Tree (1)

BAB 7

Tree

Deskripsi Binary Tree

Dasar Tree

Deklarasi Tree

Metode Traversal

AVL Tree

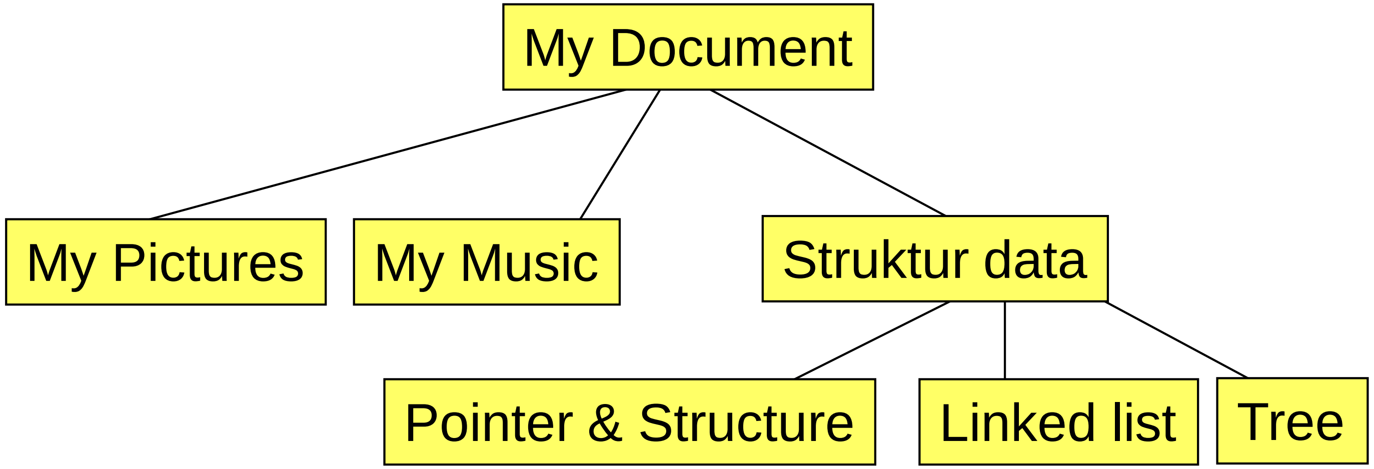
Outline

1. Apakah Tree Structure itu ?
2. Binary Tree & implementasinya
3. Tree Traversal

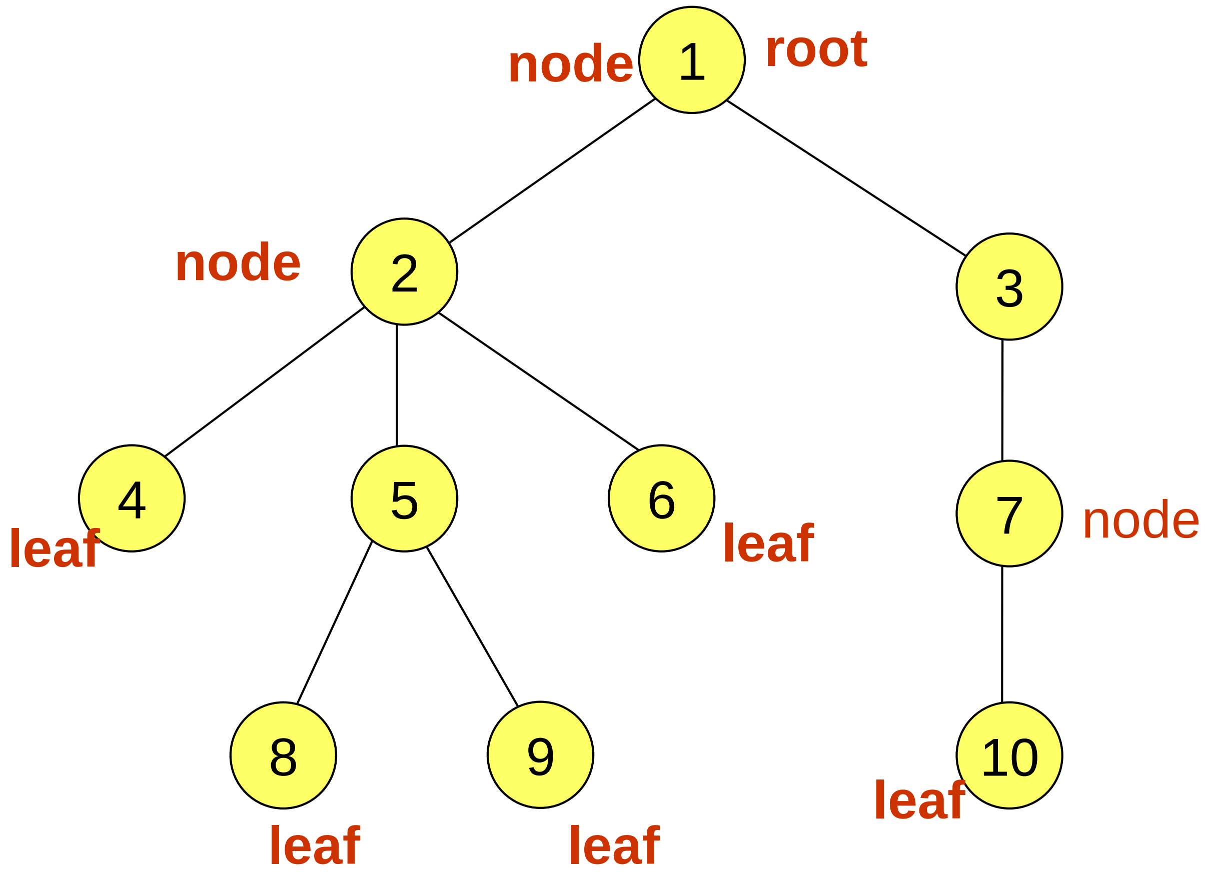
Apakah Tree Structure itu ?

Struktur data yang menunjukkan hubungan bertingkat (memiliki hierarkhi)

Contoh: direktori/folder pada windows atau linux



Nama komponen pada Tree



Hubungan antar komponen

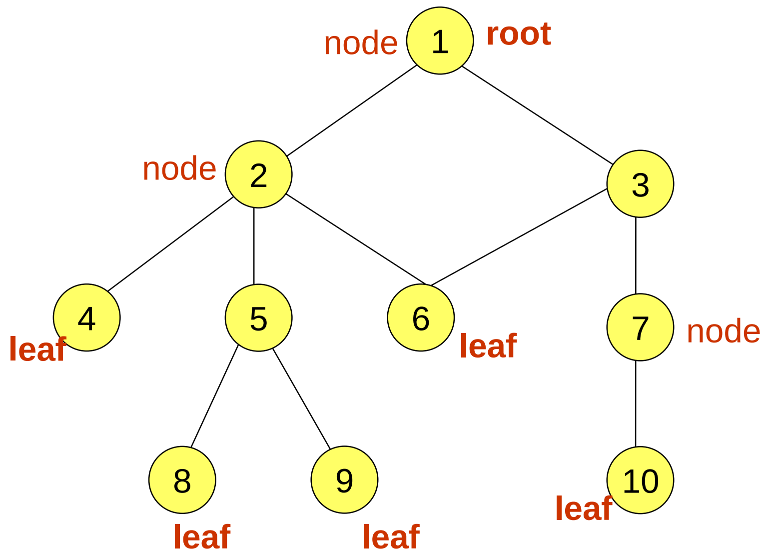
Hubungan antar elemen: parent-child, father-son, mother-daughter

Nama node: nama(angka) yang dipakai untuk membedakan sebuah node dengan node yang lain. Dalam kuliah ini adalah angka yang tertulis dalam lingkaran.

Label: nilai yang diingat oleh sebuah node

Tree vs Graph

- Tree: setiap node kecuali root hanya memiliki sebuah parent
- Graph: dapat memiliki lebih dari sebuah parent



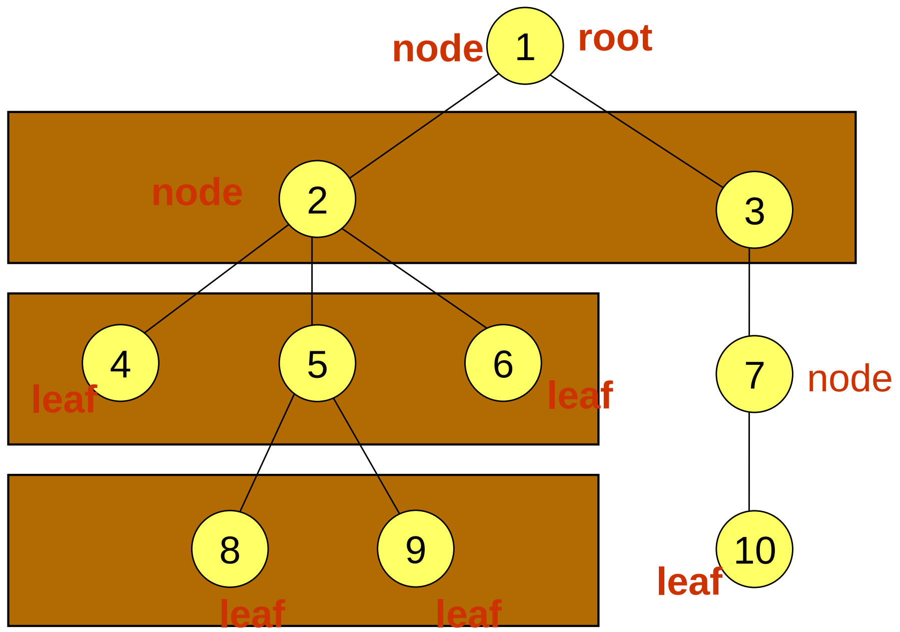
Contoh graph

Hubungan antar komponen

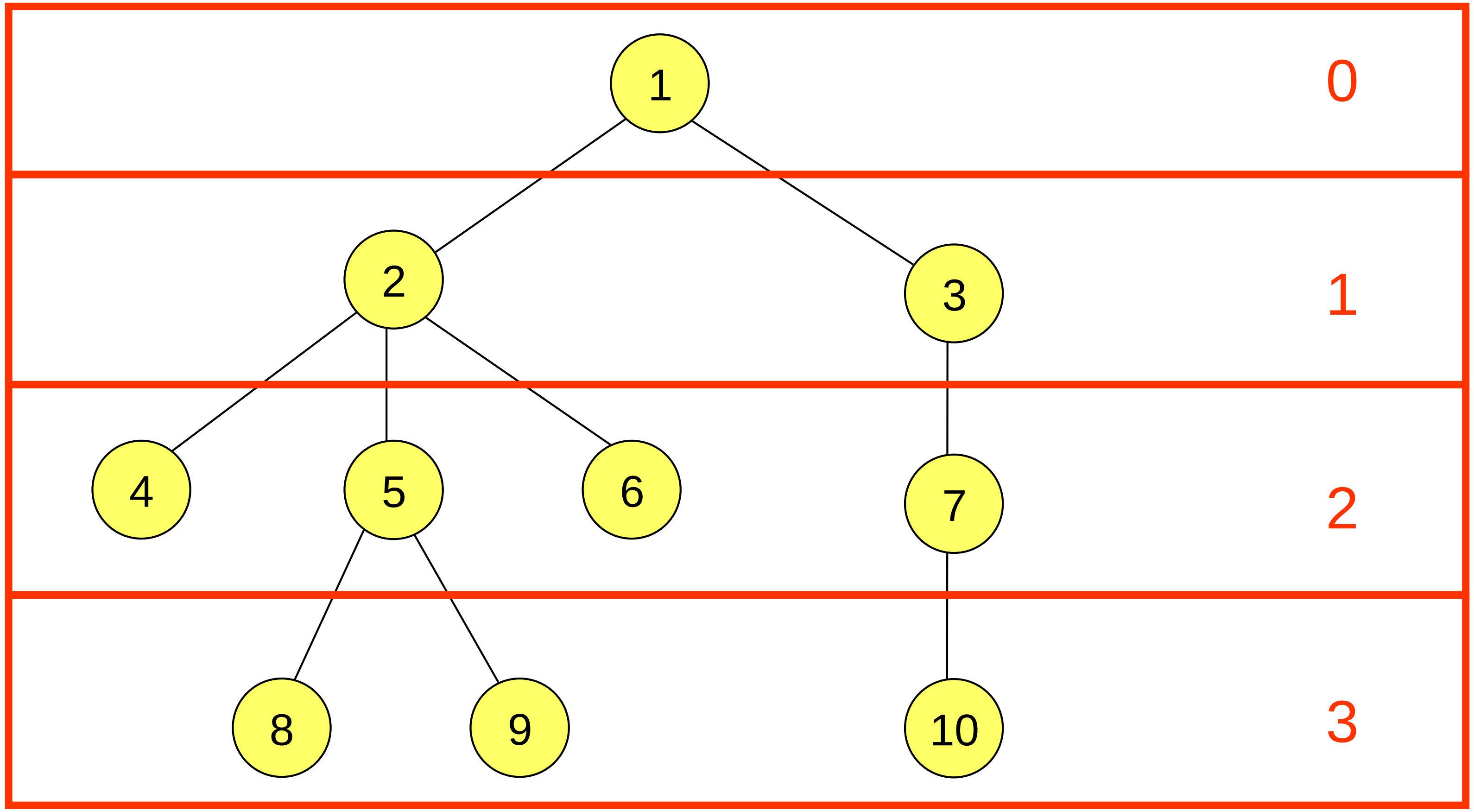
sibling : node-node yang memiliki parent yang sama

Ancestor dari node x : node yang ditemukan, ketika menyusuri tree ke atas dari node x

Descendant dari node x : node yang ditemukan ketika menyusuri tree ke bawah dari node x



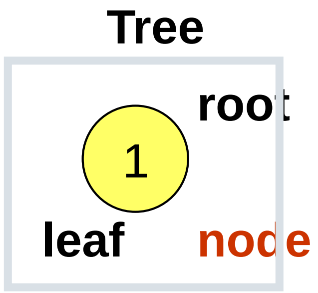
Level



Definisi TREE

Sebuah tree didefinisikan sebagai struktur y ang dibentuk secara recursive oleh kedua rule berikut:

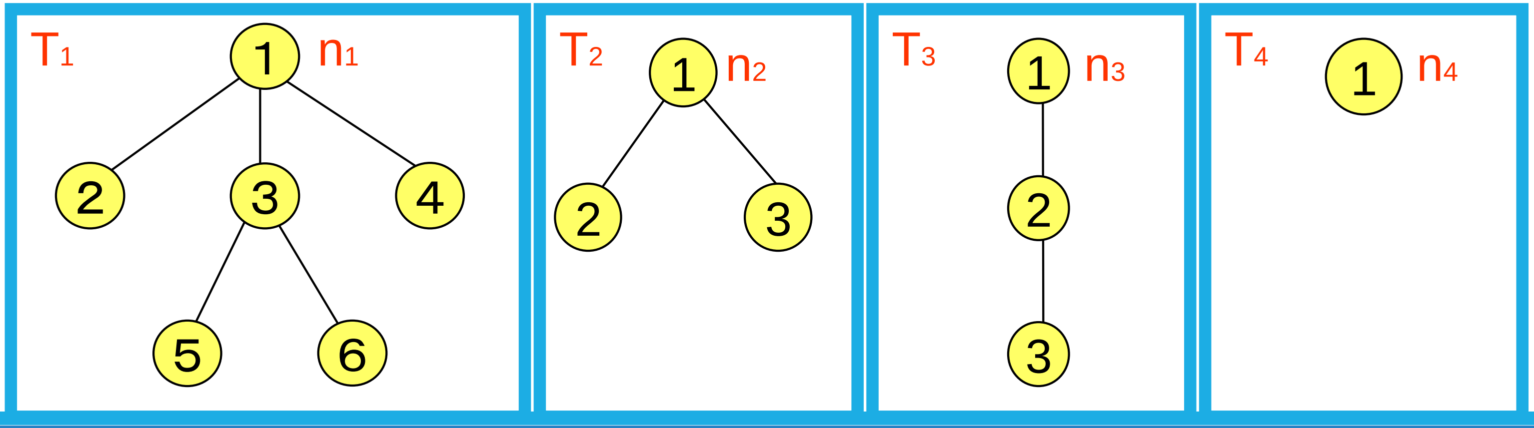
1. Sebuah node adalah sebuah tree. Node satu-satunya pada tree ini berfungsi sebagai root maupun leaf.
2. Dari k buah tree $T_1 \sim T_k$, dan masing-masing memiliki root $n_1 \sim n_k$.Jika node n adalah parent dari node$n_1 \sim n_k$, akan diperoleh sebuah tree baru T yang memiliki root n. Dalam kondisi ini, tree $T_1 \sim T_k$menjadi sub-tree dari tree T. Root dari sub-tree $n_1 \sim n_k$adalah child dari node n .



Definisi TREE

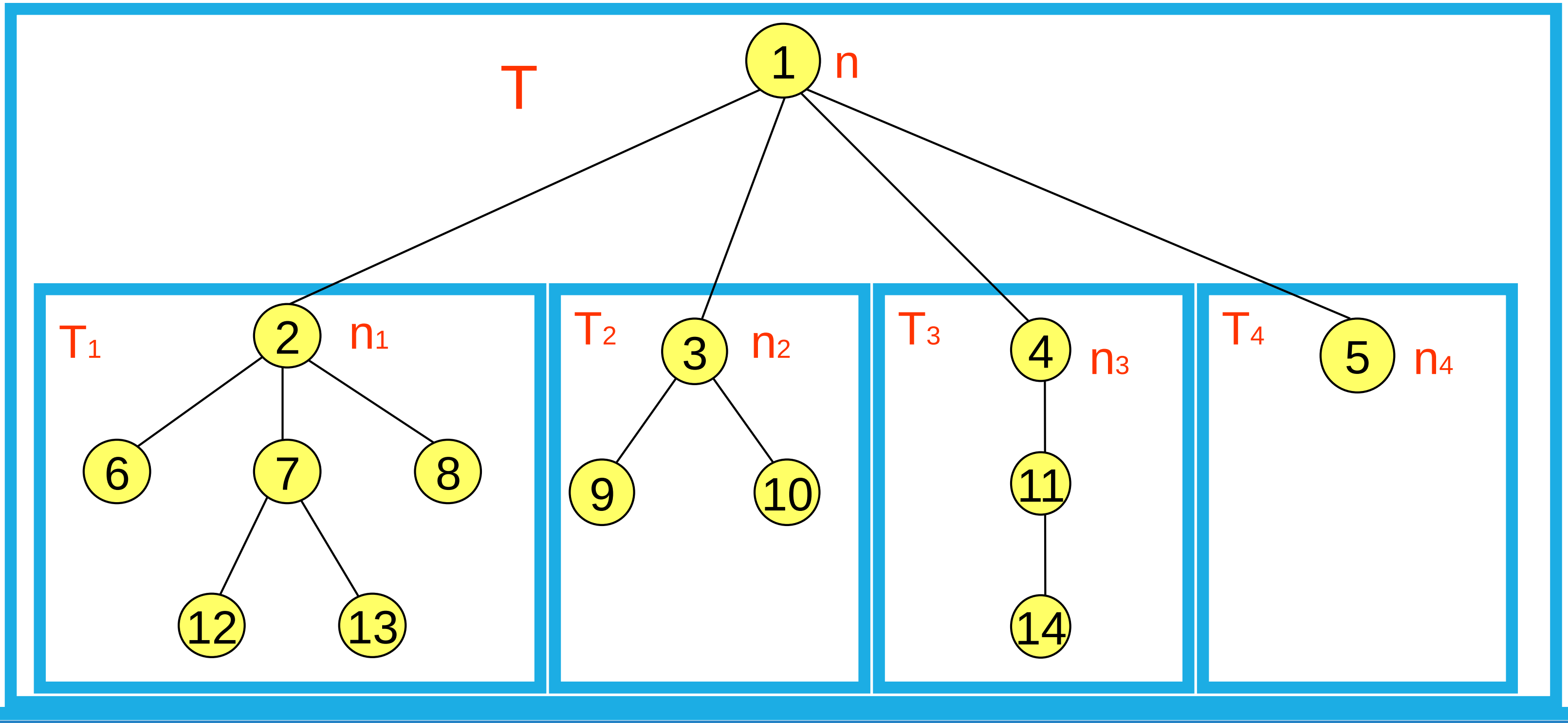
Sebuah tree didefinisikan sebagai struktur y ang dibentuk secara recursive oleh kedua rule berikut:

1. Sebuah node adalah sebuah tree. Node satu-satunya pada tree ini berfungsi sebagai root maupun leaf.
2. Dari k buah tree $T_1 \sim T_k$, dan masing-masing memiliki root $n_1 \sim n_k$.Jika node n adalah parent dari node$n_1 \sim n_k$, akan diperoleh sebuah tree baru T yang memiliki root n. Dalam kondisi ini, tree $T_1 \sim T_k$menjadi sub-tree dari tree T. Root dari sub-tree $n_1 \sim n_k$adalah child dari node n .



Definisi TREE

2. Dari k buah tree $T_1 \sim T_k$, dan masing-masing memiliki root $n_1 \sim n_k$.Jika node n adalah parent dari noden$_1 \sim n_k$, akan diperoleh sebuah tree baru T yang memiliki root n. Dalam kondisi ini, tree $T_1 \sim T_k$menjadi sub-tree dari tree T. Root dari sub-tree $n_1 \sim n_k$adalah child dari node n .



Ordered vs Unordered tree

Ordered tree

- Antar sibling terdapat urutan “usia”
- Node yang paling kiri berusia paling tua (sulung), sedangkan node yang paling kanan berusia paling muda (bungsu)
- Posisi node diatur atas urutan tertentu

Unordered tree

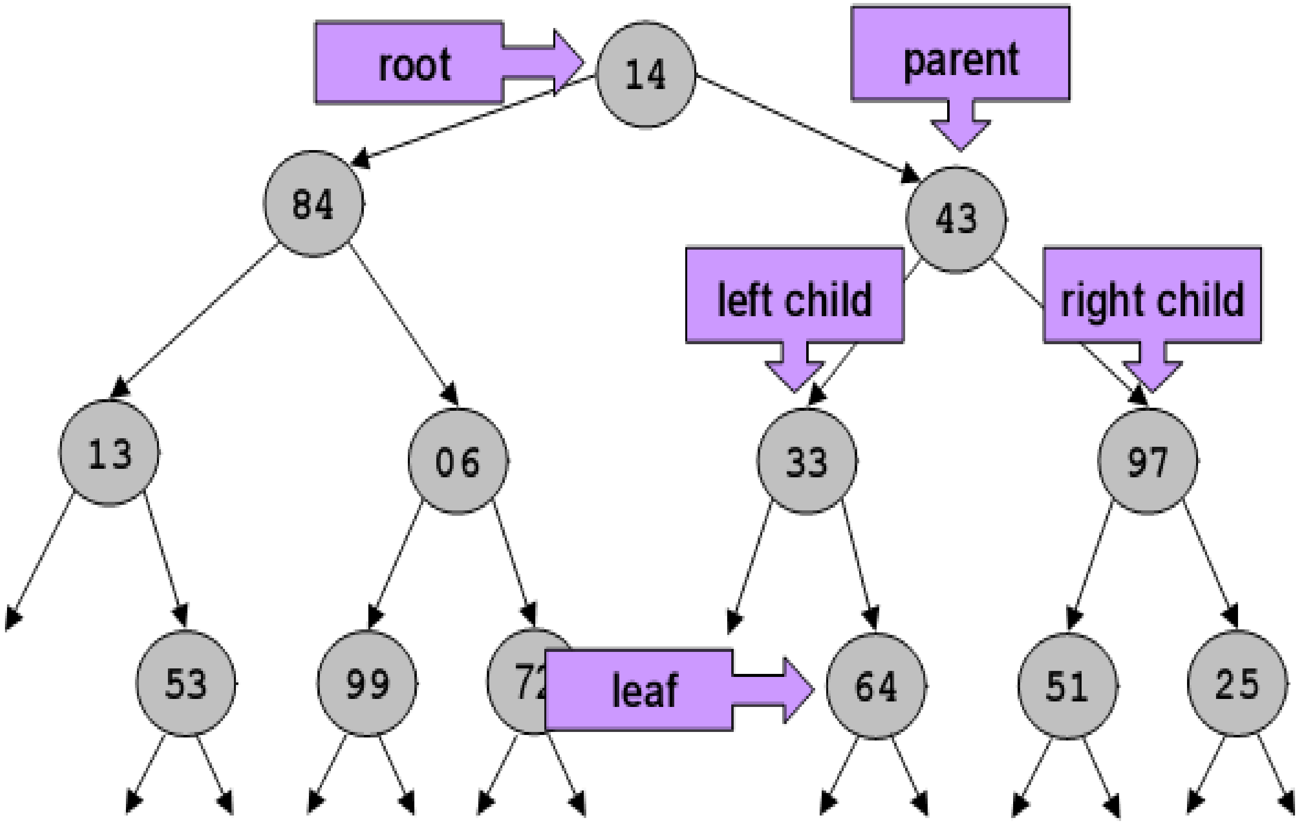
- Antar sibling tidak terdapat urutan tertentu

Jenis Tree

Binary Tree

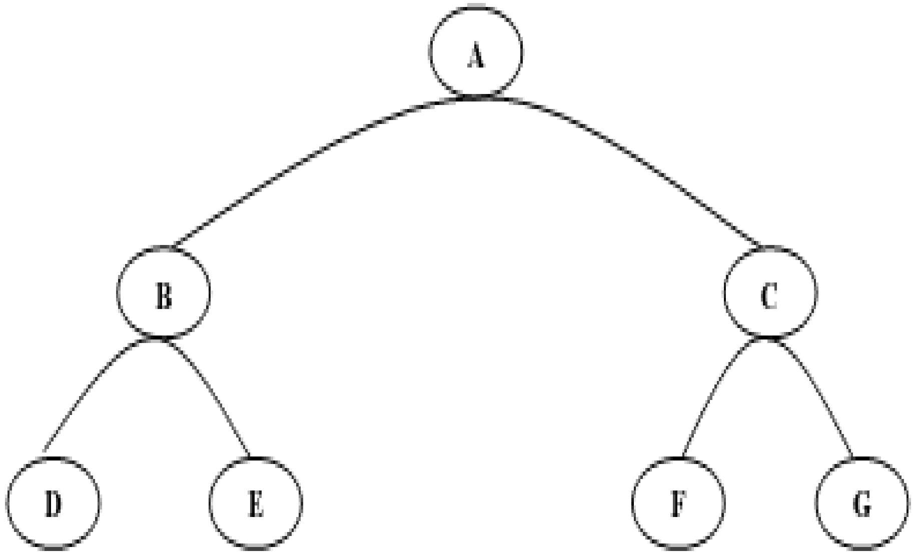
- Suatu tree dengan syarat bahwa tiap node hanya boleh memiliki maksimal **dua subtree** dan kedua subtree tersebut harus **terpisah**.
- Tiap node dalam binary tree hanya boleh memiliki **paling banyak** dua child.

Binary Tree


root
14
parent
84
43
left child
right child
13
06
33
97
53
99
leaf
64
51
25

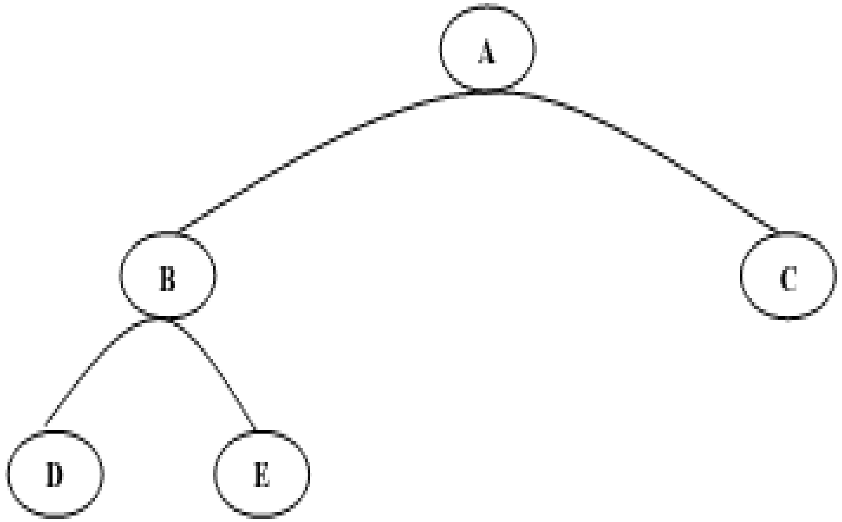
Jenis Binary Tree

- Full Binary Tree: semua node (kecuali leaf) pasti memiliki 2 anak dan tiap subtree memiliki panjang path yang sama.



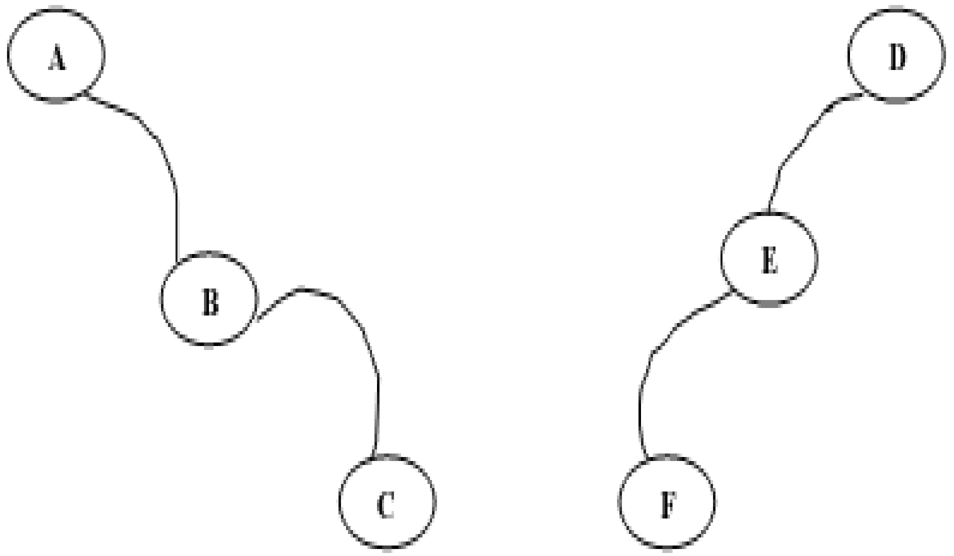
Jenis Binary Tree

- Complete Binary Tree: mirip dengan full binary tree, tapi tiap subtree boleh memiliki panjang path yang berbeda dan tiap node (kecuali leaf) memiliki 2 anak.



Jenis Binary Tree

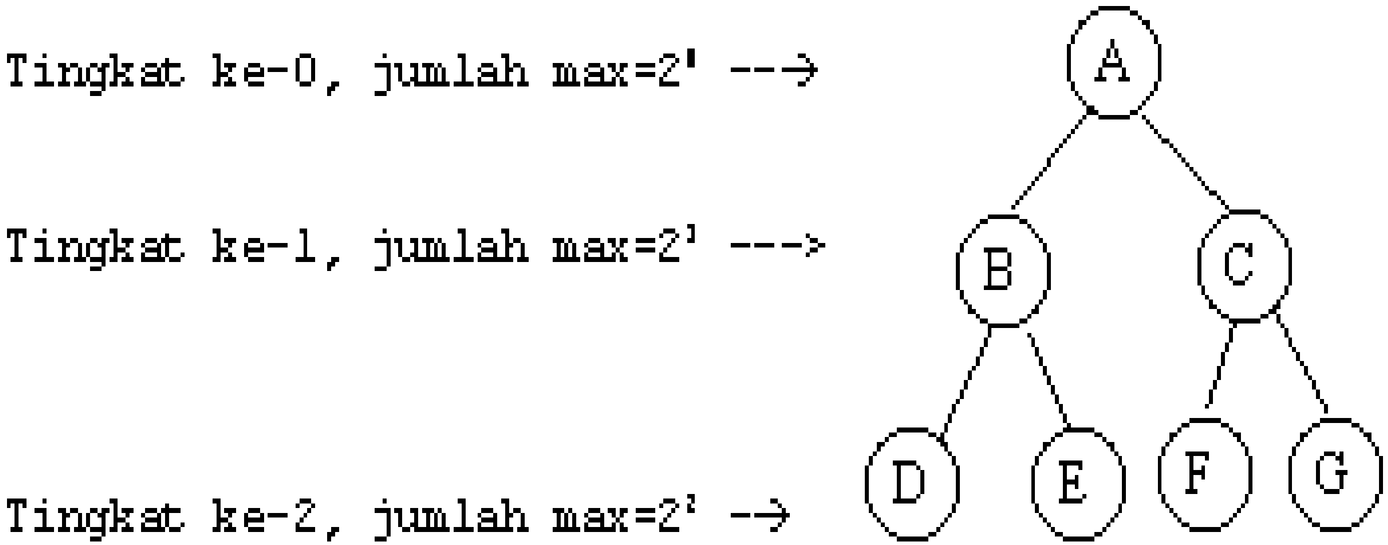
Skewed Binary Tree: binary tree yang semua nodenya (kecuali leaf) hanya memiliki satu anak.



Node pada binary tree

Jumlah maksimum node pada setiap tingkat adalah 2^n

Node pada binary tree maksimum berjumlah 2^n-1



Gambar 6.4. Pohon Biner Tingkat 2 Lengkap

Implementasi Program

Tree dapat dibuat dengan menggunakan linked list secara rekursif.

Linked list yang digunakan adalah double linked list non circular

Data yang pertama kali masuk akan menjadi node root.

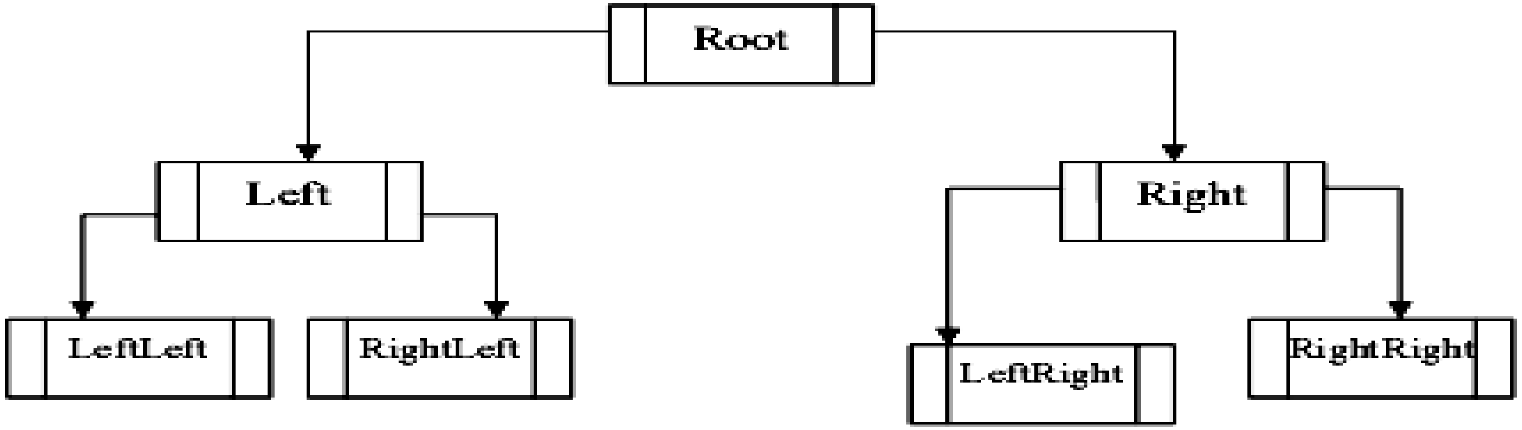
Data yang lebih kecil dari data node root akan masuk dan menempati node kiri dari node root, sedangkan jika lebih besar dari data node root, akan masuk dan menempati node di sebelah kanan node root.

Implementasi Program

Deklarasi struct

```
typedef struct Tree{
                    int data;
                    Tree *left;
                    Tree *right;
               }
```

Ilustrasi:



Deklarasi variabel:

```
Tree *pohon;
```

Operasi-operasi Tree

Create: membentuk sebuah tree baru yang kosong.

```
pohon = NULL;
```

Clear: menghapus semua elemen tree.

```
pohon = NULL;
```

Empty: mengetahui apakah tree kosong atau tidak

```
int isEmpty(Tree *pohon){
if(pohon == NULL) return 1;
else return 0;
}
```

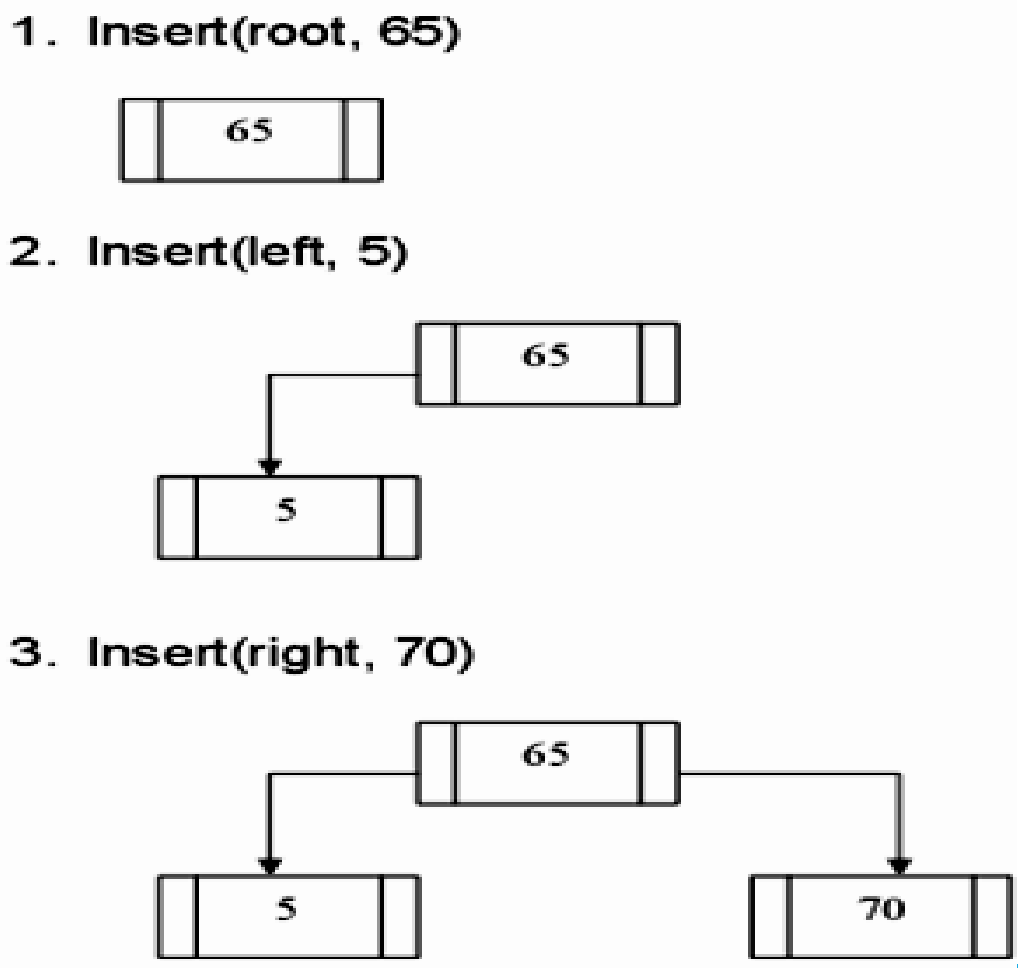
Operasi-operasi Tree

- **Insert**: menambah node ke dalam Tree secara rekursif. Jika data yang akan dimasukkan lebih besar daripada elemen root, maka akan diletakkan di node sebelah kanan, sebaliknya jika lebih kecil maka akan diletakkan di node sebelah kiri. Untuk data pertama akan menjadi elemen root.
- **Find**: mencari node di dalam Tree secara rekursif sampai node tersebut ditemukan dengan menggunakan variable bantuan ketemu. Syaratnya adalah tree tidak boleh kosong.
- **Traverse**: yaitu operasi kunjungan terhadap node-node dalam pohon dimana masing-masing node akan dikunjungi sekali.
- **Count**: menghitung jumlah node dalam Tree
- **Height** : mengetahui kedalaman sebuah Tree
- **Find Min dan Find Max** : mencari nilai terkecil dan terbesar pada Tree
- **Child** : mengetahui anak dari sebuah node (jika punya)

Jenis Transverse

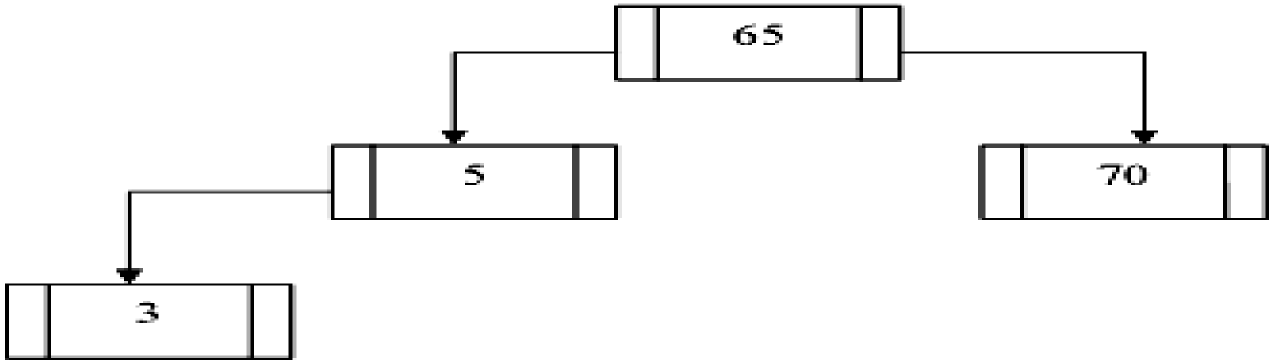
- **PreOrder**: cetak node yang dikunjungi, kunjungi left, kunjungi right
- **InOrder**: kunjungi left, cetak node yang dikunjungi, kunjungi right
- **PostOrder**: kunjungi left, kunjungi right, cetak node yang dikunjungi

Ilustrasi Insert

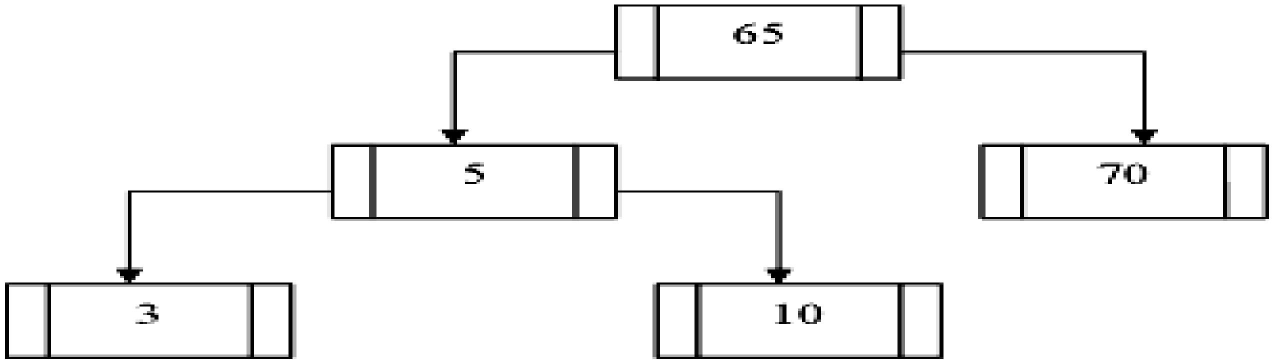


Ilustrasi Insert

4. insert(left,3)

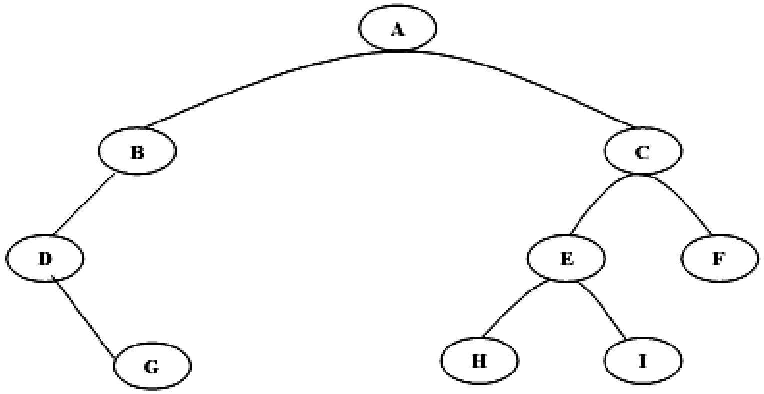


5. Insert(right, 10)



Ilustrasi Kunjungan

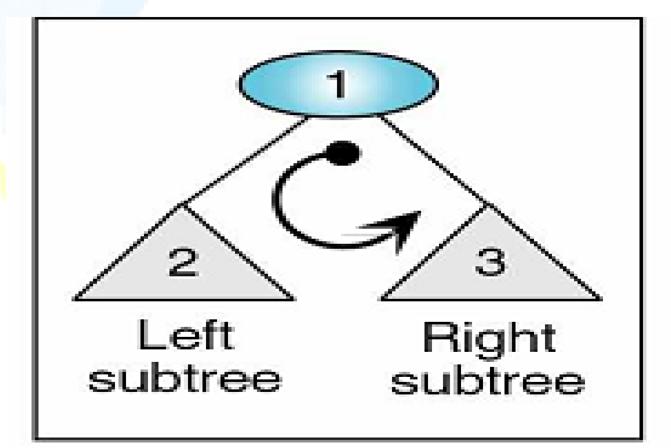
Misal terdapat Tree sebagai berikut:



1. Kunjungan PreOrder (notasi prefiks)

   Hasil kunjungan: “ABDGCEHIF”

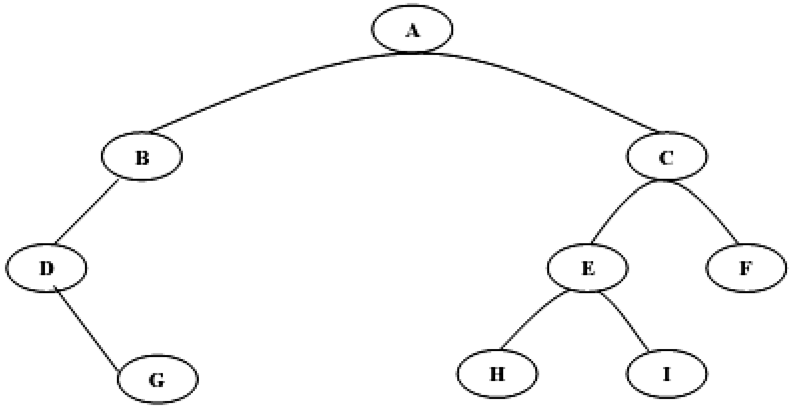
```
void preOrder(Tree *root){
      if(root != NULL){
            printf("%d ",root->data);
            preOrder(root->left);
            preOrder(root->right);
      }
}
```



(a) Preorder traversal

Ilustrasi Kunjungan

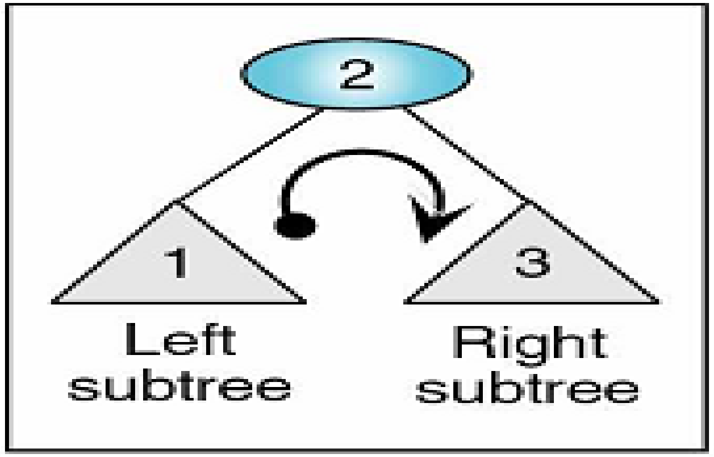
Misal terdapat Tree sebagai berikut:



2. Kunjungan InOrder (notasi infiks)

Hasil kunjungan: “DGBAHEICF”

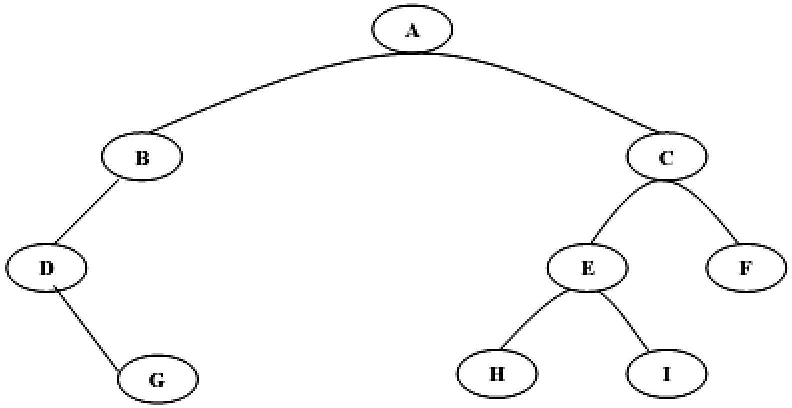
```
void inOrder(Tree *root){
      if(root != NULL){
            inOrder(root->left);
            printf("%d ",root->data);
            inOrder(root->right);
      }
}
```



(b) Inorder traversal

Ilustrasi Kunjungan

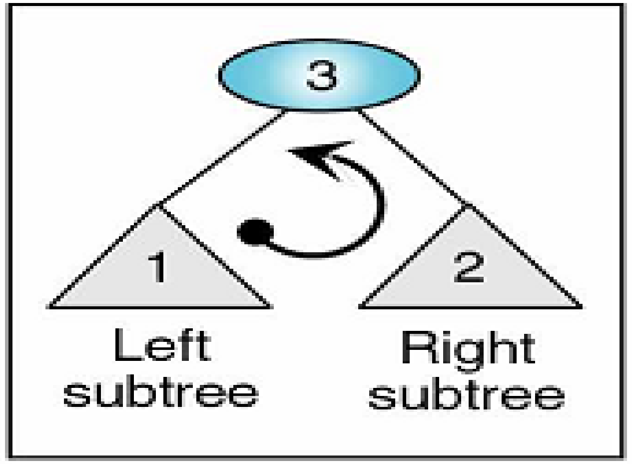
Misal terdapat Tree sebagai berikut:



3. Kunjungan PostOrder (notasi postfiks)

Hasil kunjungan: "GDBAHIEFCA"

```
void postOrder(Tree *root){
    if(root != NULL){
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ",root->data);
    }
}
```



(c) Postorder traversal

Ilustrasi Kunjungan

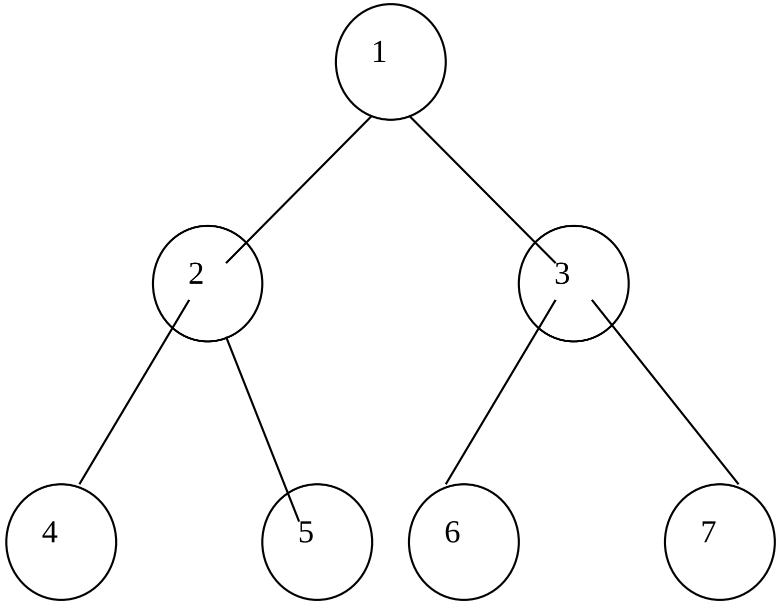
Kunjungan LevelOrder

Hasil kunjungan: “ABCDEFGHI”

Algoritma:

- Siapkan antrian yang kosong
- Inisialisasi: masukkan root ke dalam antrian
- Iterasi: selama Antrian tidak kosong, lakukan:
  - Kunjungi elemen pada antrian
  - Masukkan node->kiri dan node->kanan ke dalam antrian asal node tersebut tidak NULL.
  - Keluarkan elemen pertama pada antrian

Level Order

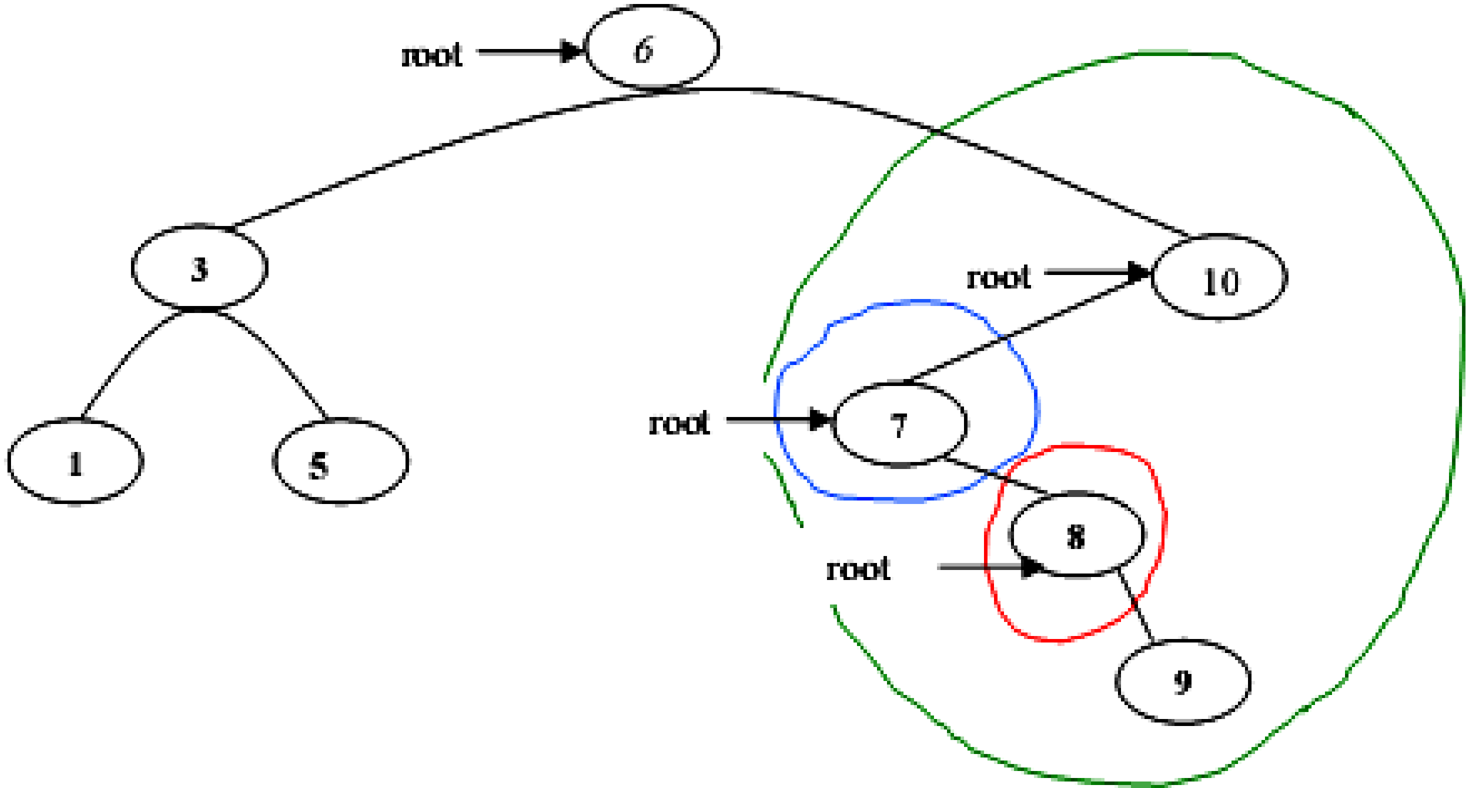


-Masukkan root ke antrian
Antrian : 1
-Kunjungi root (1), masukkan node kiri dan kanan
Antrian : 1, 2, 3
-Keluarkan antrian terdepan (node 1)
Antrian : 2, 3
-Kunjungi node 2, masukkan 4 dan 5
Antrian : 2, 3, 4, 5
-Keluarkan node terdepan (node 2)
Antrian : 3, 4, 5
-Kunjungi node 3, masukkan 6 dan 7
Antrian : 3, 4, 5, 6, 7
-Keluarkan antrian terdepan (node 3)
Antrian : 4, 5, 6, 7
-Kunjungi node 4, tidak ada anak, keluarkan (4)
-Kunjungi node 5, tidak ada anak, keluarkan (5)
-Kunjungi node 6, tidak ada anak, keluarkan (6)
-Kunjungi node 7, tidak ada anak, keluarkan (7)

Ilustrasi Searching

Ilustrasi:

Misal dicari data 8



Jumlah Node Tree

```
int count(Tree *root)
{
  if (root == NULL) return 0;
  return count(root->left) + count(root->right) + 1;
}
```

Penghitungan jumlah node dalam tree dilakukan dengan cara mengunjungi setiap node, dimulai dari root ke subtree kiri, kemudian ke subtree kanan dan masing-masing node dicatat jumlahnya, dan terakhir jumlah node yang ada di subtree kiri dijumlahkan dengan jumlah node yang ada di subtree kanan ditambah 1 yaitu node root.

Kedalaman (height) Node Tree

```
int height(Tree *root)
{
  if (root == NULL) return -1;
  int u = height(root->left), v = height(root->right);
  if (u > v) return u+1;
  else return v+1;
}
```

Penghitungan kedalaman dihitung dari setelah root, yang dimulai dari subtree bagian kiri kemudian ke subtree bagian kanan. Untuk masing-masing kedalaman kiri dan kanan akan dibandingkan, jika ternyata subtree kiri lebih dalam, maka yang dipakai adalah jumlah kedalaman subtree kiri, demikian sebaliknya. Hal ini didasarkan pada prinsip binary tree, dimana tree-nya selalu memiliki maksimal 2 node anak.

Find Min Node

```
Tree *FindMin(Tree *root)
{
 if(root == NULL)
        return NULL;
 else
        if(root->left == NULL)
                return root;
        else
                return FindMin(root->left);
}
```

Penggunaan:

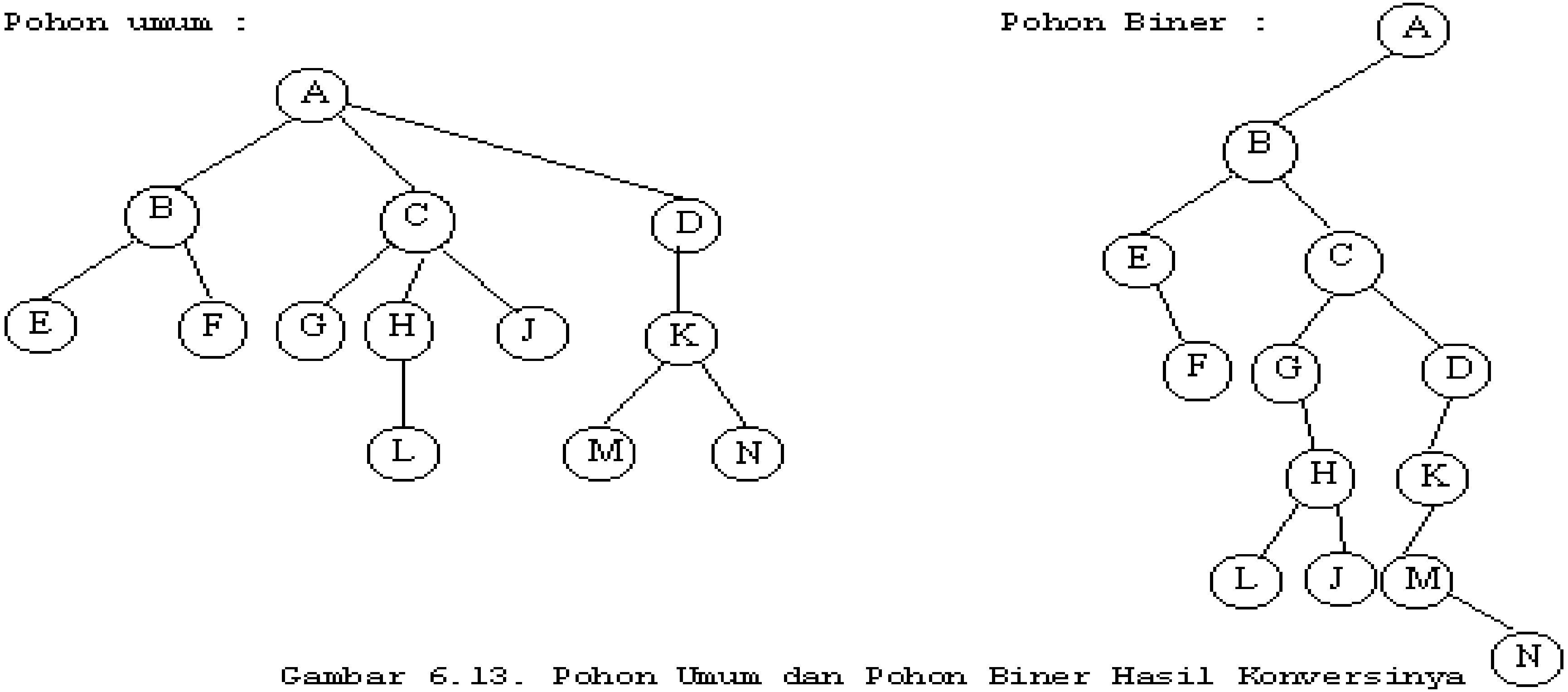
```
Tree *t = FindMin(pohon);
```

Mencari Leaf (daun)

```
void leaf(Tree *root){
 if(root == NULL) printf("kosong!");
 if(root->left!=NULL) leaf(root->left);
 if(root->right!=NULL) leaf(root->right);
   if(root->right == NULL && root->left == NULL)
 printf("%d ",root->data);
}
```

Konversi Tree Biasa ke Binary Tree

Anak pertama menjadi anak kiri, anak ke-2 menjadi cucu kanan, ke-3 jadi cicit kanan dst



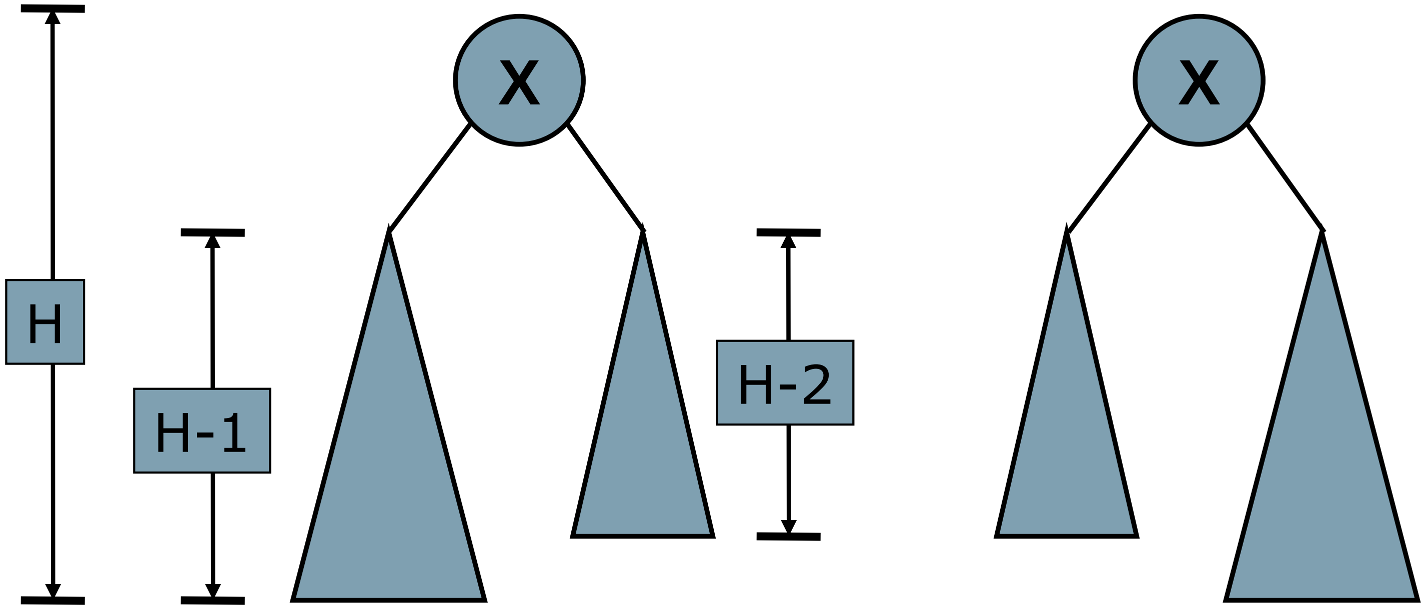
Gambar 6.13. Pohon Umum dan Pohon Biner Hasil Konversinya

Outline

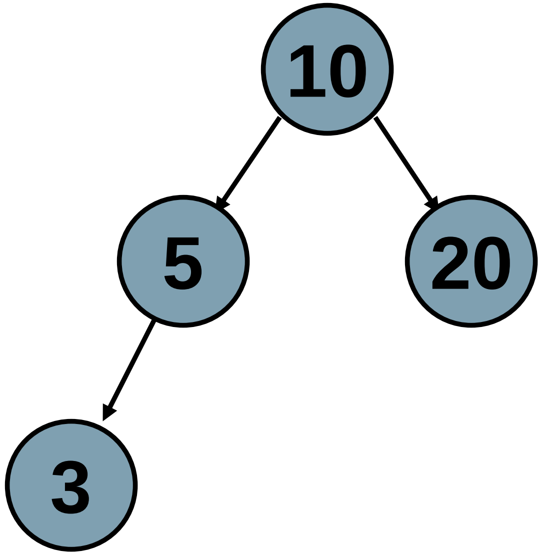
- AVL Tree
  - Definisi
  - Sifat
  - Operasi

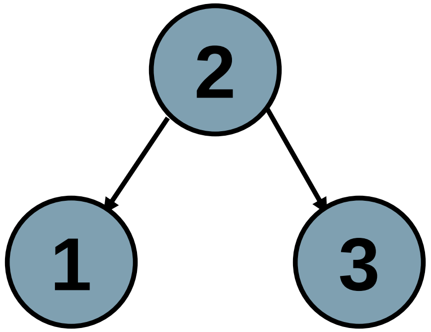
AVL Tree

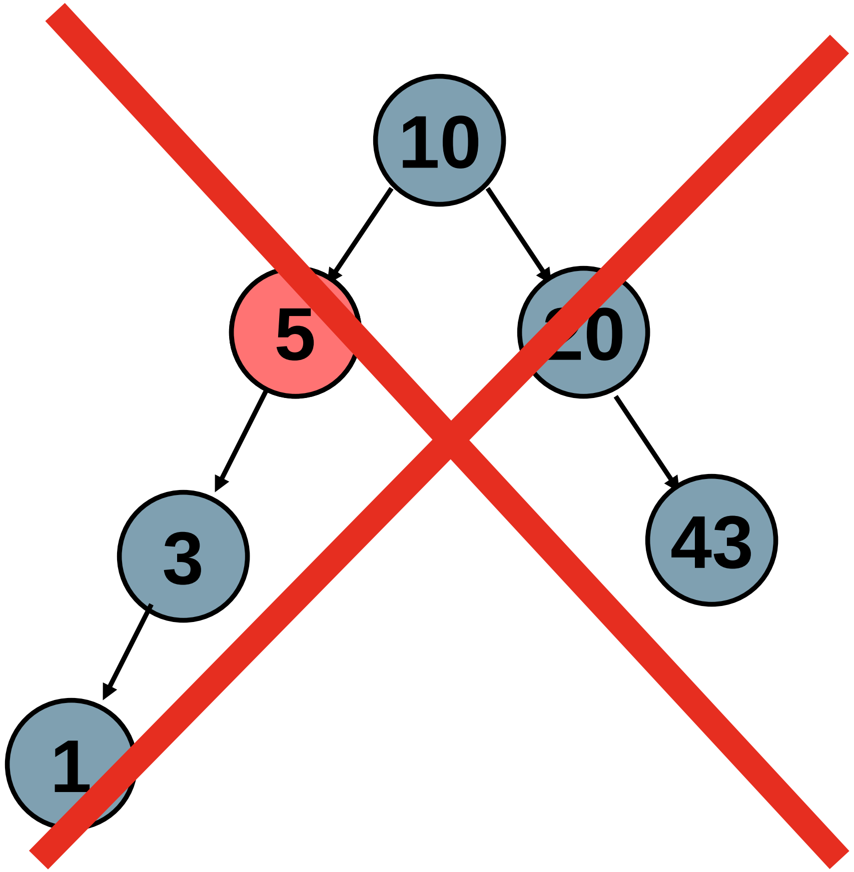
- Binary Search Trees yang tidak imbang memiliki efisiensi yang buruk. Worst case: O(n).
- AVL (Adelson-Velskii & Landis) tree adalah BST yang imbang.
- Setiap node di AVL tree memiliki balance factor bernilai -1, 0, atau 1.



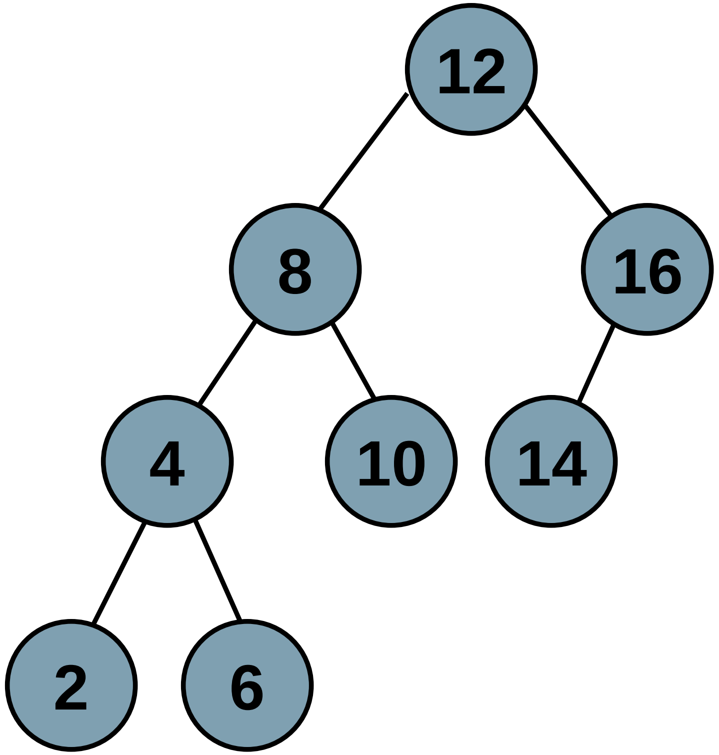
AVL Tree





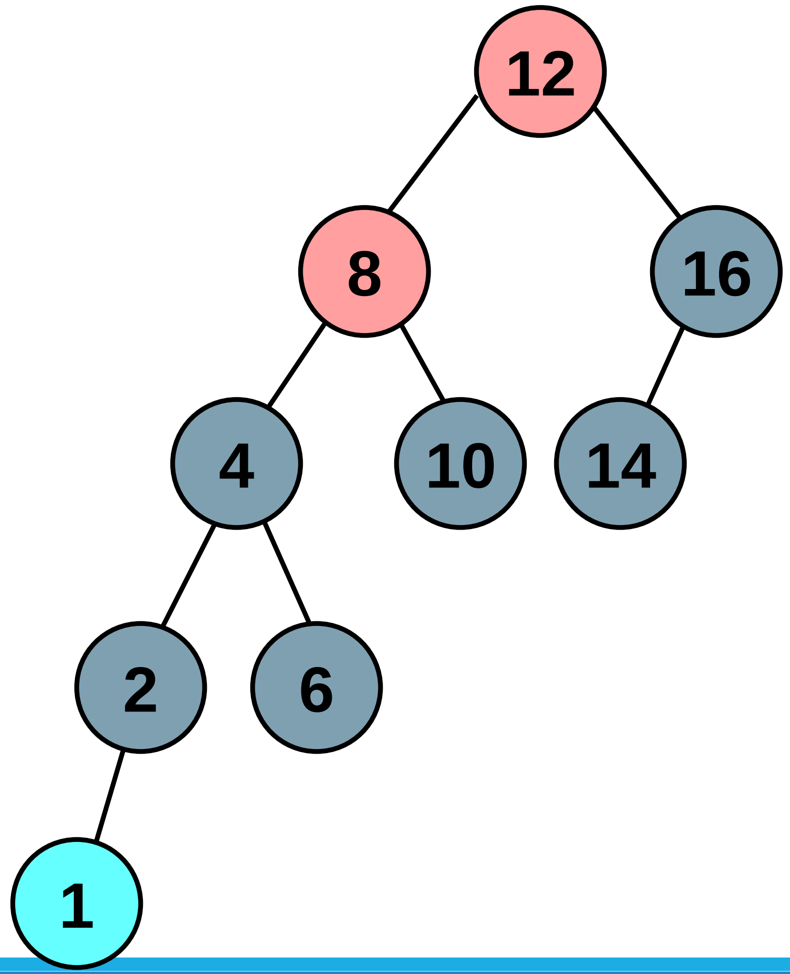


AVL Tree



Penyisipan node di AVL Tree

- Setelah insert 1



Penambahan node di AVL Tree

- Untuk menjaga tree tetap imbang, setelah penyisipan sebuah node, dilakukan pemeriksaan dari node baru → root. Node pertama yang memiliki |balance factor| > 1 diseimbangkan
- Proses penyeimbangan dilakukan dengan:
  - Single rotation
  - Double rotation

AVL Tree Balance Factor

Balance factor = $H_L - H_R$

Balance factor node di AVL tree harus +1, 0, -1

Identifier:

- **LH** left high (+1) left subtree lebih panjang dari right subtree.
- **EH** even high (0) subtree kiri dan kanan heightnya sama.
- **RH** right high (-1) left subtree lebih pendek dari right subtree.

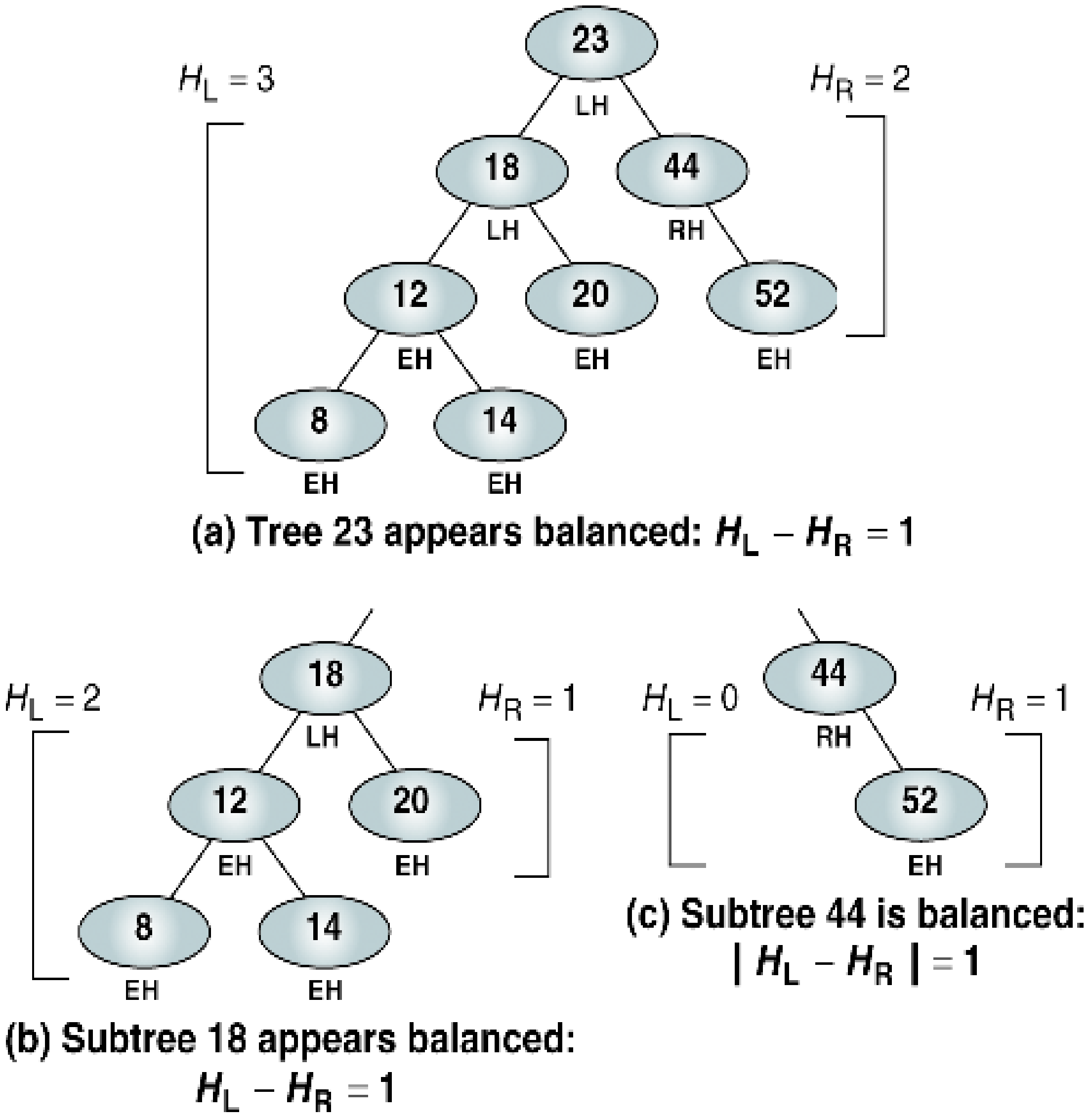


FIGURE 8-2 AVL Tree

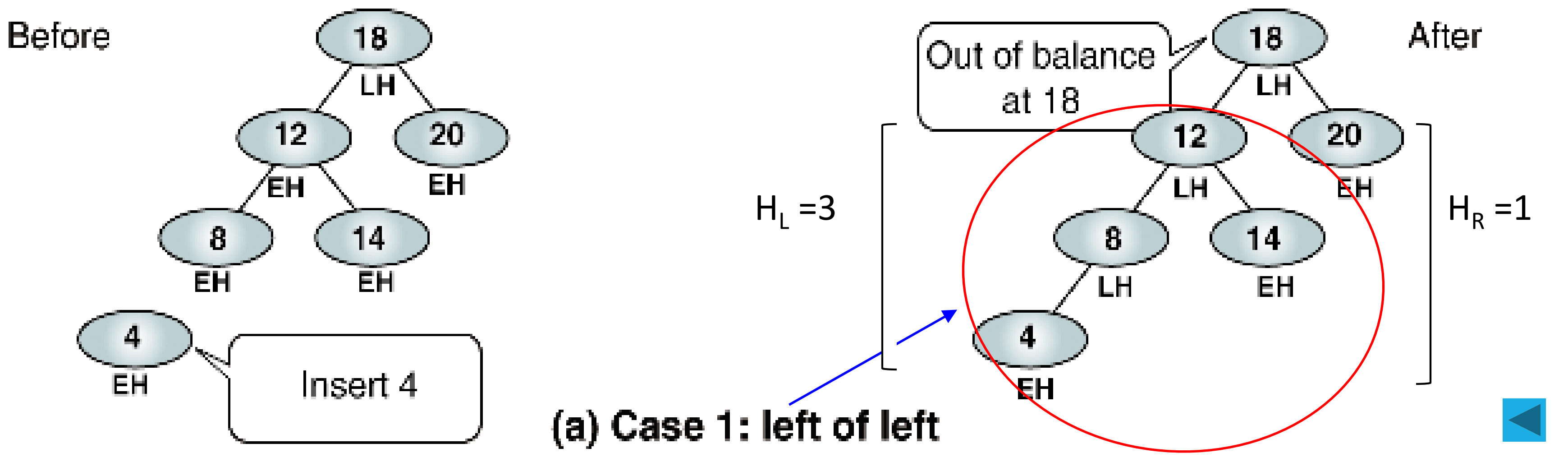
Menyeimbangkan AVL Tree

AVL trees diseimbangkan dengan merotasikan node ke kiri atau ke kanan

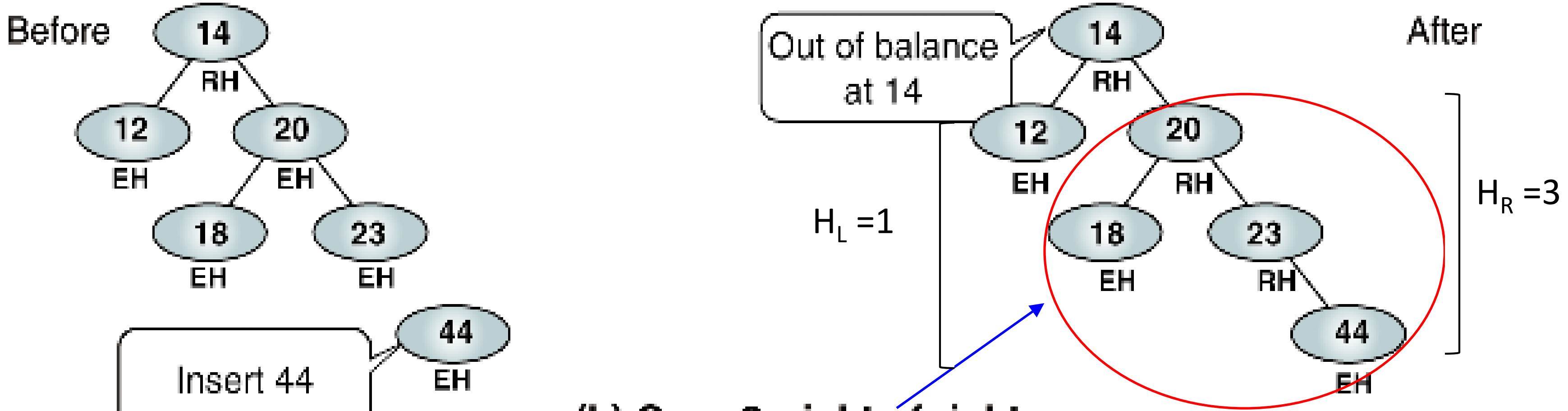
Kasus penyeimbangan pada sebuah node:

1. Left of left: mengalami left high dan left subtreenya mengalami left high.
2. Right of right: mengalami right high dan right subtreenya mengalami right high.
3. Right of left: Mengalami left high dan left subtreenya mengalami right high.
4. Left of right: Mengalami right high dan right subtreenya mengalami left high.

FIGURE 8-3 Out-of-balance AVL Trees

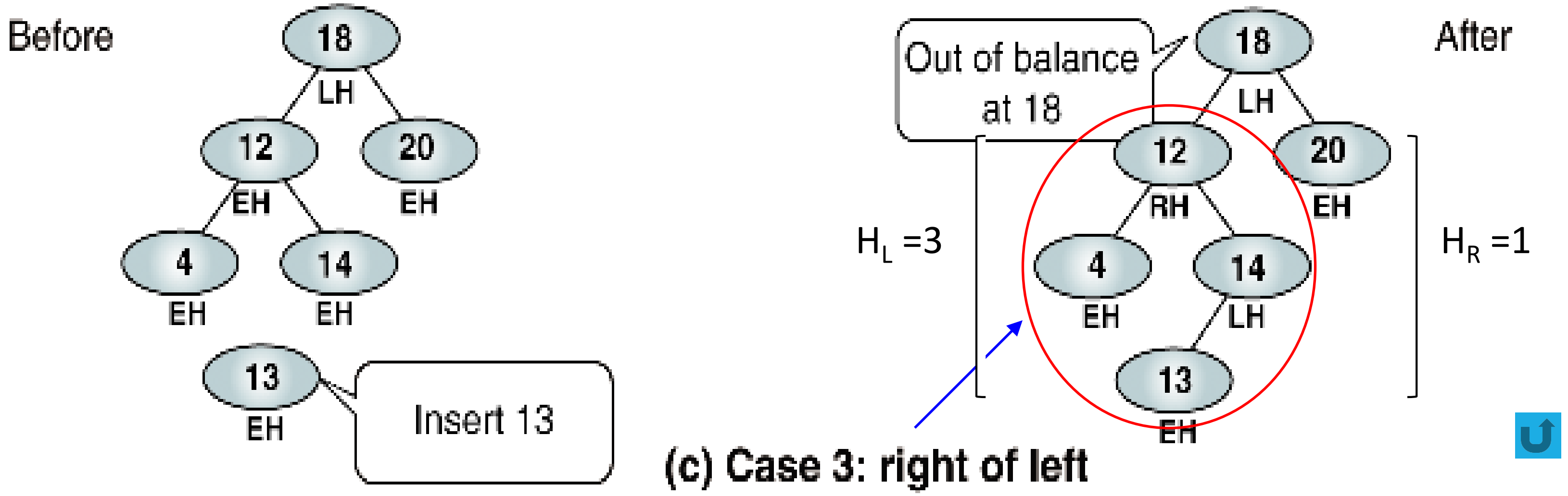


(a) Case 1: left of left

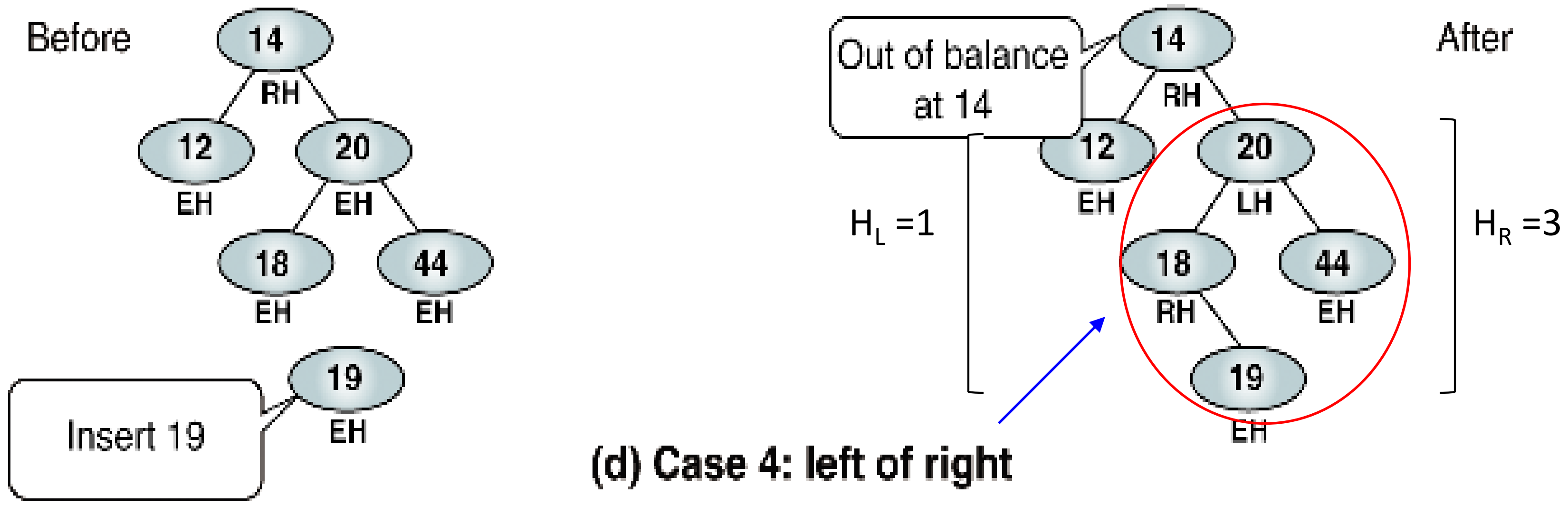


(b) Case 2: right of right

FIGURE 8-3 Out-of-balance AVL Trees (continued)



(c) Case 3: right of left



(d) Case 4: left of right

Case 1:

Left of Left

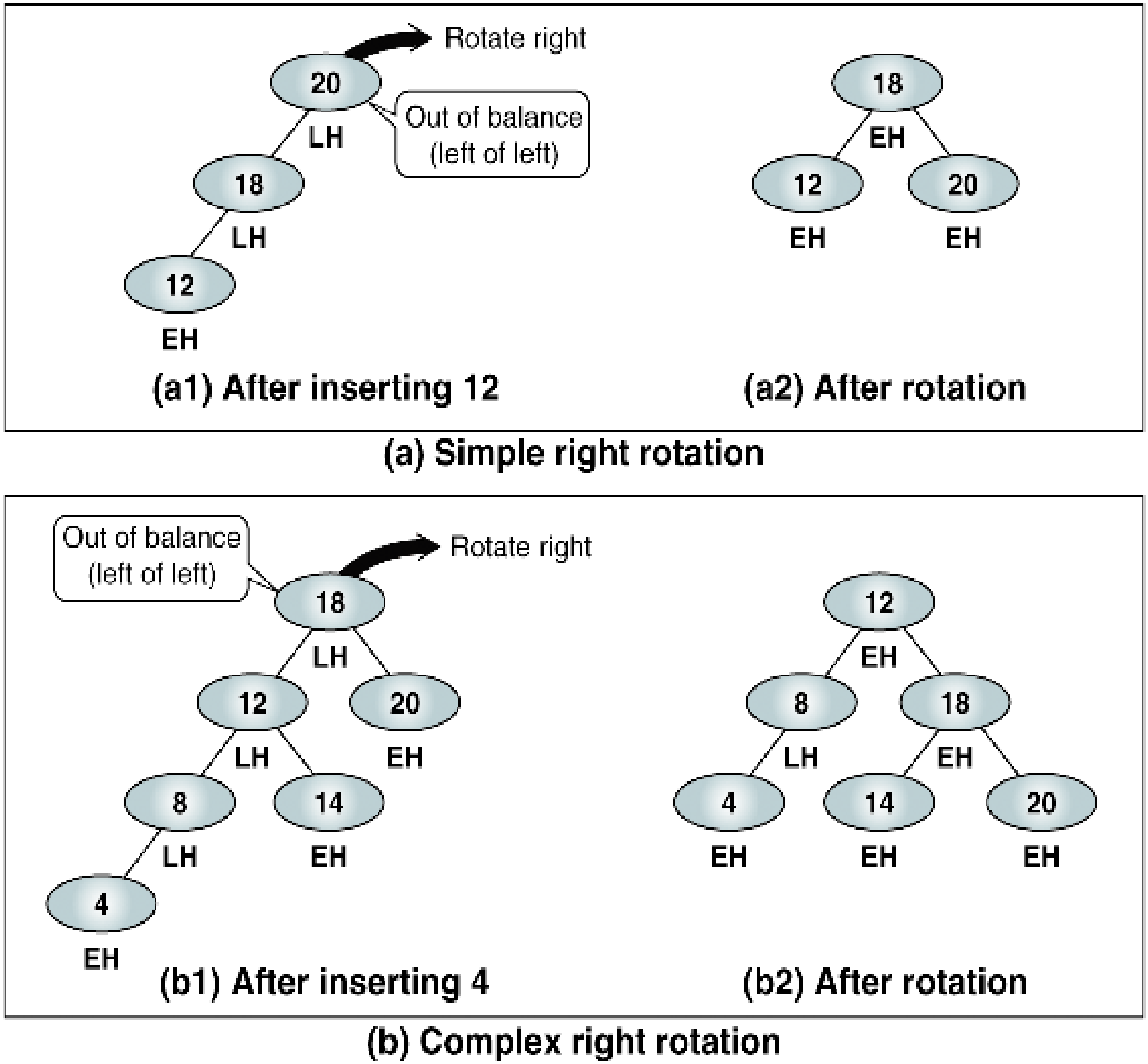
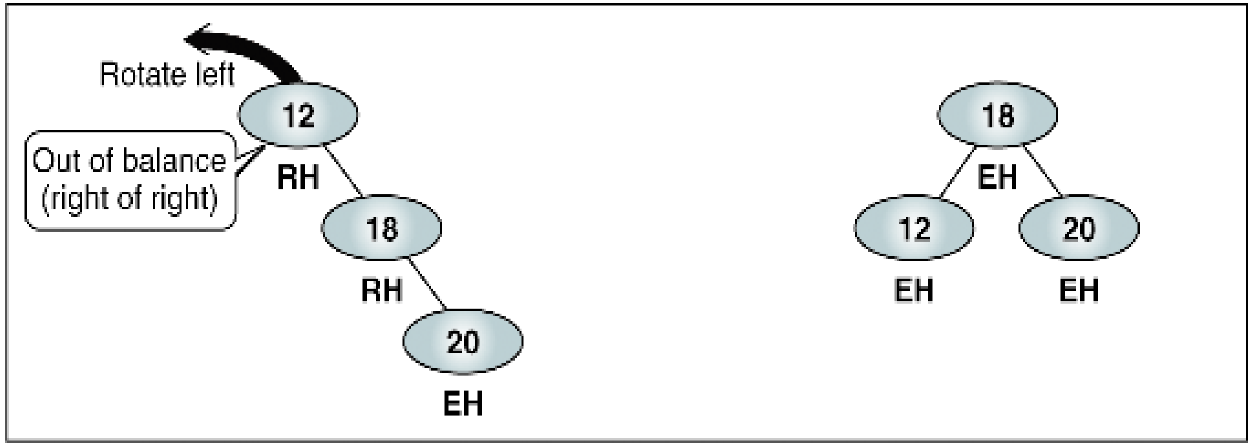


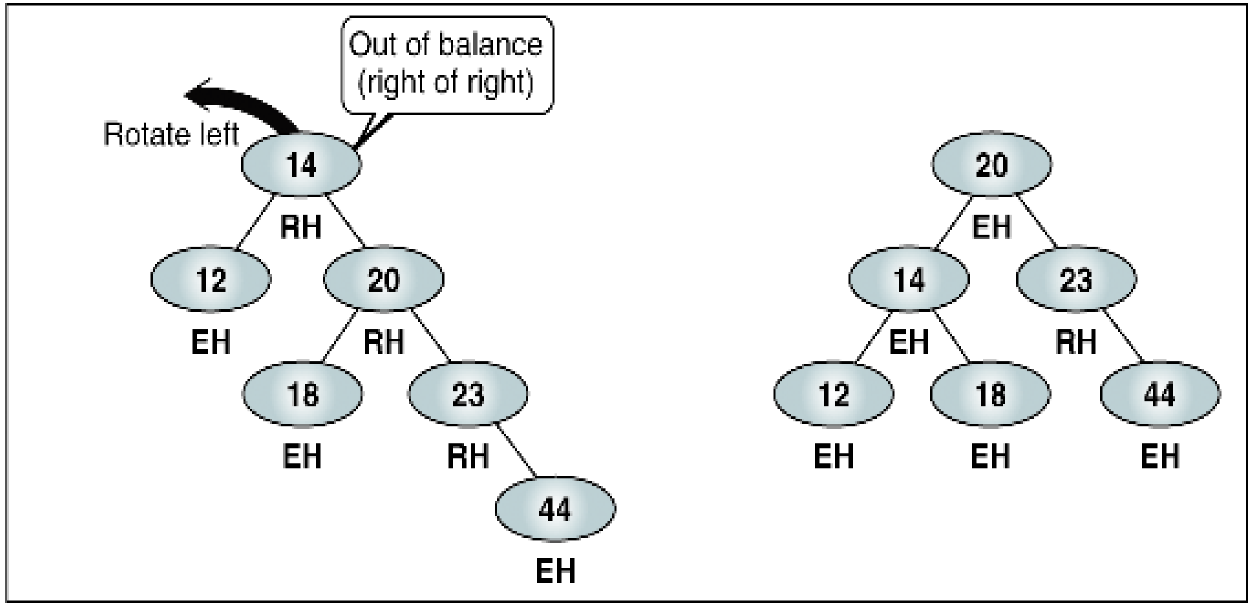
FIGURE 8-4 Left of Left—Single Rotation Right

Case 2:

Right of Right



(a) Simple left rotation



(b) Complex left rotation

FIGURE 8-5 Right of Right—Single Rotation Left

Case 3:

Right of Left

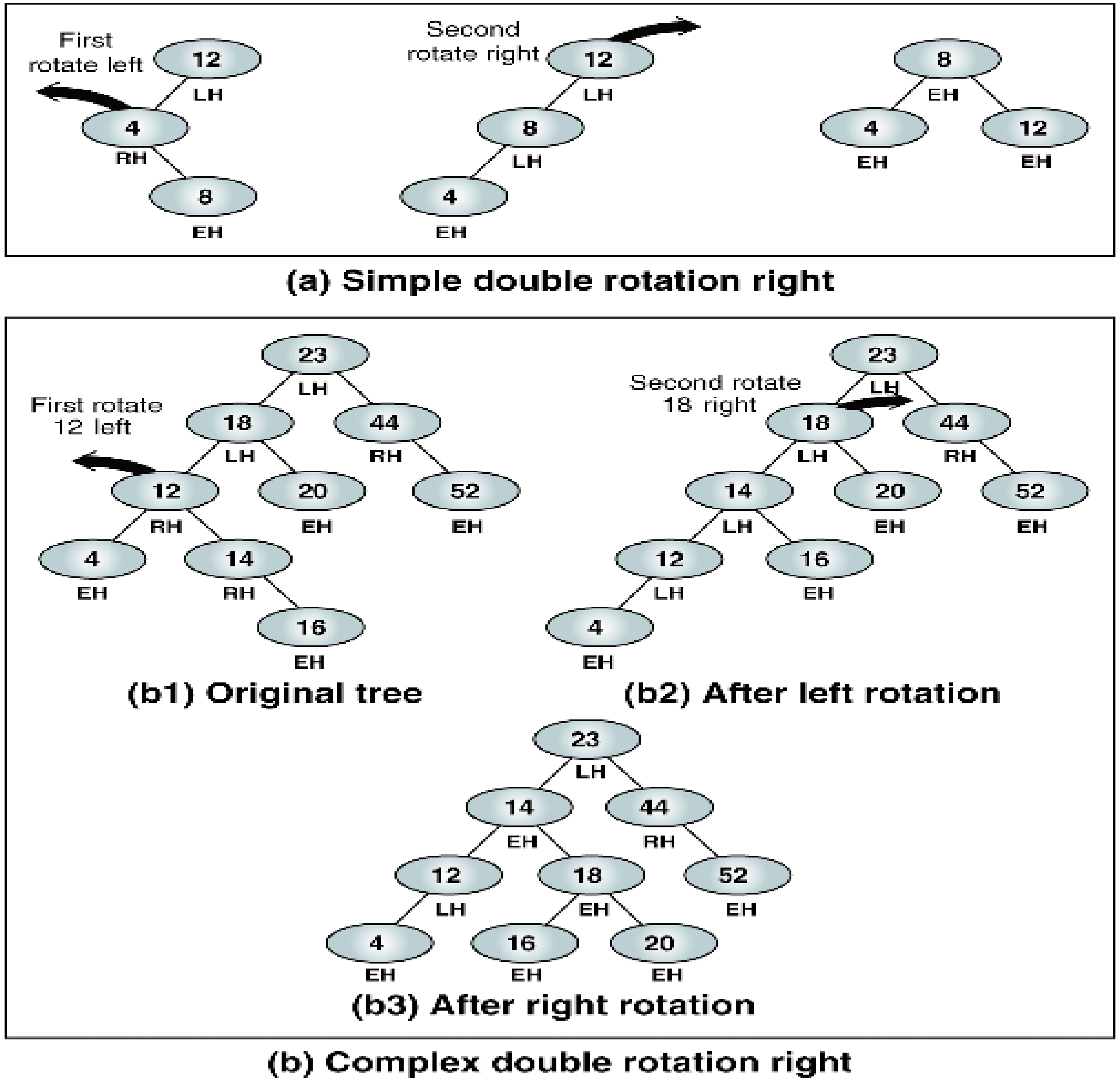


FIGURE 8-6 Right of Left—Double Rotation Right

Case 4:

Left of Right

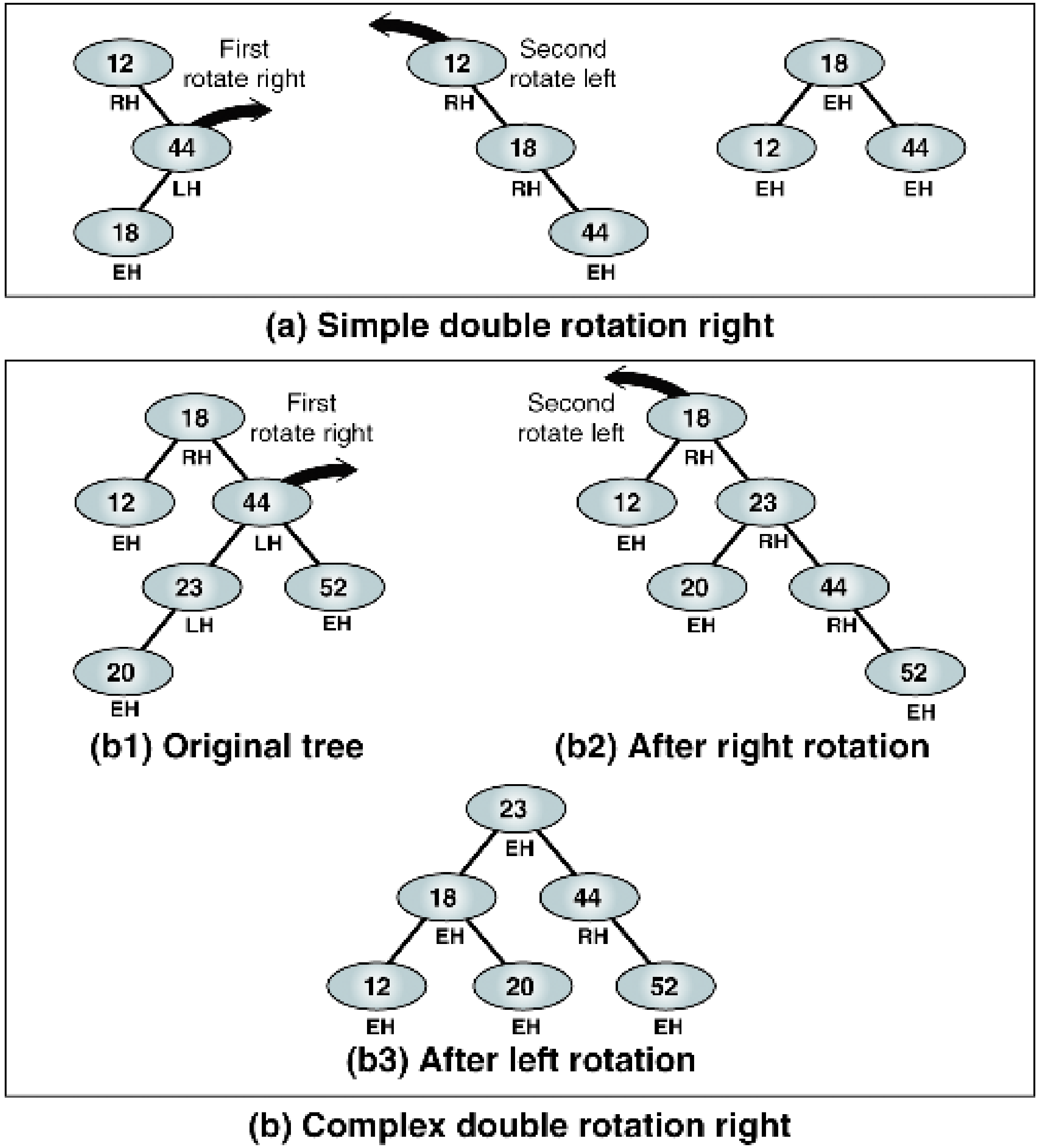
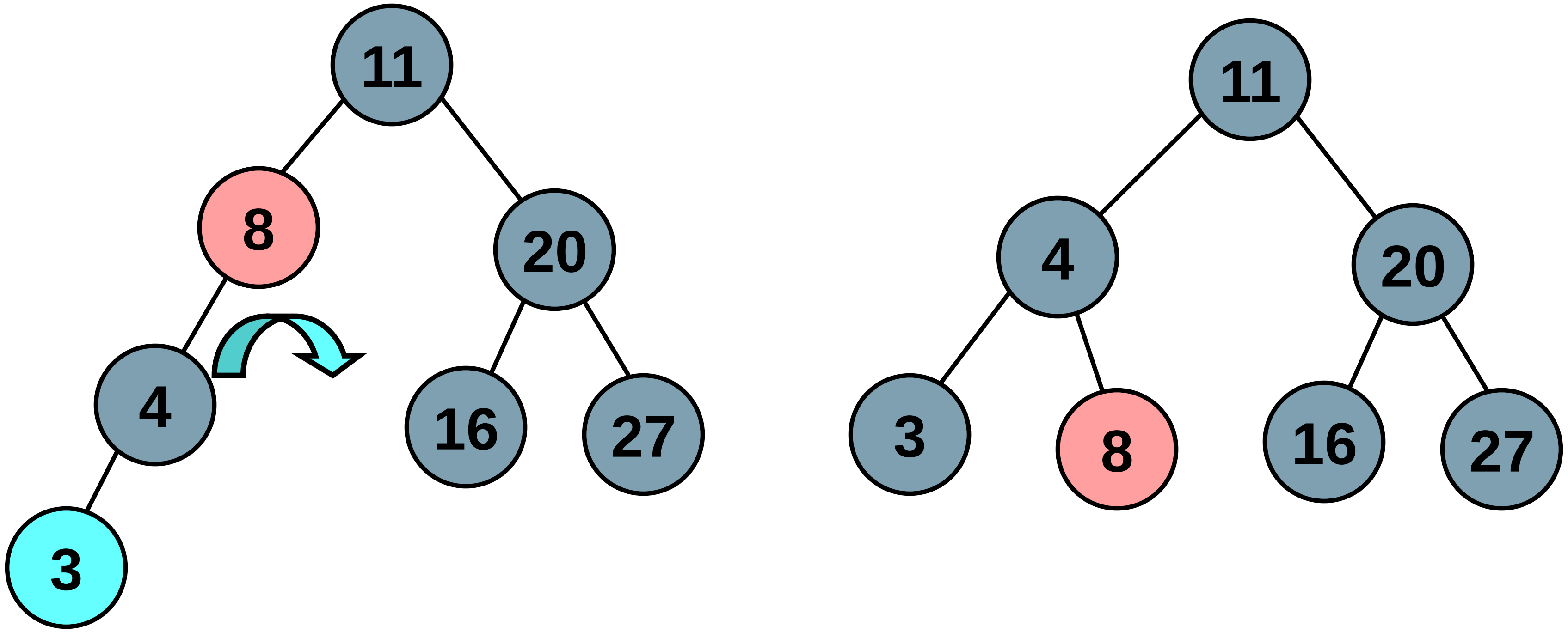


FIGURE 8-7 Left of Right—Double Rotation Right

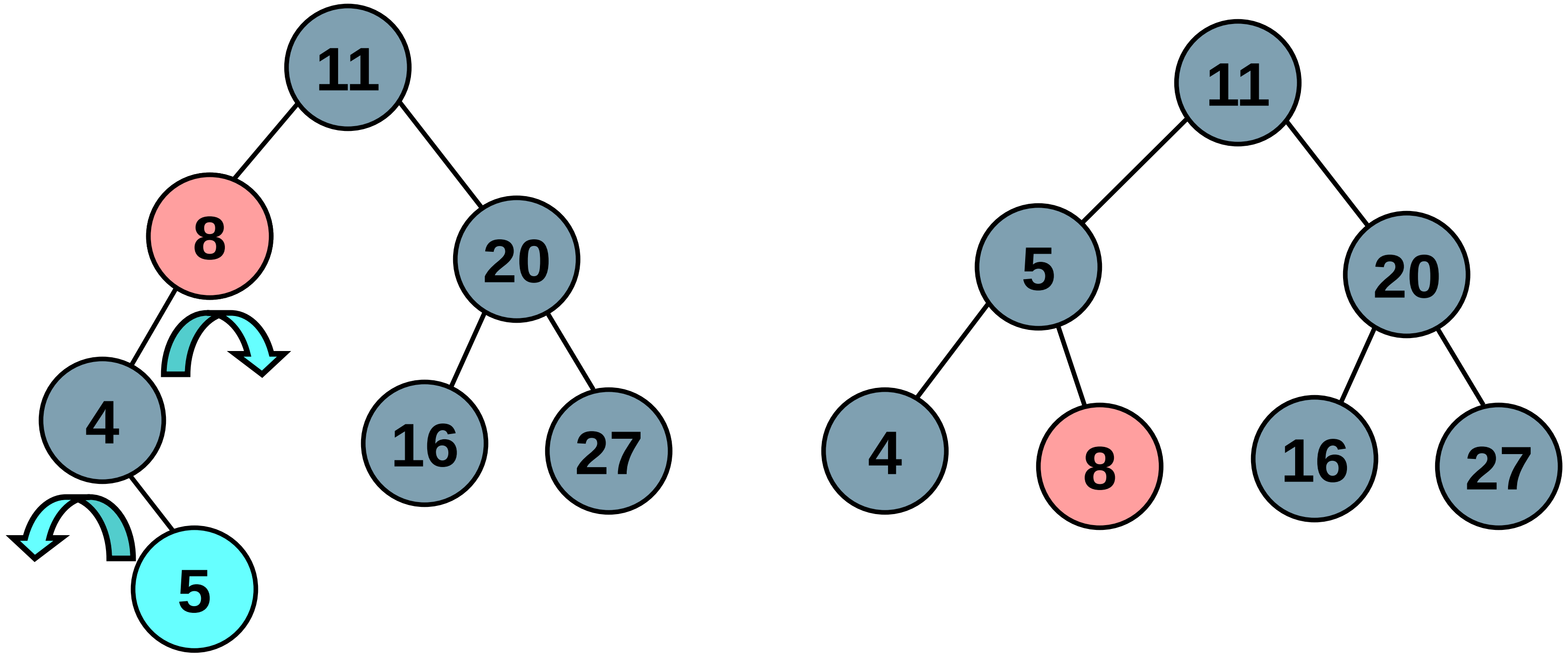
Contoh

- Sisipkan 3 ke AVL tree



Contoh

- Penyisipan 5 ke AVL tree



Menghapus node di AVL Tree

- Proses menghapus sebuah node di AVL tree hampir sama dengan BST. Penghapusan sebuah node dapat menyebabkan tree tidak imbang
- Setelah menghapus sebuah node, lakukan pengecekan dari node yang dihapus → root.
- Gunakan single atau double rotation untuk menyeimbangkan node yang tidak imbang.
- Pencarian node yang imbalance diteruskan sampai root.

Menghapus node di AVL Tree

- Tahap penghapusan:
  - Case 1: X merupakan leaf, hapus X
  - Case 2: jika X memiliki 1 child, gunakan child tersebut untuk menggantikan X. Kemudian hapus X
  - Case 3: Jika X memiliki 2 child, ganti nilai X dengan node terbesar pada left subtree atau node terkecil pada right subtree. Hapus node yang nilainya digunakan untuk mengganti X
- Tahap menyeimbangkan node yang balance factornya tidak -1, 0, 1, dilakukan dari node yang dihapus menuju root.

ALGORITHM 8-1 AVL Tree Insert

```
Algorithm AVLInsert (root, newData)
Using recursion, insert a node into an AVL tree.
   Pre    root is pointer to first node in AVL tree/subtree
          newData is pointer to new node to be inserted
   Post   new node has been inserted
   Return root returned recursively up the tree
 1 if (subtree empty)
   Insert at root
   1  insert newData at root
   2  return root
 2 end if
 3 if (newData < root)
   1  AVLInsert (left subtree, newData)
   2  if (left subtree taller)
      1  leftBalance (root)
   3  end if
 4 else
   New data >= root data
   1  AVLInsert (right subtree, newPtr)
   2  if(right subtree taller)
      1  rightBalance (root)
   3  end if
 5 end if
 6 return root
end AVLInsert
```

ALGORITHM 8-2 AVL Tree Left Balance

```
Algorithm leftBalance (root)
This algorithm is entered when the root is left high (the
left subtree is higher than the right subtree).
  Pre   root is a pointer to the root of the [sub]tree
  Post  root has been updated (if necessary)
1 if (left subtree high)
   1 rotateRight (root)
2 else
   1 rotateLeft (left subtree)
   2 rotateRight (root)
3 end if
end leftBalance
```

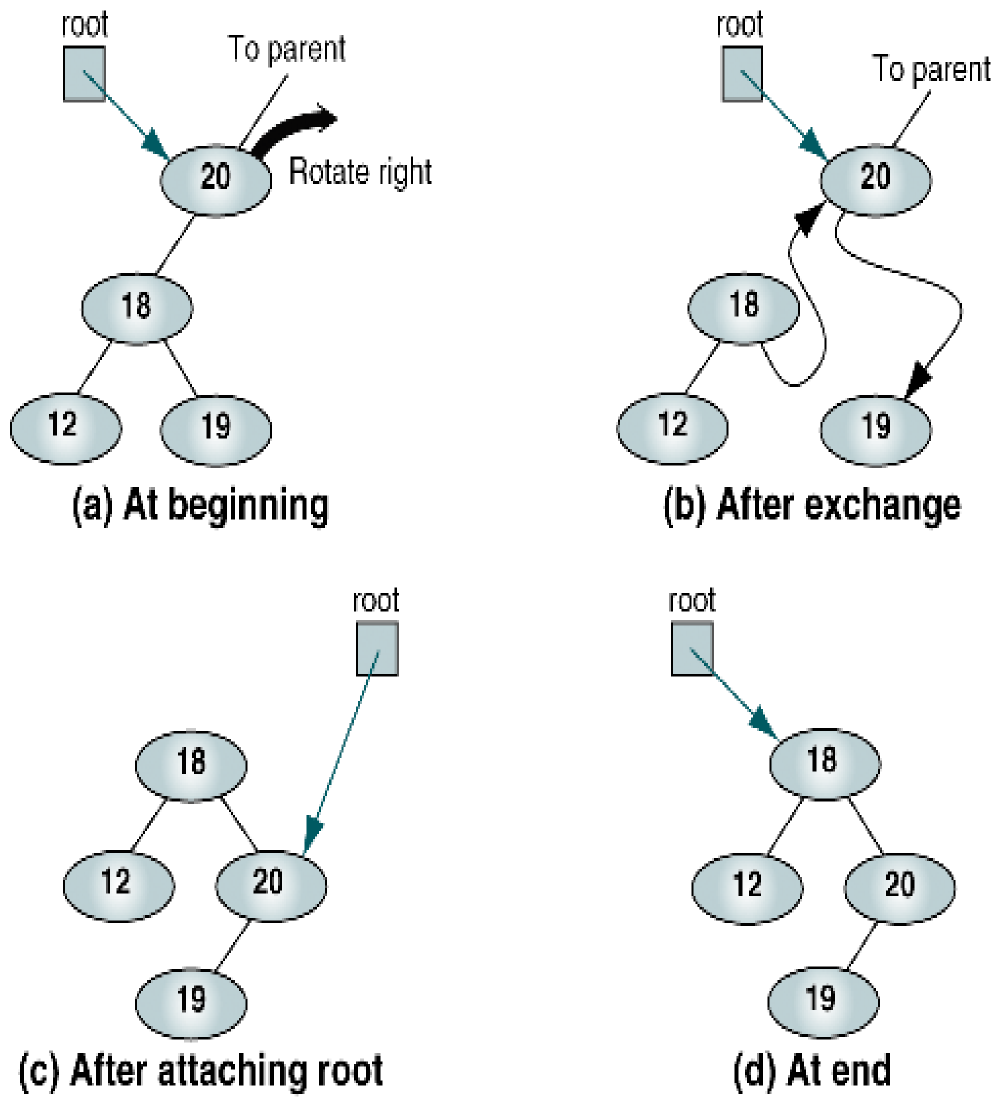


FIGURE 8-10 AVL Tree Rotate Right

ALGORITHM 8-3 Rotate AVL Tree Right and Left

```
Algorithm rotateRight (root)
This algorithm exchanges pointers to rotate the tree right.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
 1 exchange left subtree with right subtree of left subtree
 2 make left subtree new root
end rotateRight

Algorithm rotateLeft (root)
This algorithm exchanges pointers to rotate the tree left.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
 1 exchange right subtree with left subtree of right subtree
 2 make right subtree new root
end rotateLeft
```

ALGORITHM 8-4 AVL Tree Delete

```
Algorithm AVLDelete (root, dltKey, success)
This algorithm deletes a node from an AVL tree and
rebalances if necessary.
   Pre    root is a pointer to a [sub]tree
          dltKey is the key of node to be deleted
          success is reference to boolean variable
   Post   node deleted if found, tree unchanged if not
          success set true (key found and deleted)
             or false (key not found)
   Return pointer to root of [potential] new subtree
1 if Return (empty subtree)
     Not found
  1  set success to false
  2  return null
2 end if
3 if (dltKey < root)
  1  set left-subtree to AVLDelete(left subtree, dltKey,
                                   success)
```

continued

ALGORITHM 8-4 AVL Tree Delete *(continued)*

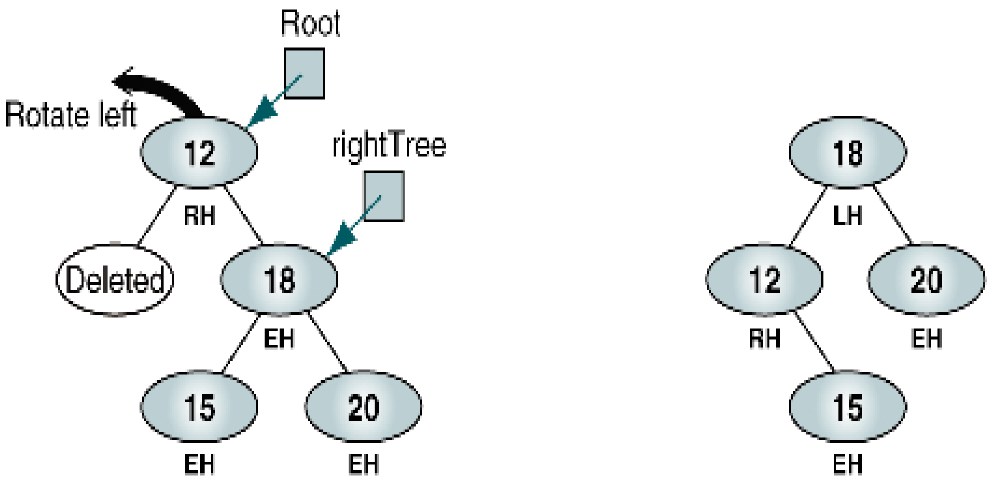
```
    2  if (tree shorter)
       1  set root to deleteRightBalance(root)
    3  end if
 4  elseif (dltKey > root)
    1  set right subtree to AVLDelete(root->right, dltKey,
                                                   success)
    2  if (tree shorter)
       1  set root to deleteLeftBalance (root)
    3  end if
 5  else
       Delete node found--test for leaf node
    1  save root
```

ALGORITHM 8-4 AVL Tree Delete *(continued)*

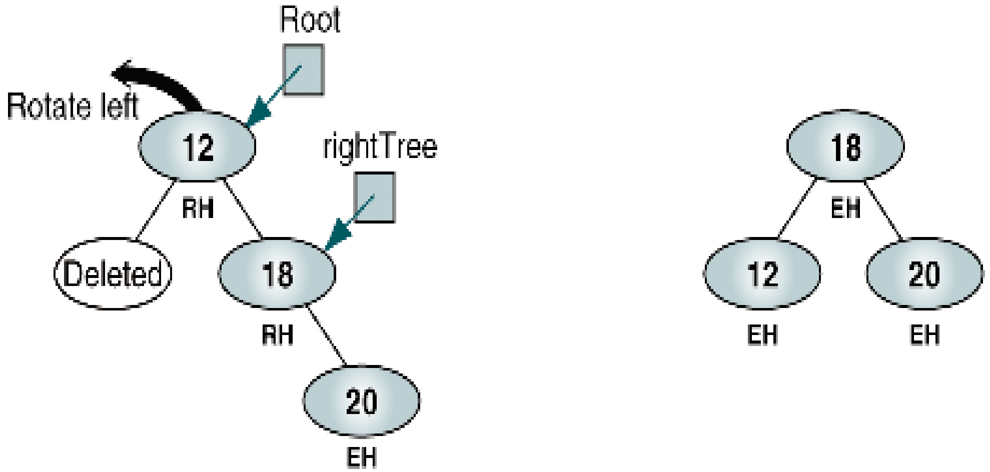
```
   2  if (no right subtree)
      1  set success to true
      2  return left subtree
   3  elseif (no left subtree)
         Have right but no left subtree
      1  set success to true
      2  return right subtree
   4  else
         Deleted node has two subtrees
         Find substitute--largest node on left subtree
      1  find largest node on left subtree
      2  save largest key
      3  copy data in largest to root
      4  set left subtree to AVLDelete(left subtree,
                                      largest key, success)
      5  if (tree shorter)
         1  set root to dltRightBal (root)
      6  end if
   5  end if
 6  end if
 7  return root
end AVLDelete
```

ALGORITHM 8-5 AVL Tree Delete Right Balance

```
Algorithm deleteRightBalance (root)
The [sub]tree is shorter after a deletion on the left branch.
If necessary, balance the tree by rotating.
   Pre    tree is shorter
   Post   balance restored
   Return new root
1  if (tree not balanced)
      No rotation required if tree left or even high
   1  set rightOfRight to right subtree
   2  if (rightOfRight left high)
         Double rotation required
      1  set leftOfRight to left subtree of rightOfRight
         Rotate right then left
      2  right subtree = rotateRight (rightOfRight)
      3  root          = rotateLeft (root)
   3  else
         Single rotation required
      1  set root to rotateLeft (root)
   4  end if
2  end if
3  return root
end deleteRightBalance
```



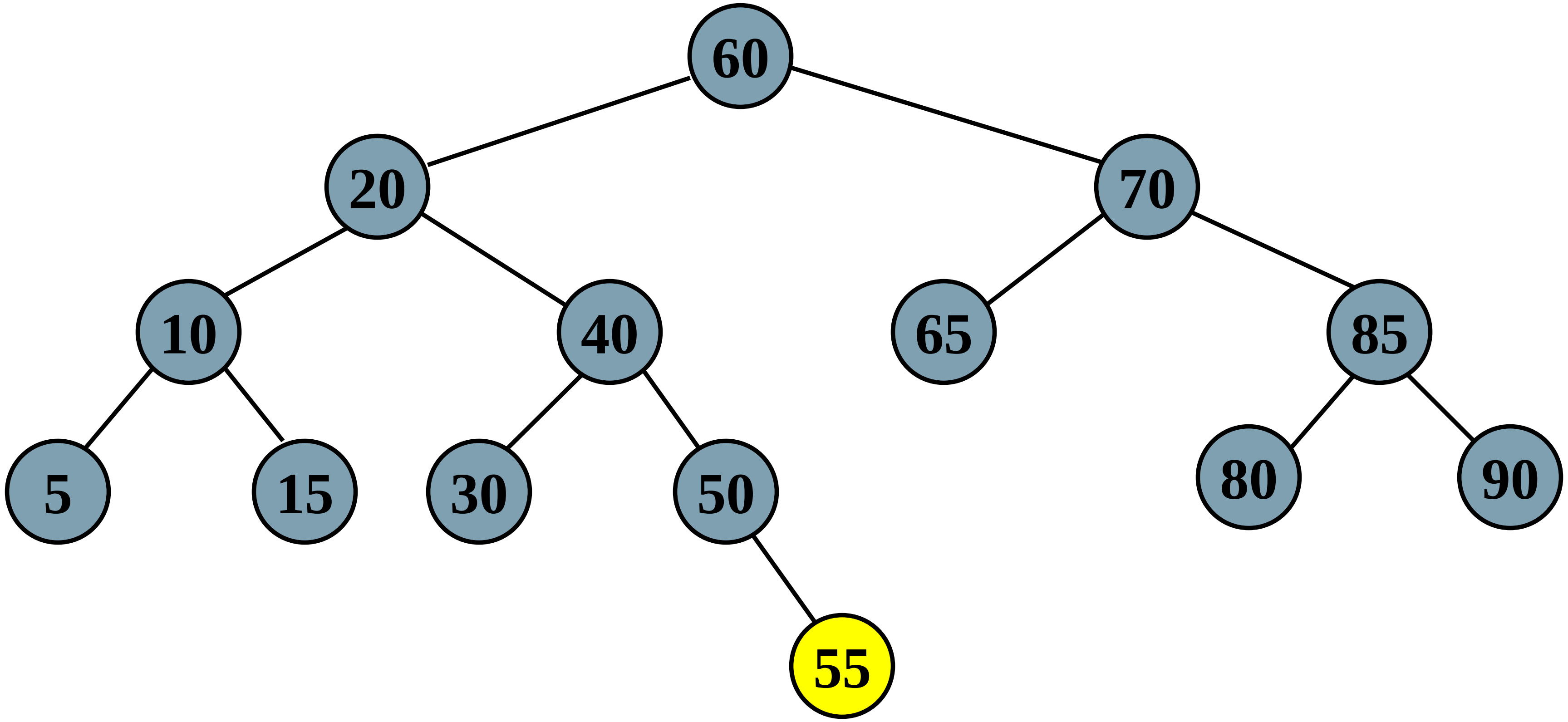
(a) Right subtree is even balanced



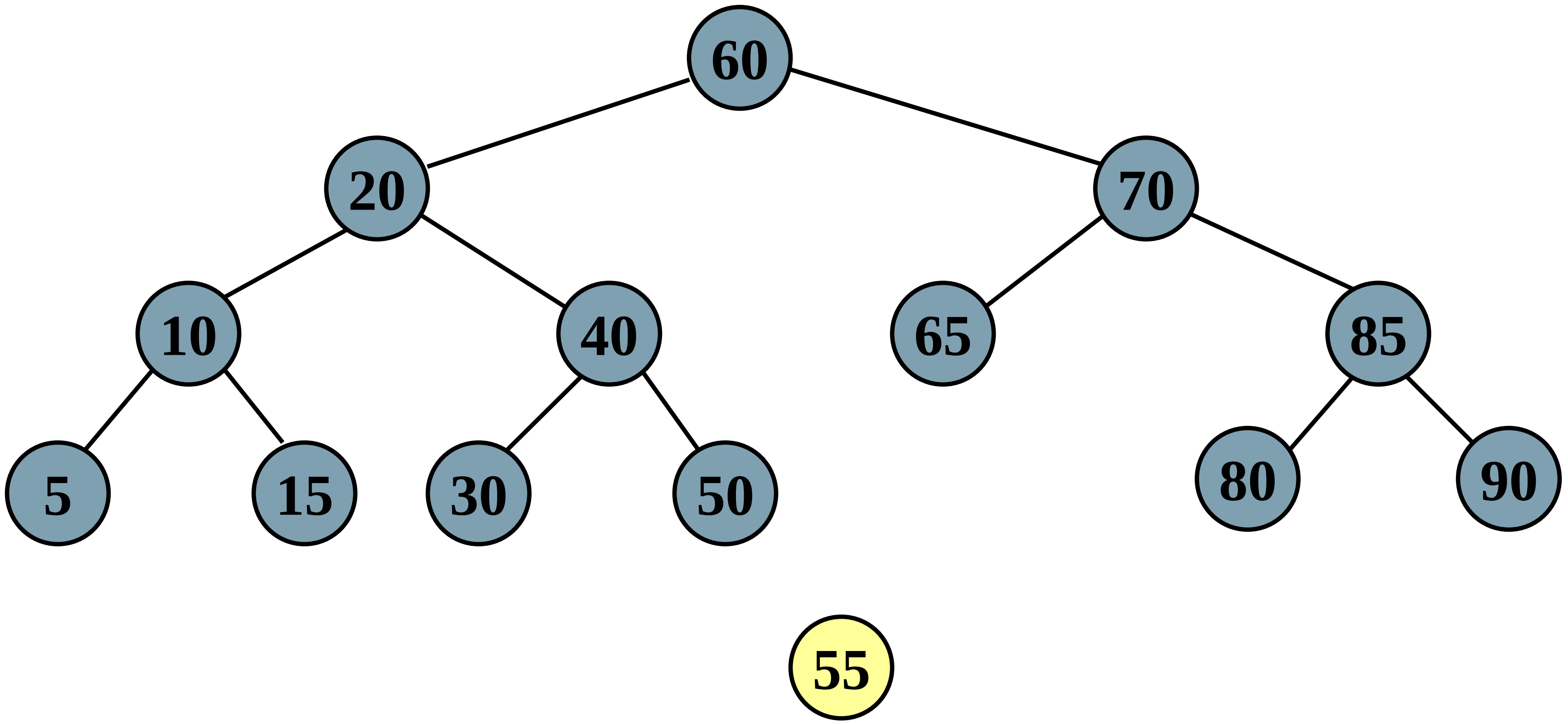
(b) Right subtree is not even balanced

FIGURE 8-11 AVL Tree Delete Balancing

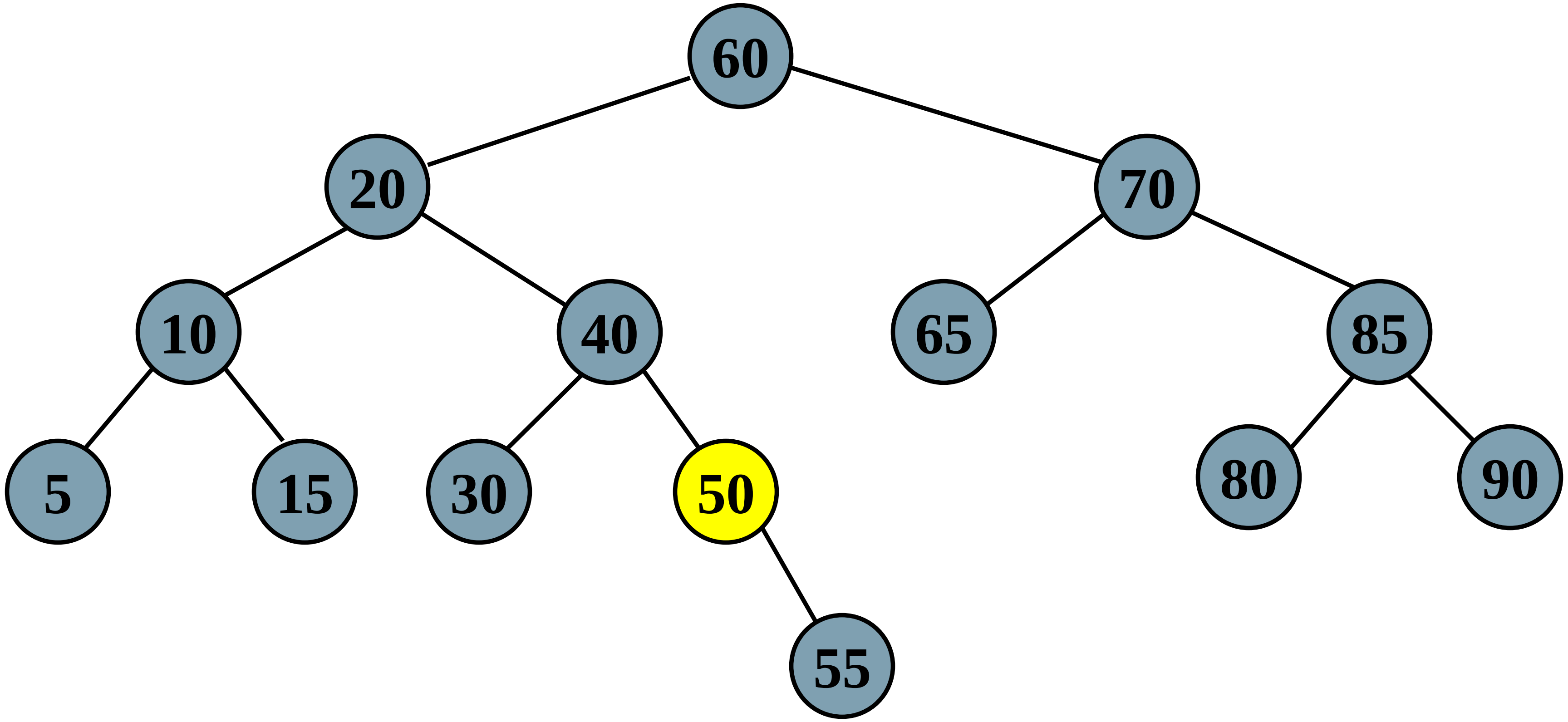
Delete 55 (case 1)



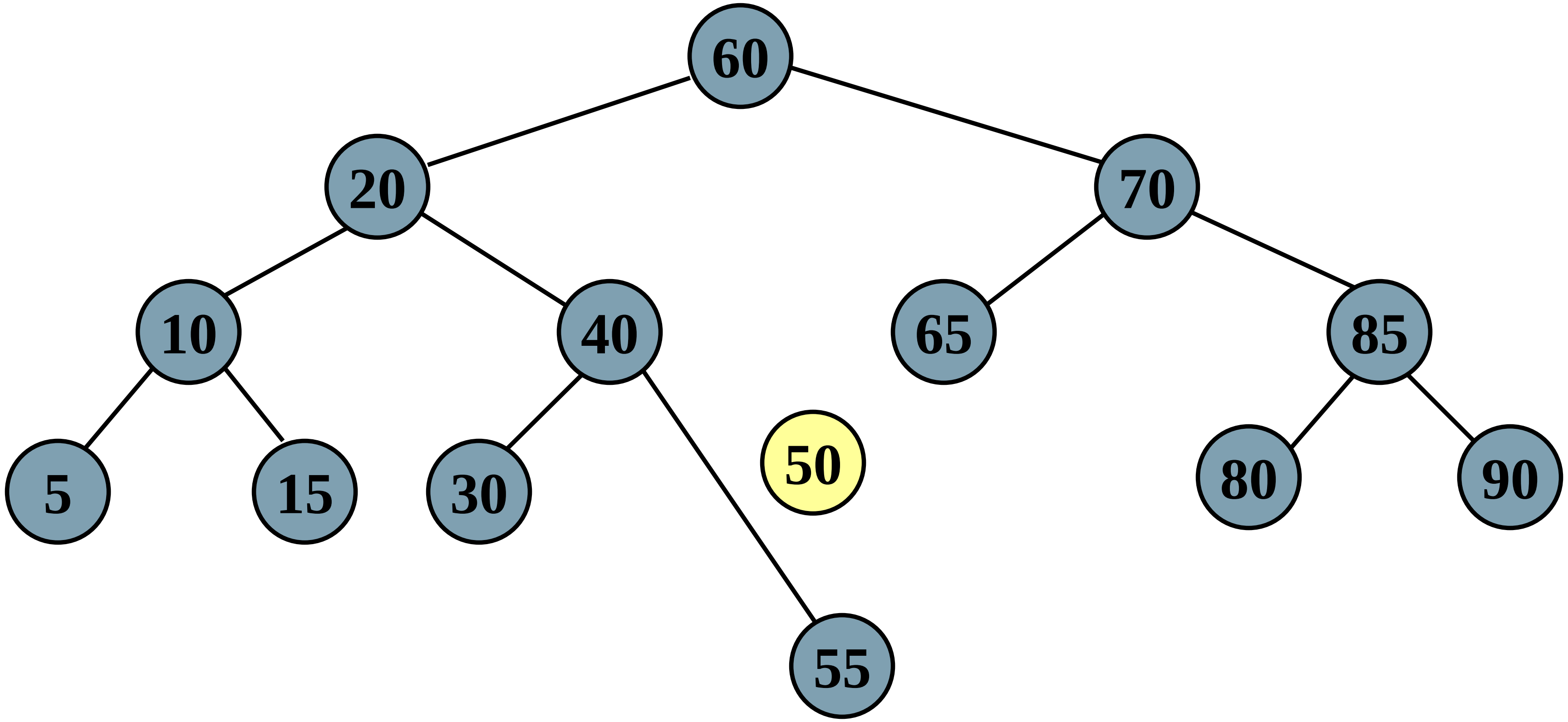
Delete 55 (case 1)



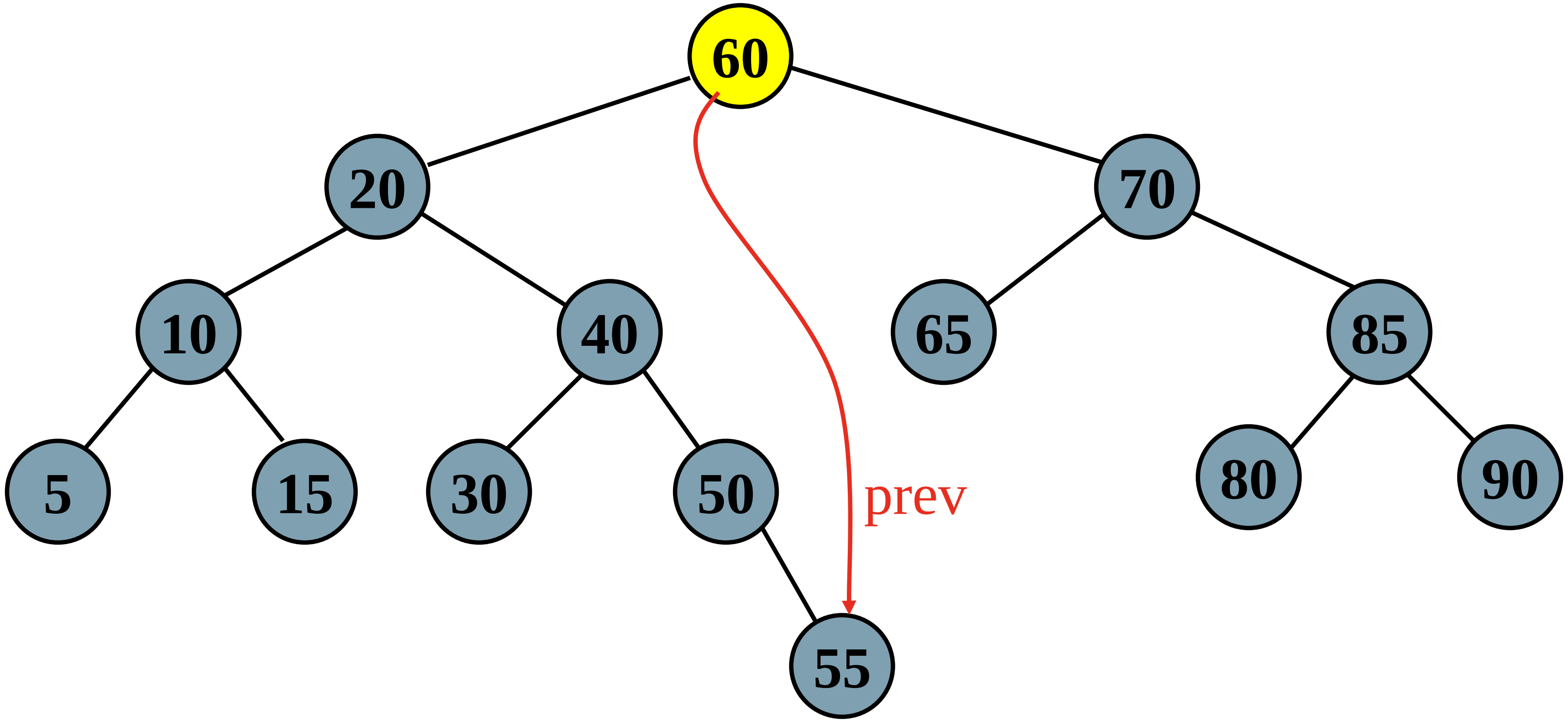
Delete 50 (case 2)



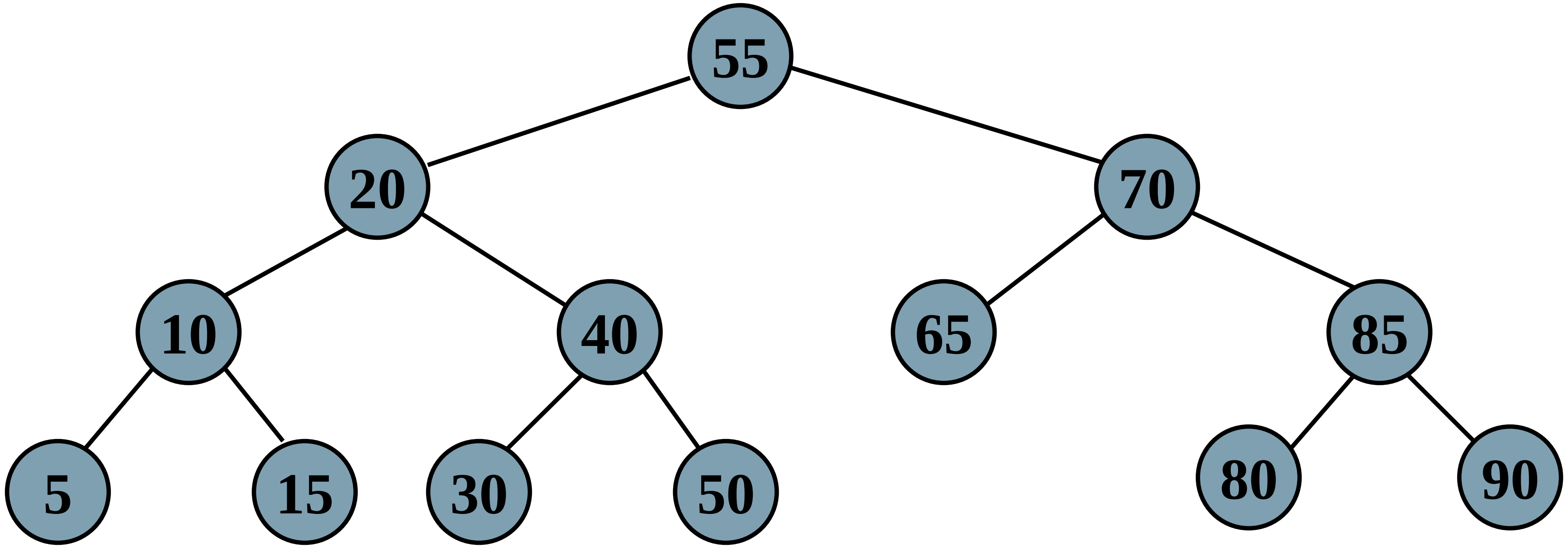
Delete 50 (case 2)



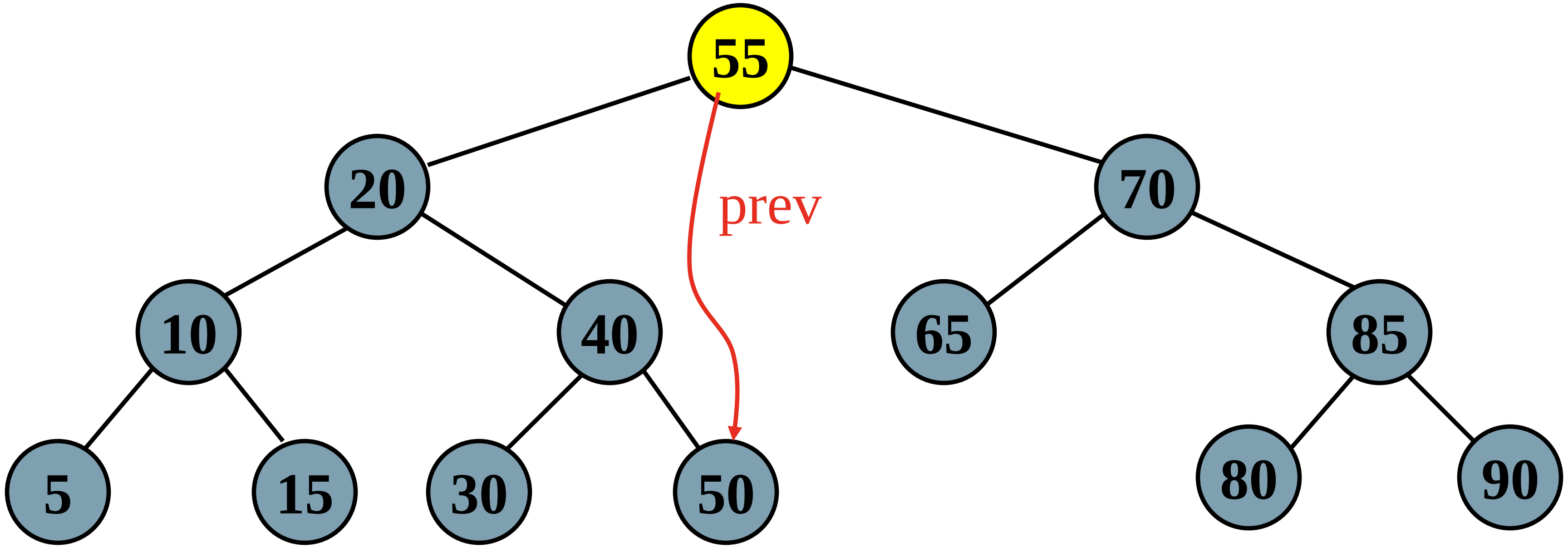
Delete 60 (case 3)



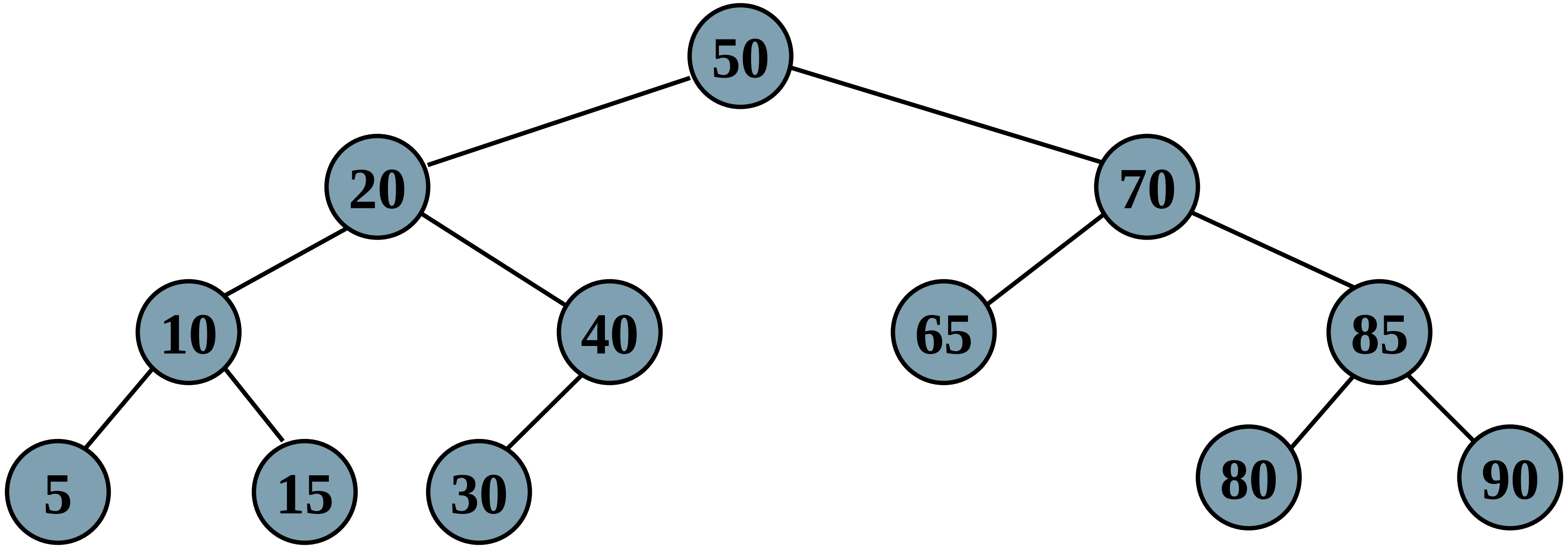
Delete 60 (case 3)



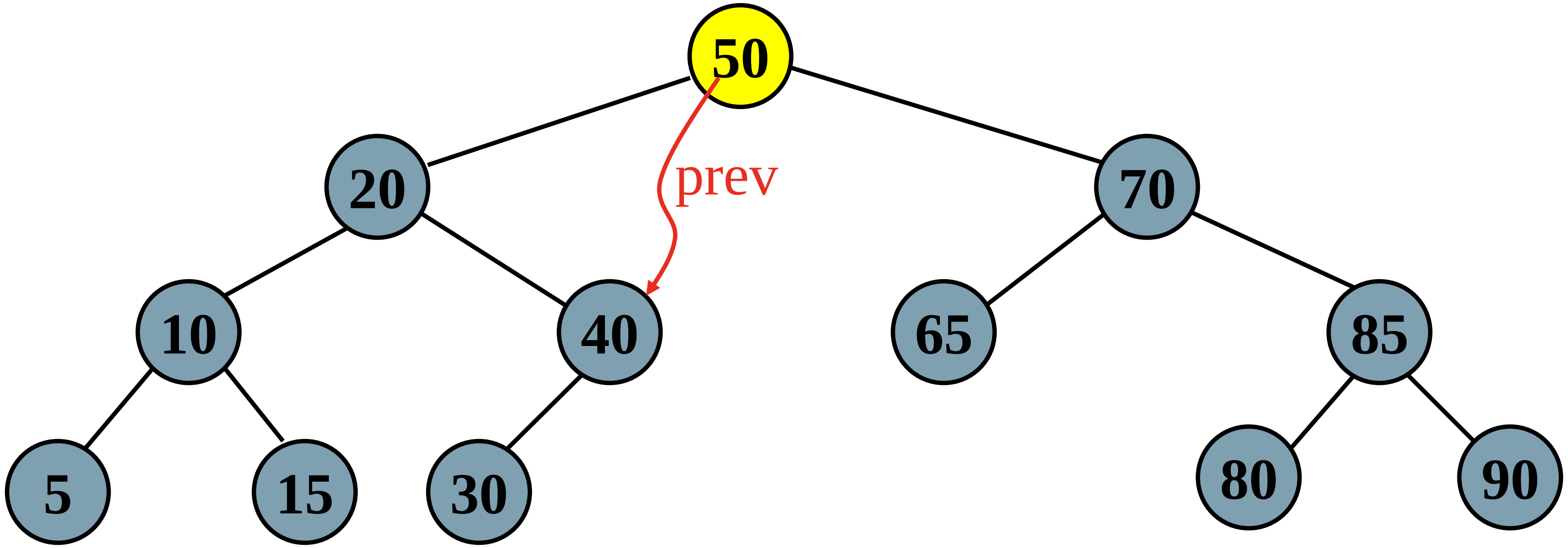
Delete 55 (case 3)



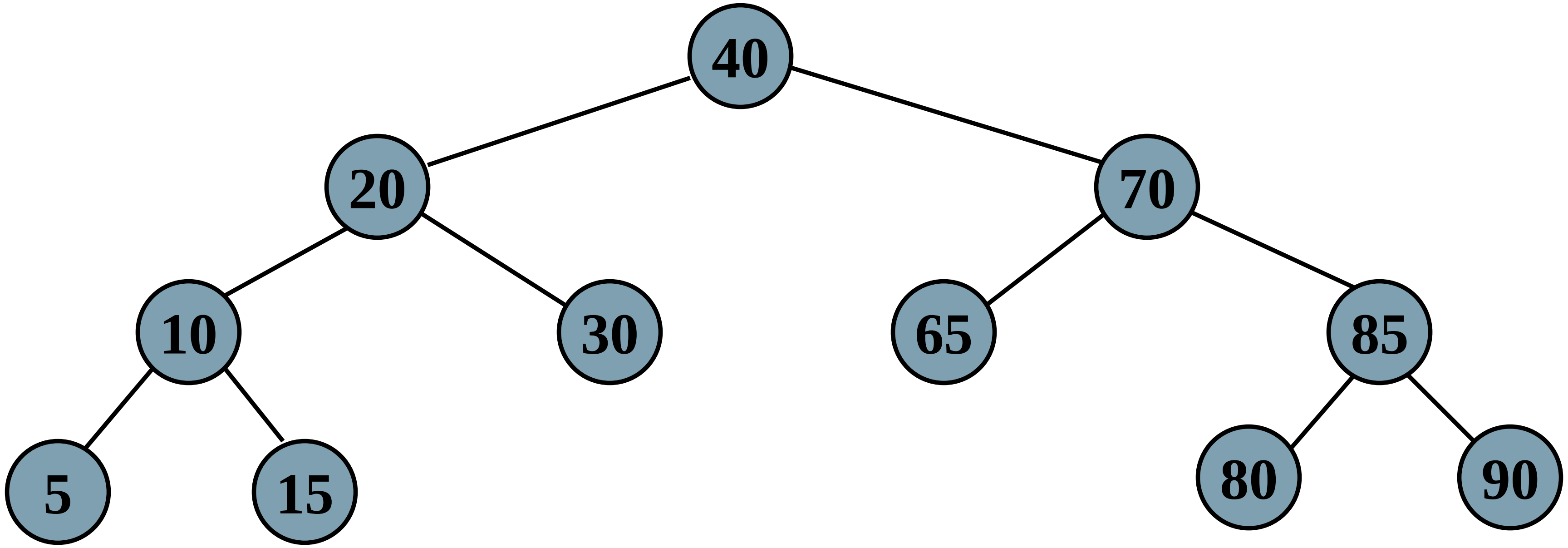
Delete 55 (case 3)



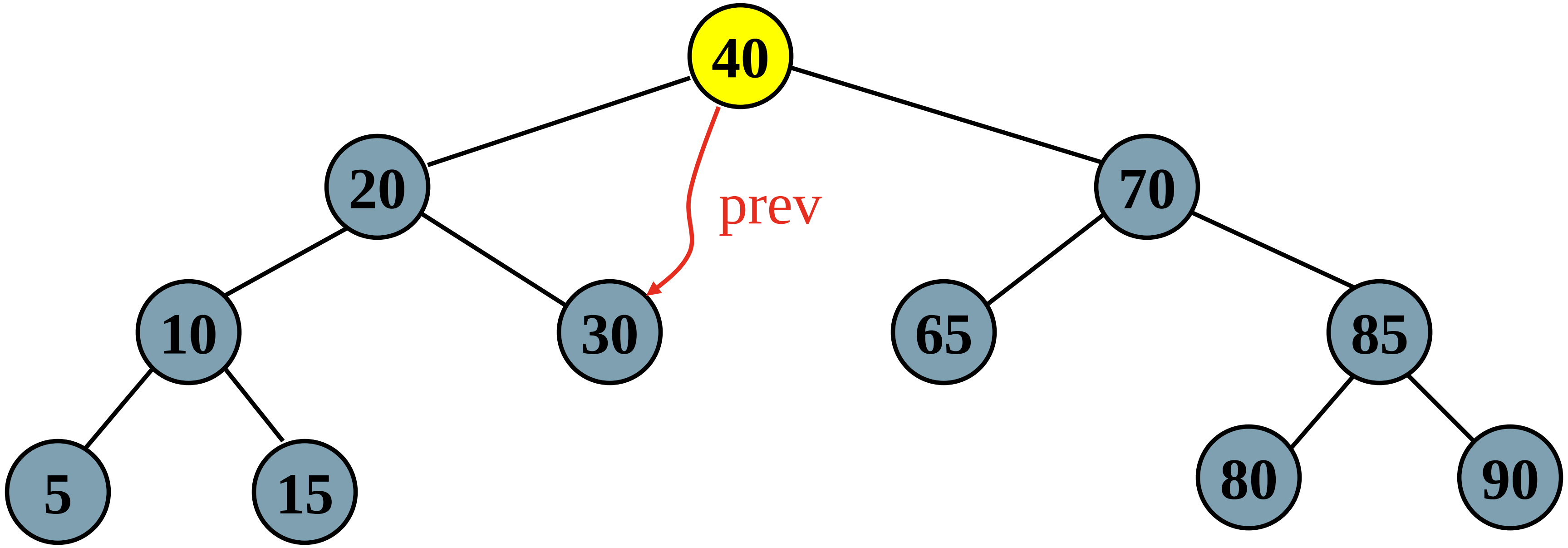
Delete 50 (case 3)



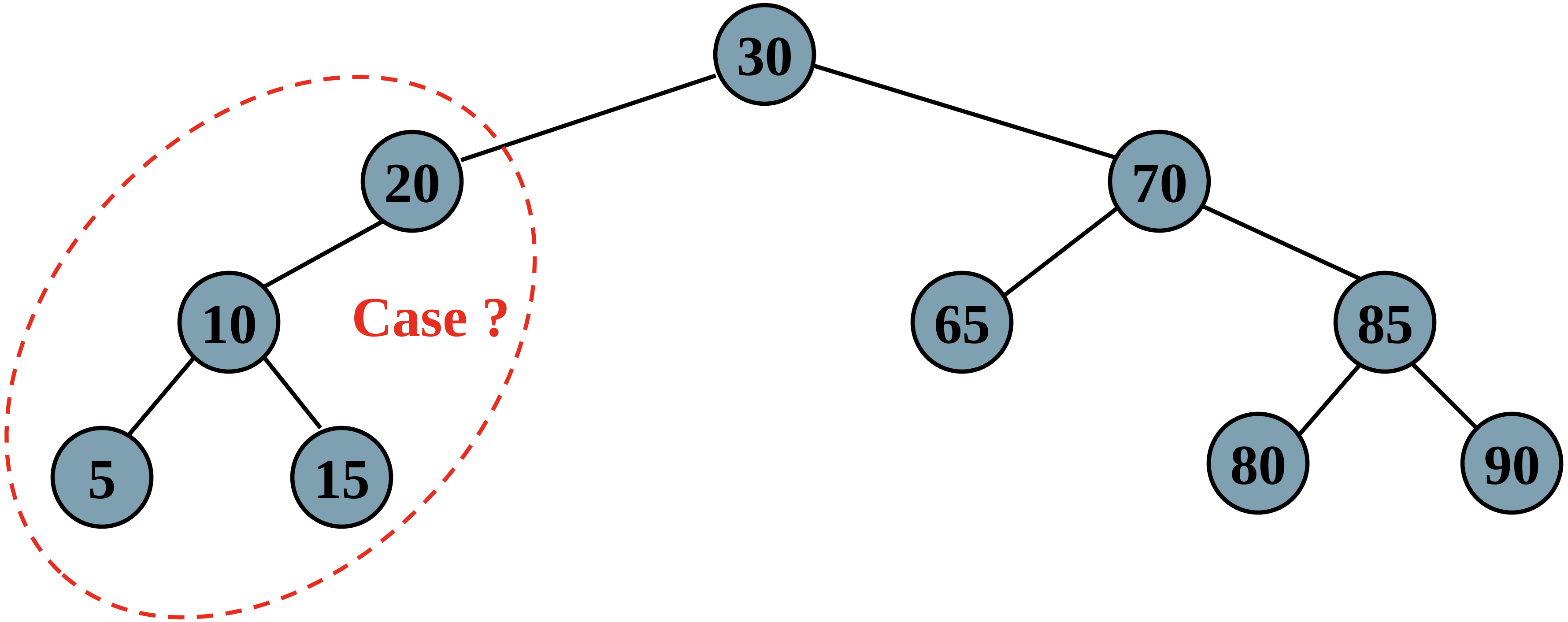
Delete 50 (case 3)



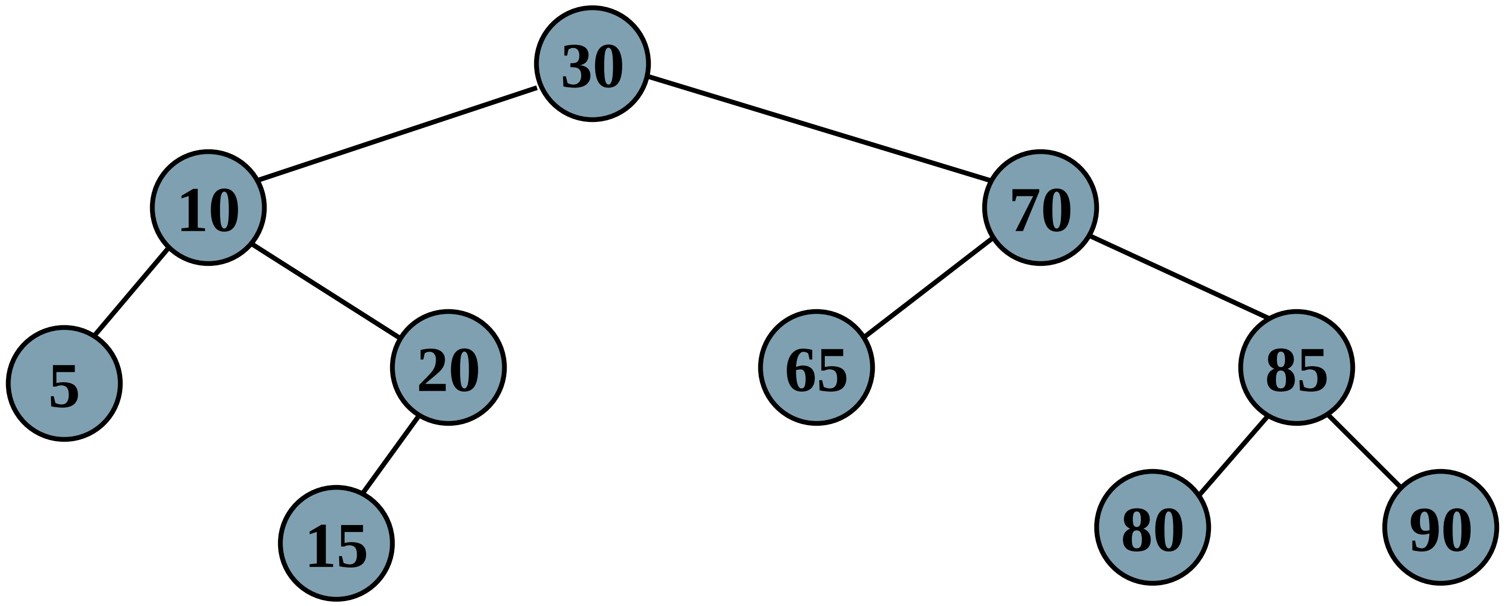
Delete 40 (case 3)



Delete 40 : Rebalancing



Delete 40: after rebalancing



Terima Kasih

Pertanyaan?