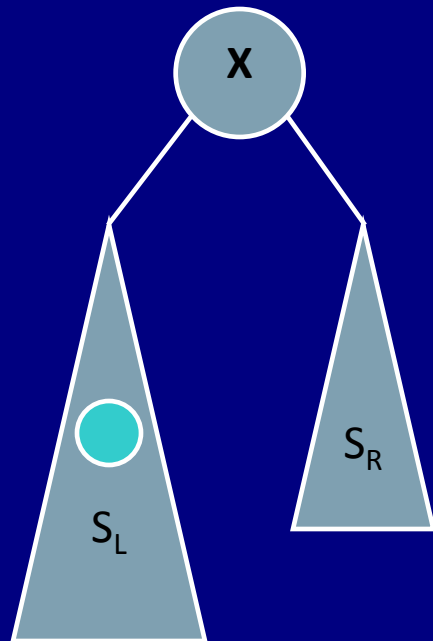


# removeMax

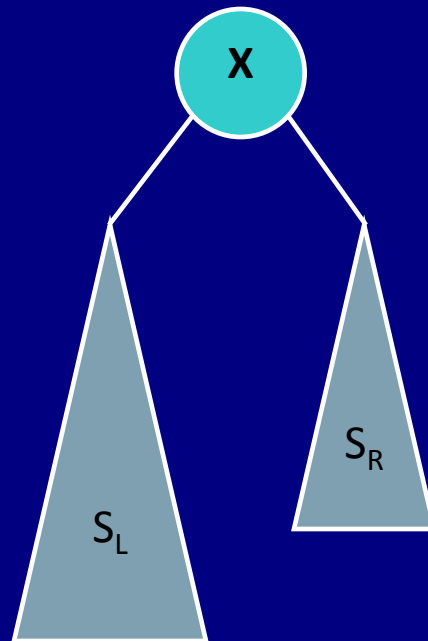
■ code?



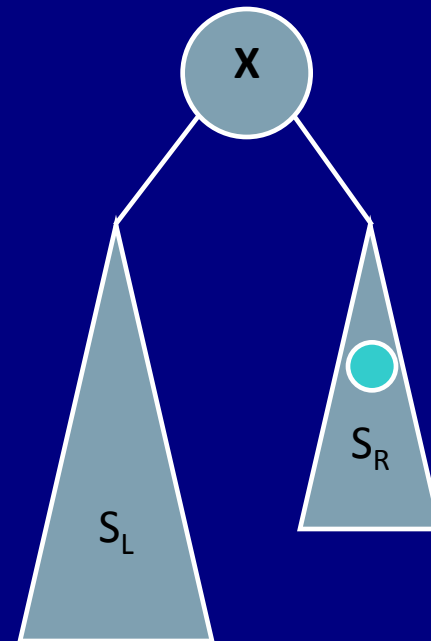
# Find k-th element



$$k < S_L + 1$$



$$k == S_L + 1$$



$$k > S_L + 1$$



# Find k-th element

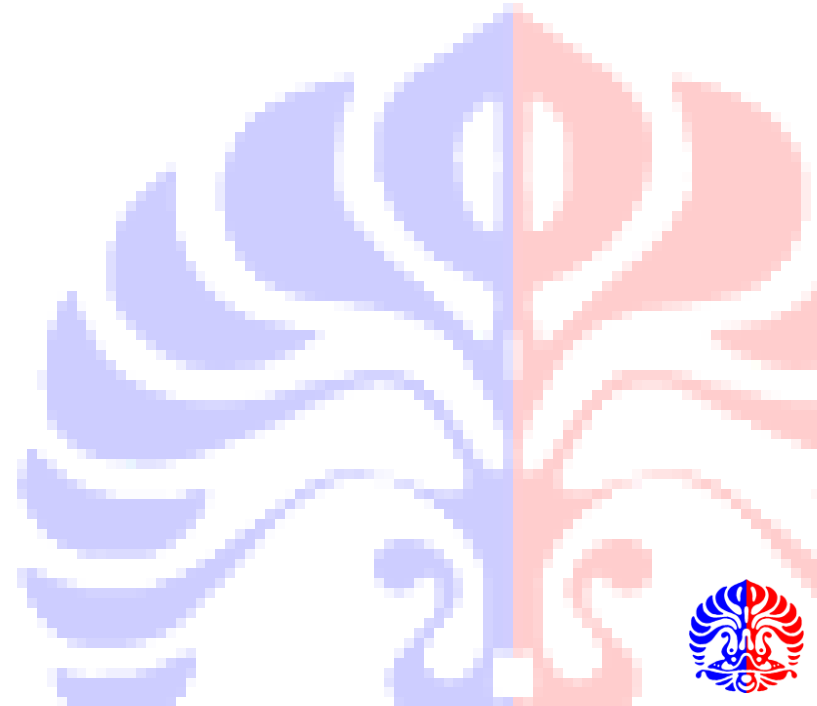
```
BinaryNode findKth(int k, BinaryNode t)
{
    if (t == null) throw exception;
    int leftSize = (t.left != null) ?
        t.left.size : 0;

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



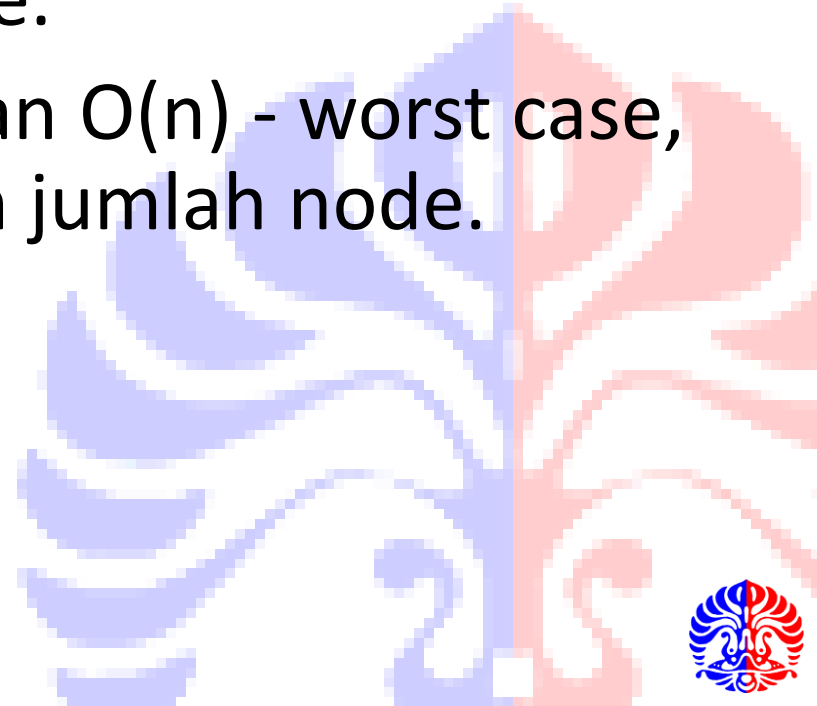
# Analysis

- 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:

- Se jauh 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

