Algoritme dan Struktur Data

AVL Tree

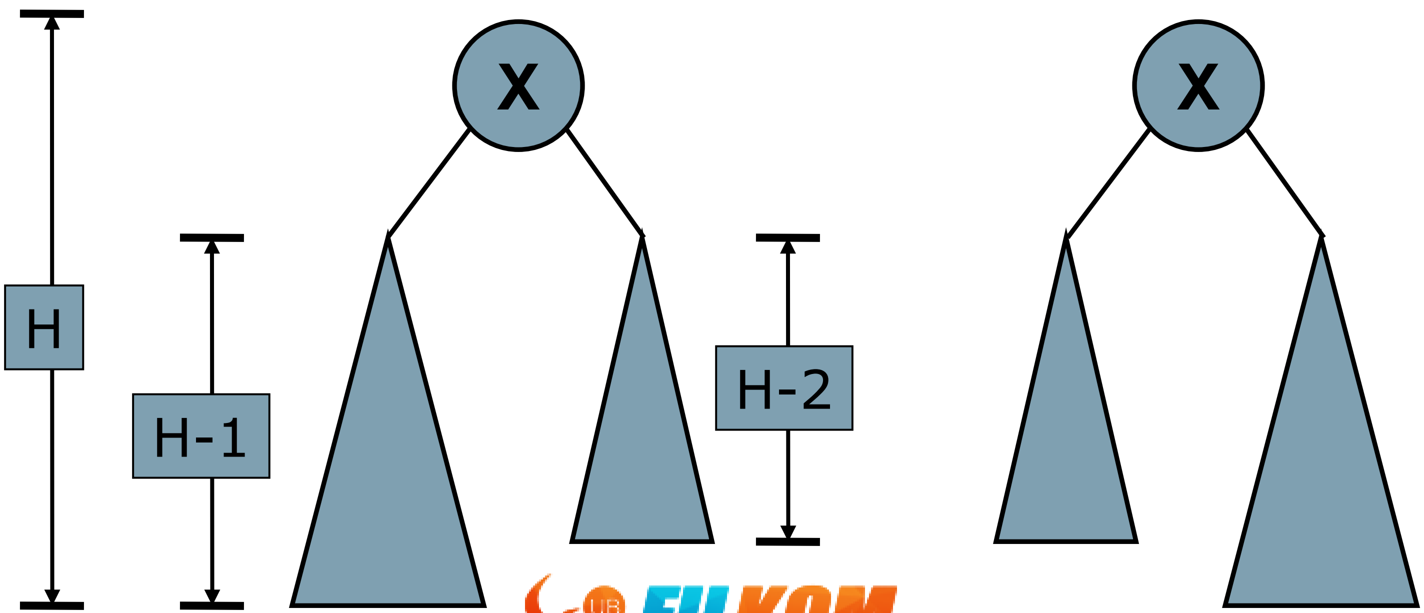
Putra Pandu Adikara
Fakultas Ilmu Komputer
Universitas Brawijaya

Tujuan

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

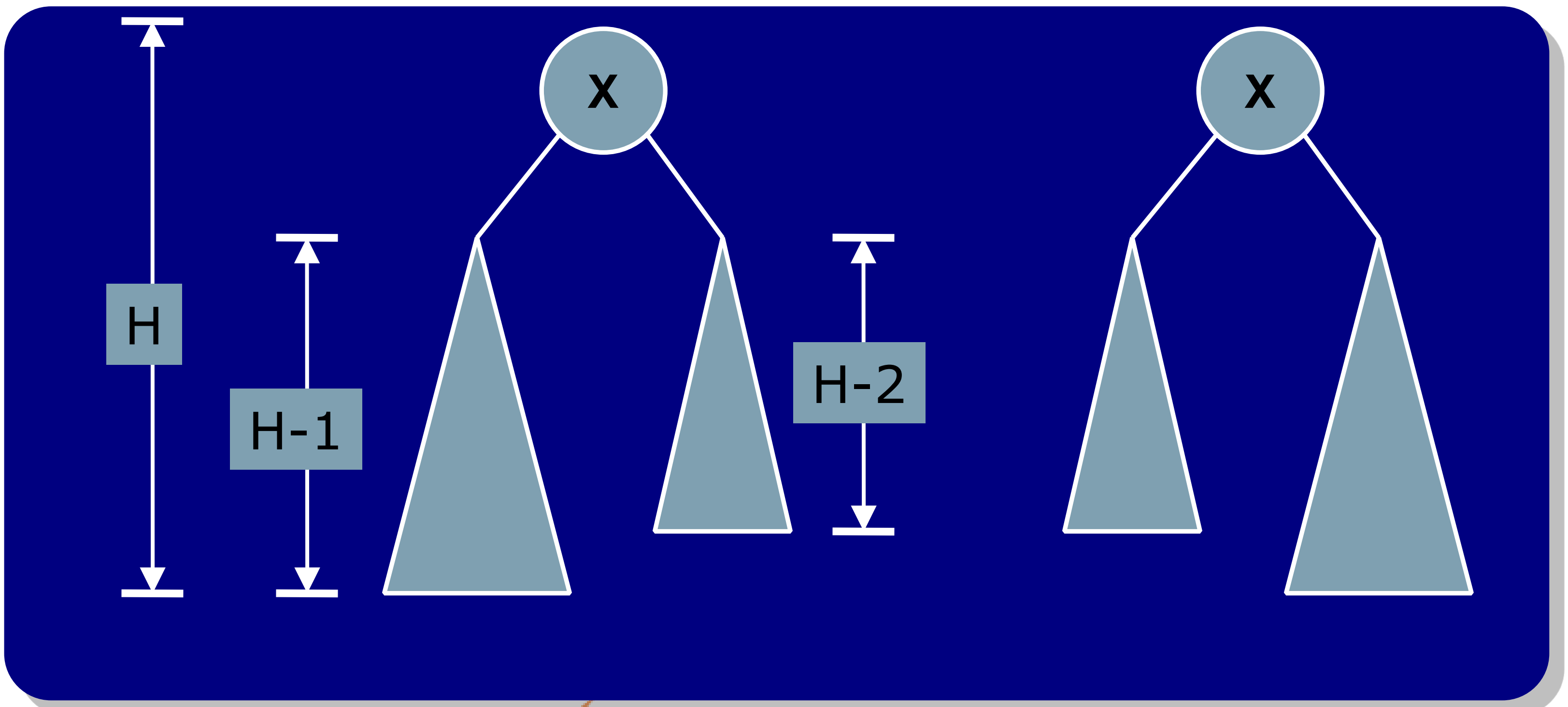
AVL Trees

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

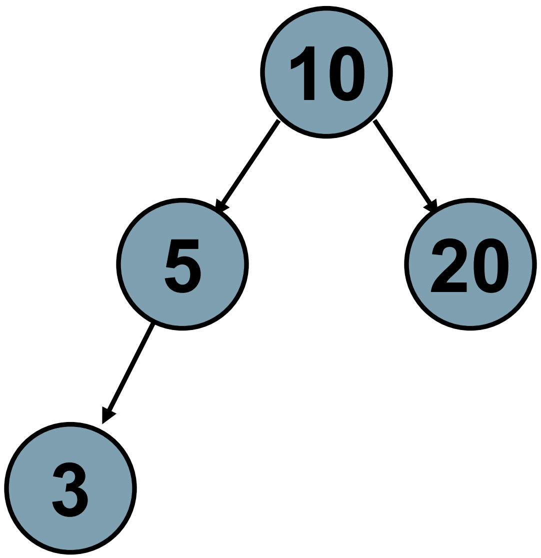


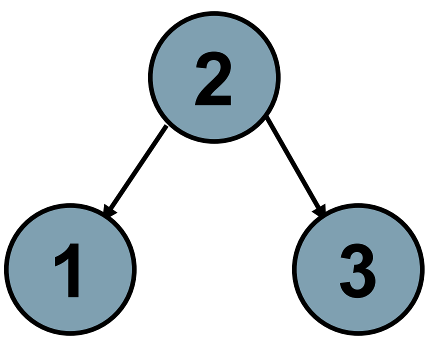
AVL Trees

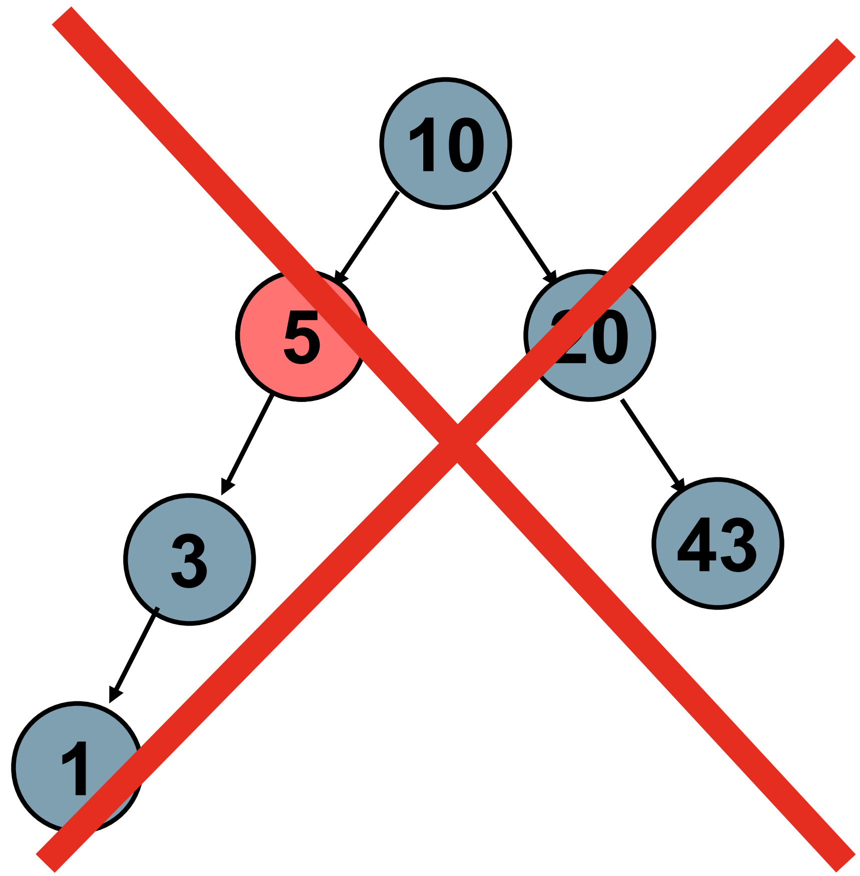
- Untuk setiap *node* dalam *tree*, ketinggian *subtree* di anak kiri dan *subtree* di anak kanan hanya berbeda maksimum 1.



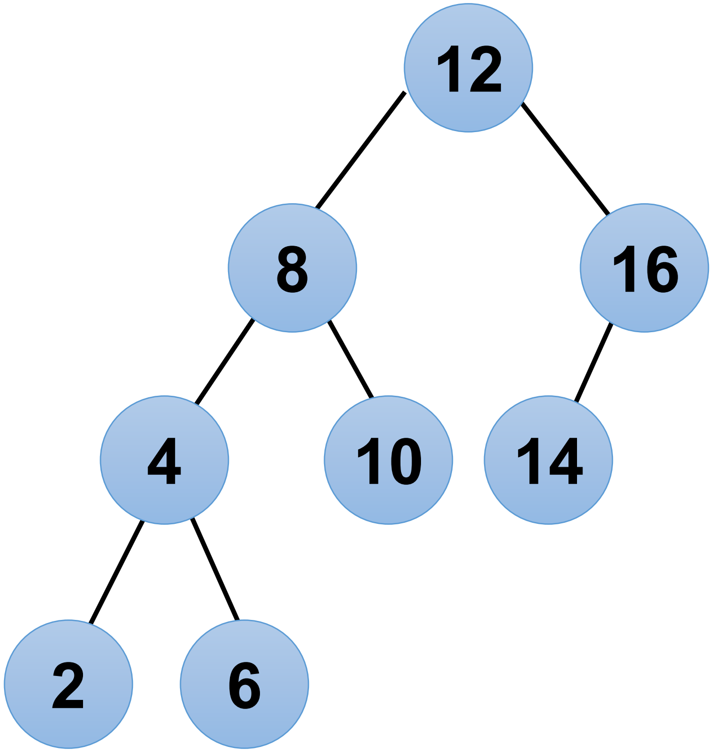
AVL Tree





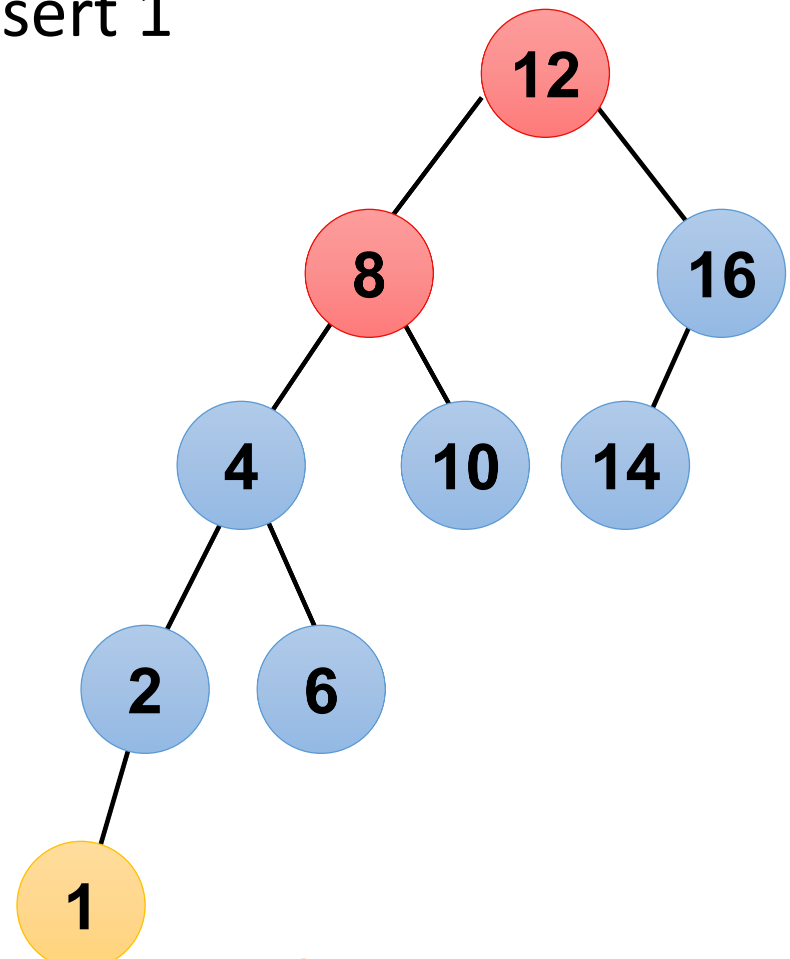


AVL Tree



Penyisipan node di AVL Tree

- Setelah insert 1



Insertion pada AVL Tree

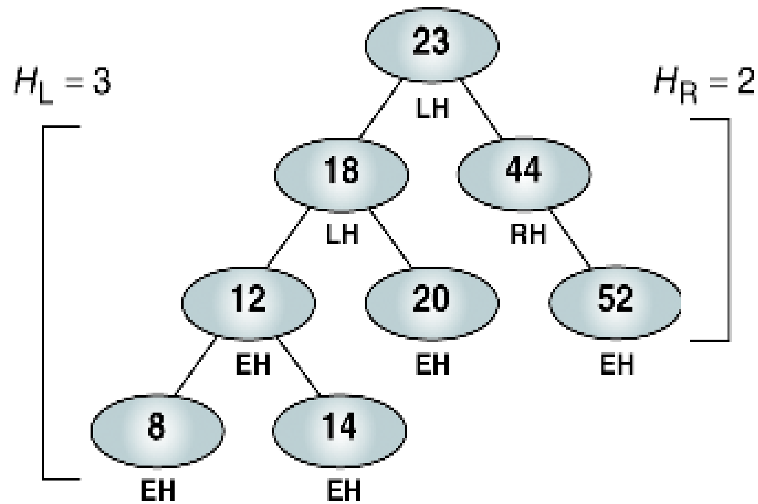
- Untuk menjamin kondisi *balance* pada AVL tree, setelah penambahan sebuah *node* jalur dari *node* baru tersebut hingga *root* disimpan dan diperiksa kondisi *balance* pada tiap *node*-nya.
- Node pertama yang memiliki |balance factor| > 1 diseimbangkan
- Jika setelah penambahan, kondisi *balance* tidak terpenuhi pada *node* tertentu, maka lakukan salah satu rotasi berikut:
  - Single rotation
  - Double rotation

AVL Tree Balance Factor

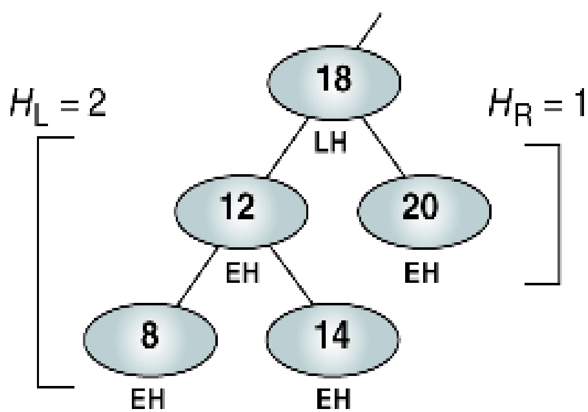
- Untuk efisiensi:

- 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.

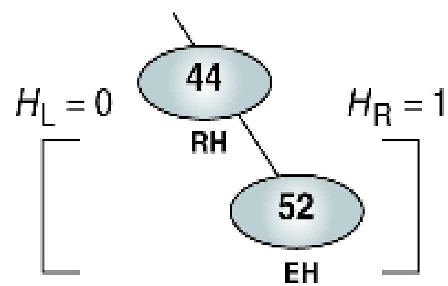
Kondisi Balance (AVL Tree)



(a) Tree 23 appears balanced: $H_L - H_R = 1$

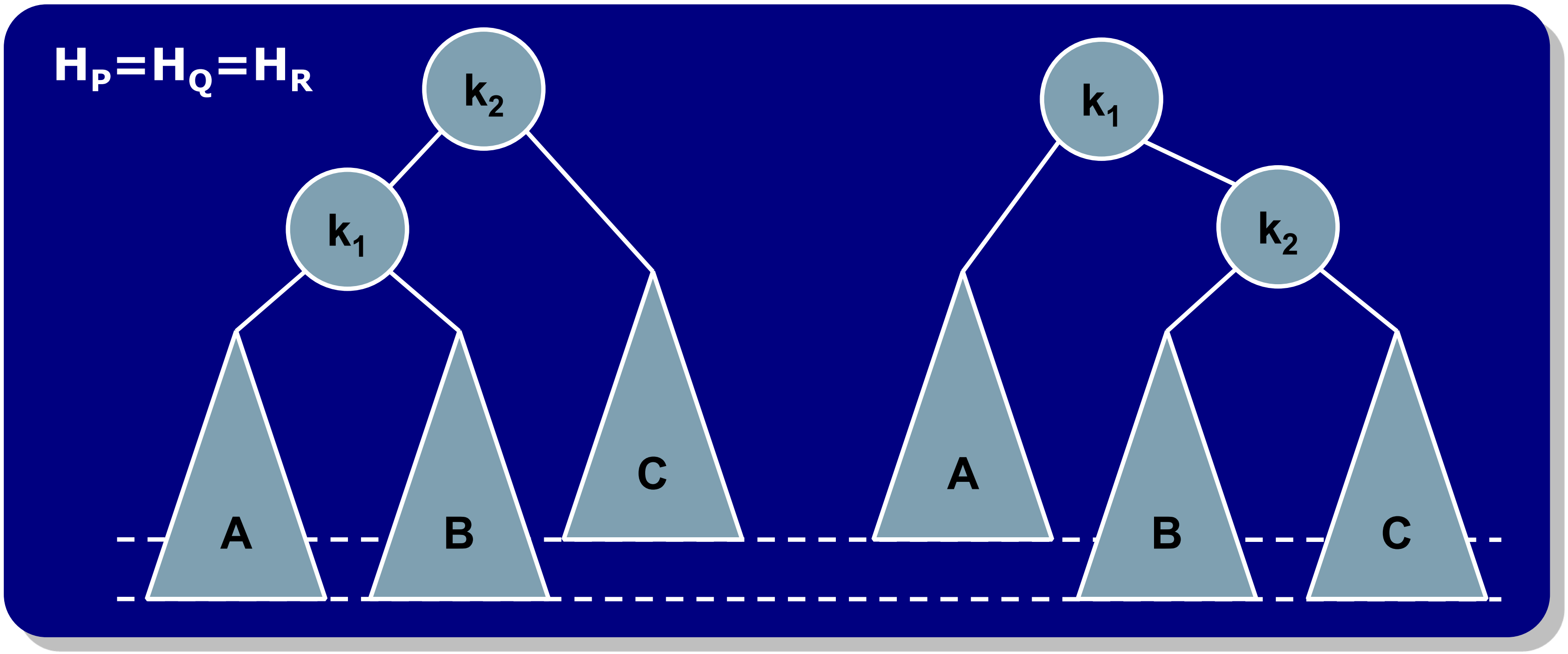


(b) Subtree 18 appears balanced: $H_L - H_R = 1$



(c) Subtree 44 is balanced: $|H_L - H_R| = 1$

Kondisi tidak balance



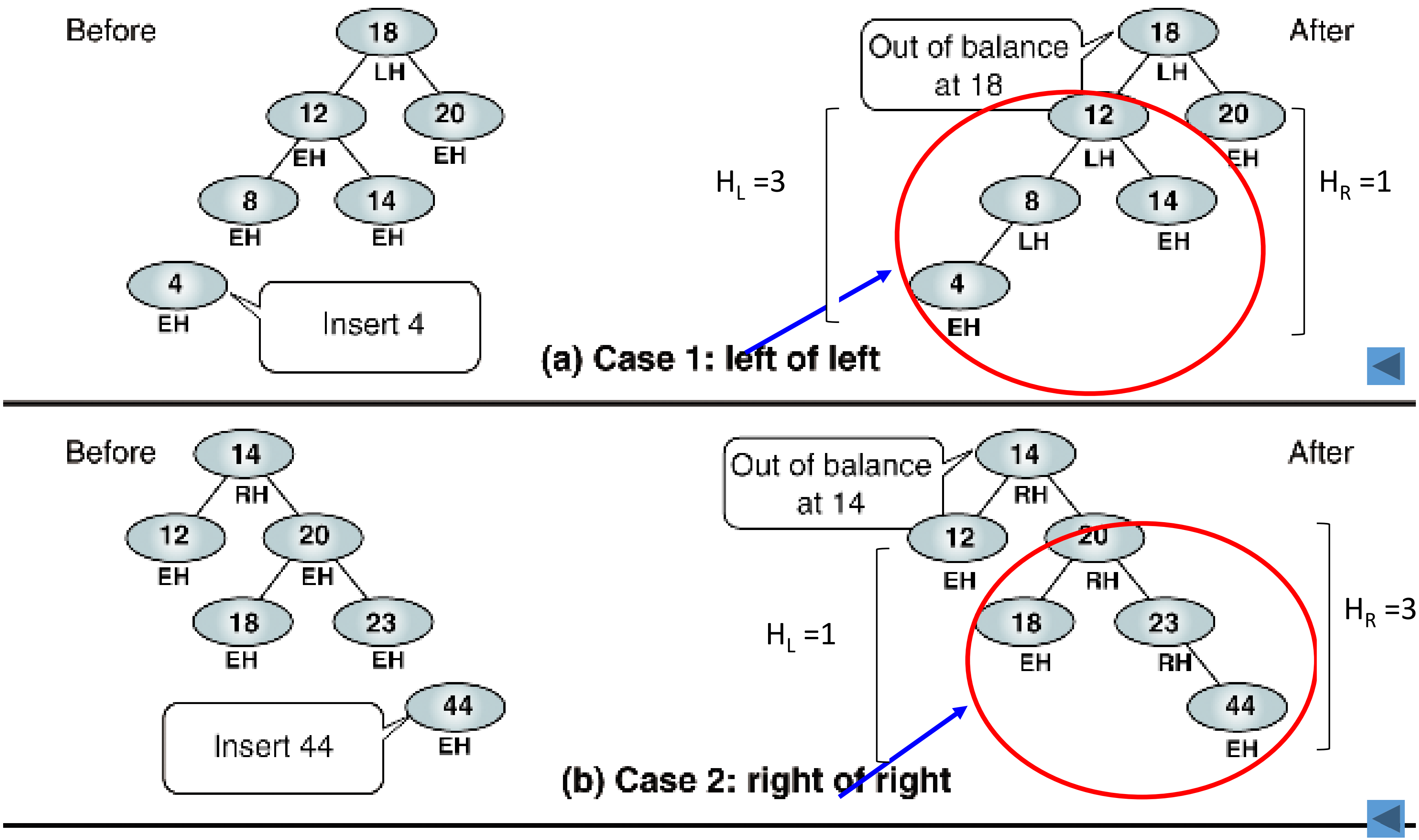
- Sebuah penambahan pada subtree:
  - A (outside) - case 1
  - B (inside) - case 3
- Sebuah penambahan pada subtree:
  - B (inside) - case 2
  - C (outside) - case 4

Menyeimbangkan AVL Tree

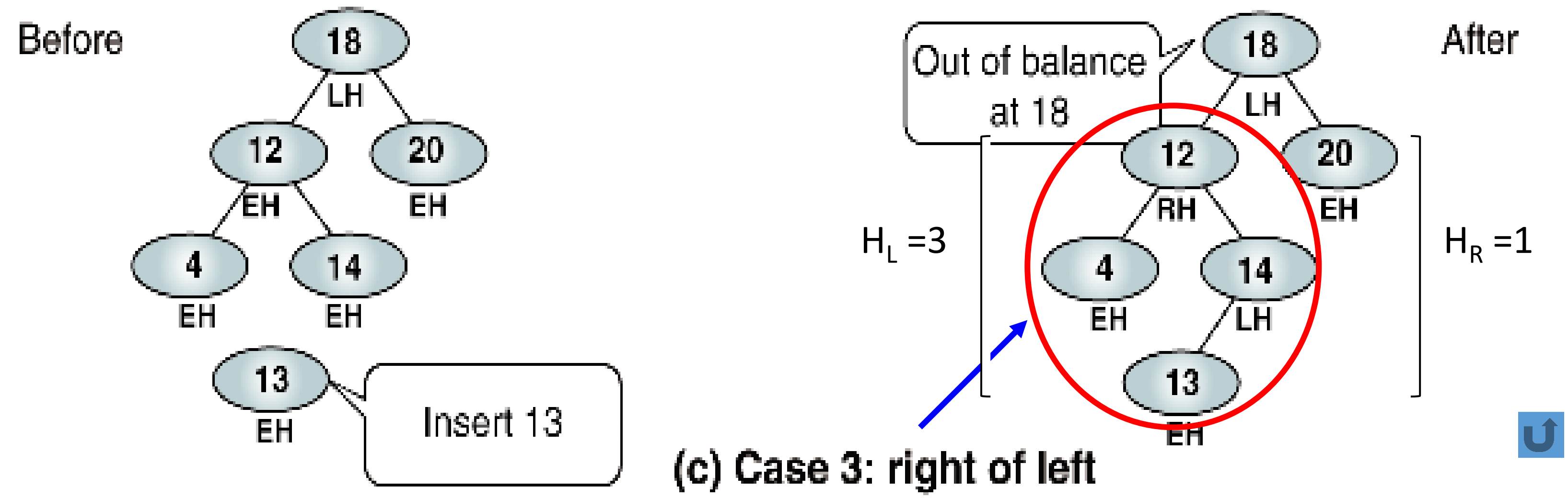
- AVL trees diseimbangkan dengan merotasikan node ke kiri atau ke kanan
- Kasus penyeimbangan pada sebuah node:

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.

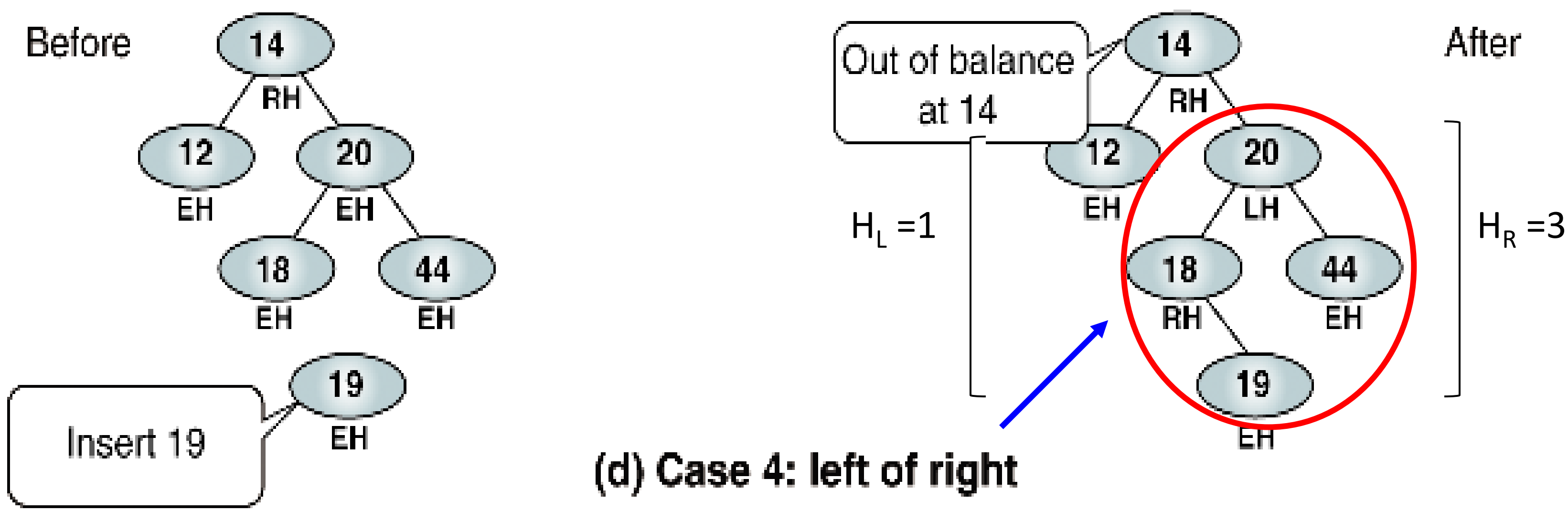
Out-of-Balance AVL Trees



Out-of-Balance AVL Trees (Cont.)

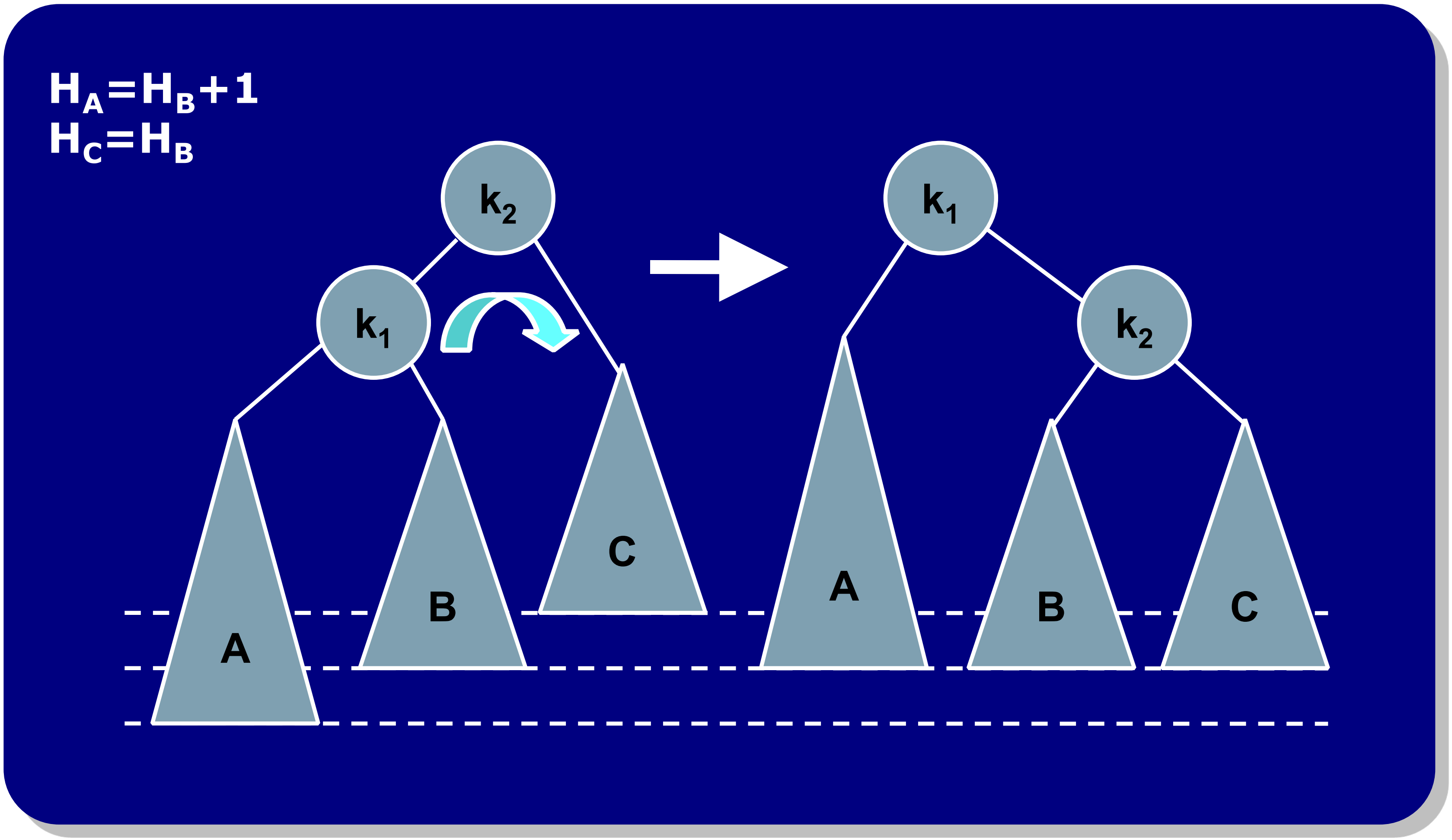


(c) Case 3: right of left

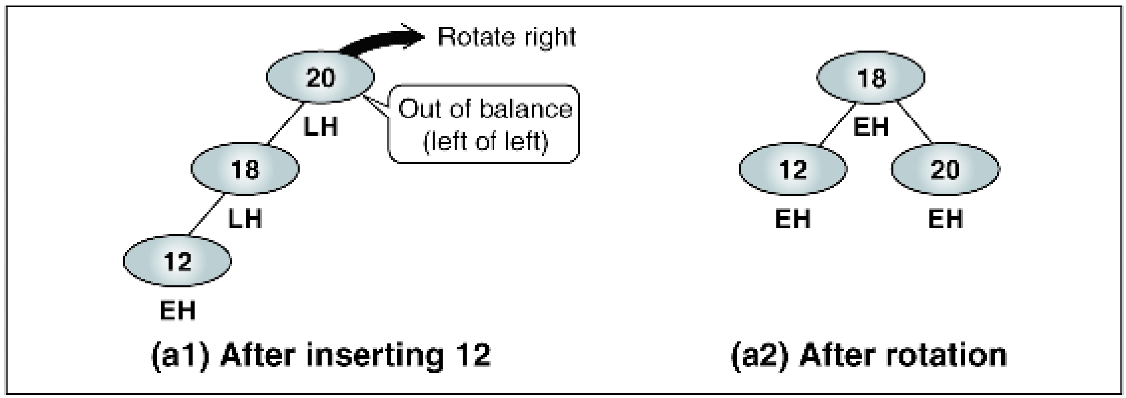


(d) Case 4: left of right

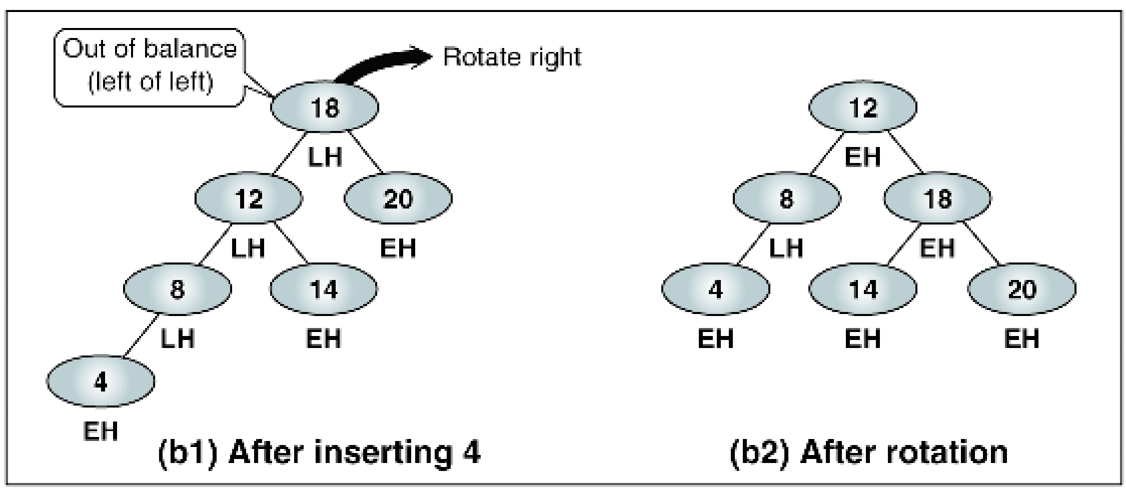
Single Rotation (case 1)



Case 1: Left of Left – Single Rotation Right

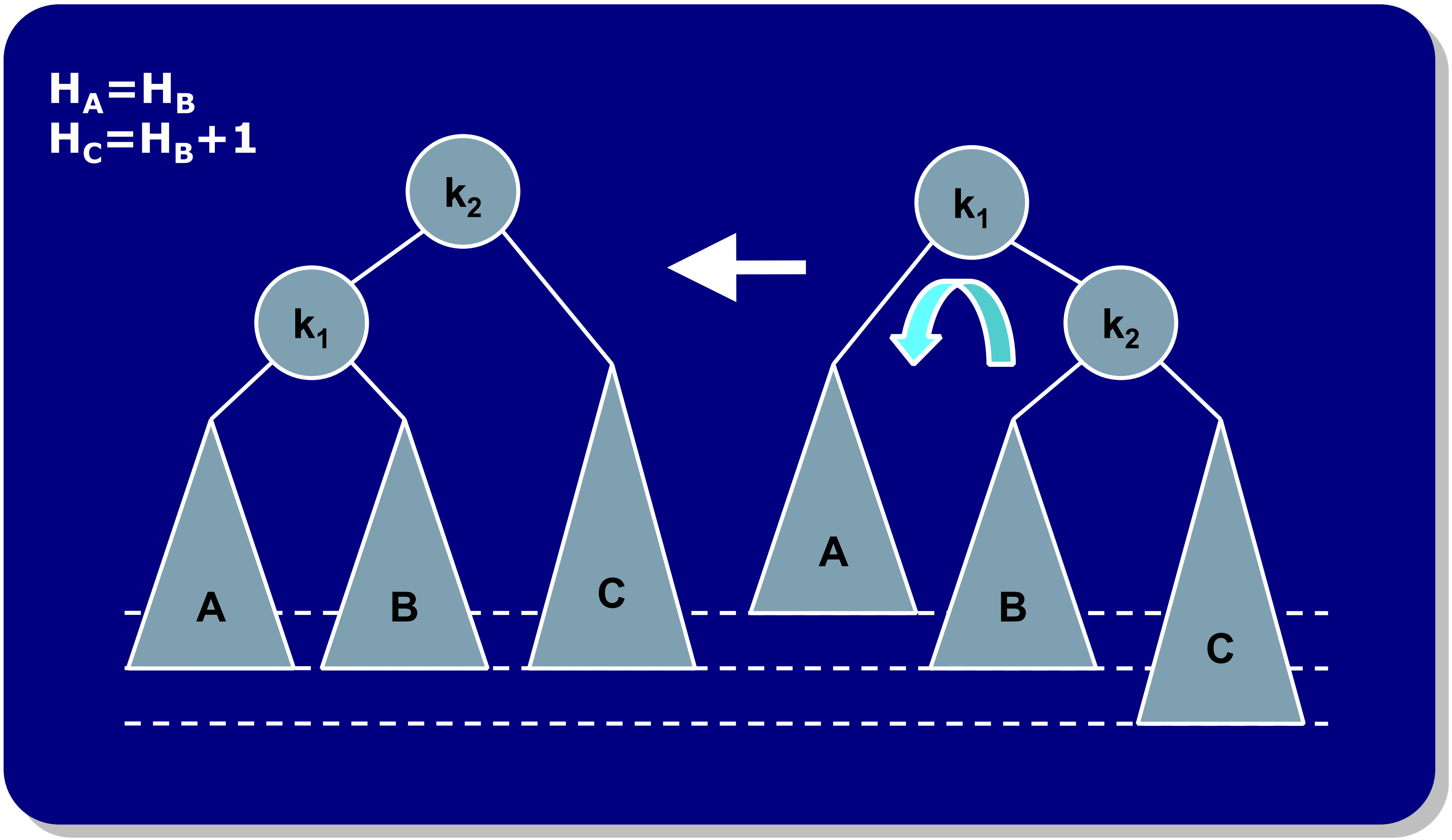


(a) Simple right rotation

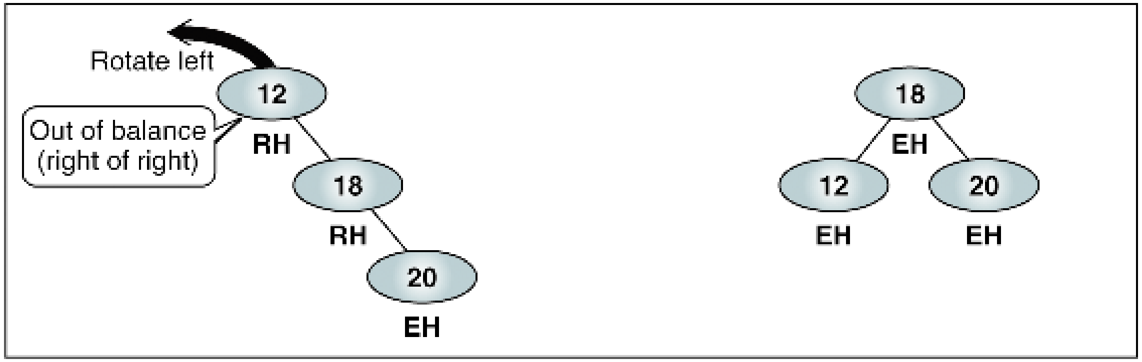


(b) Complex right rotation

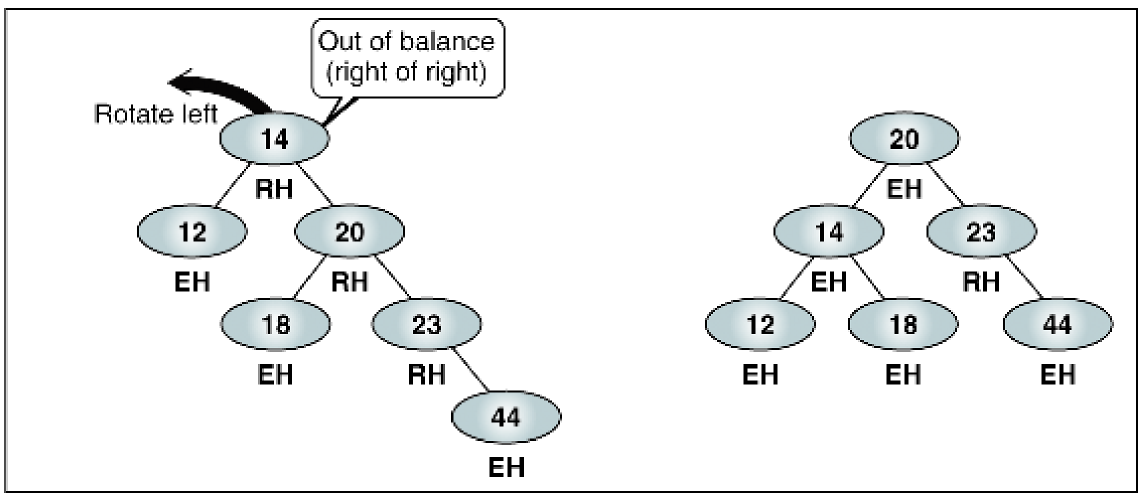
Single Rotation (case 2)



Case 2: Right of Right – Single Rotation Left



(a) Simple left rotation



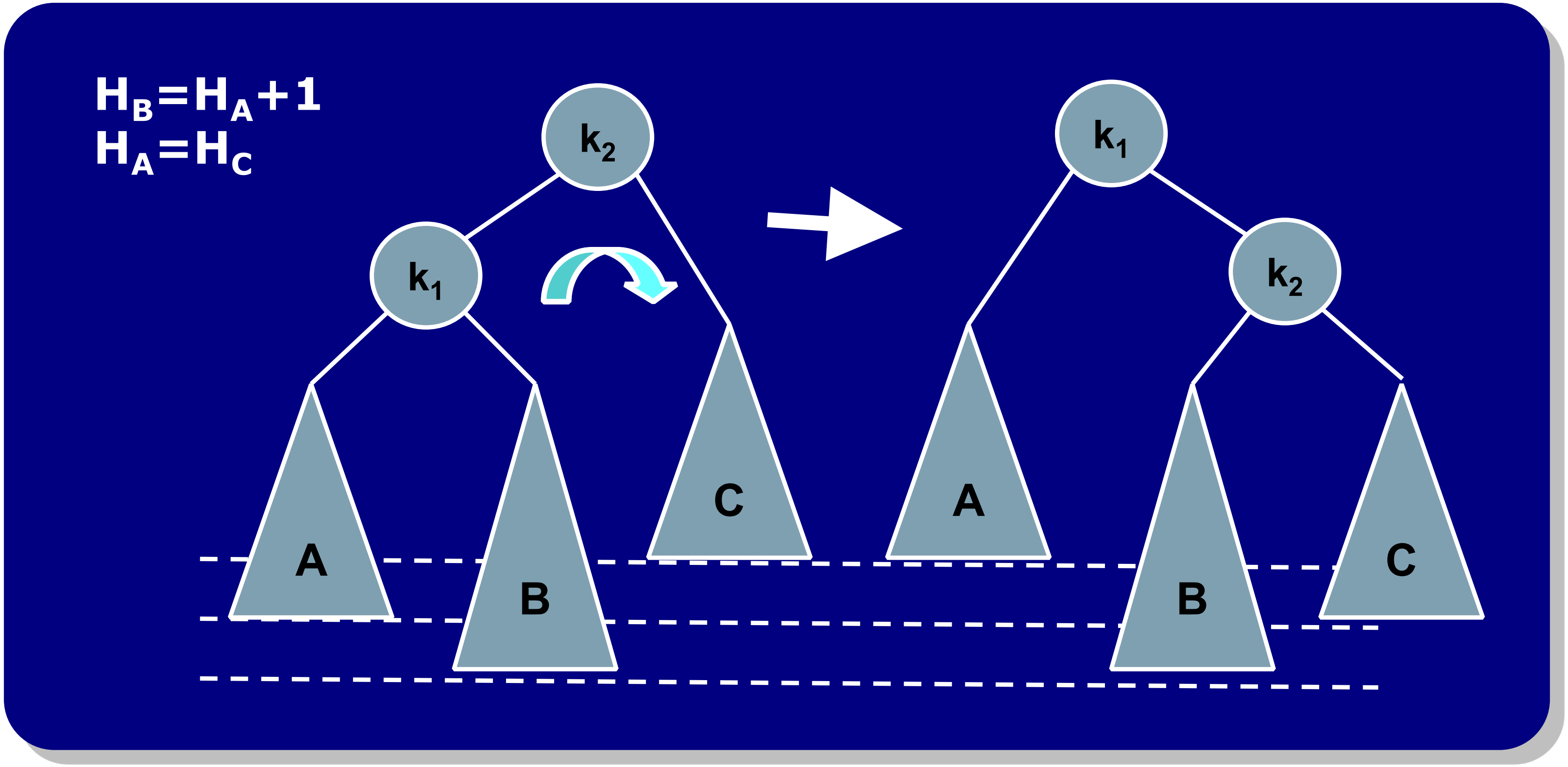
(b) Complex left rotation

Code: Single Rotation

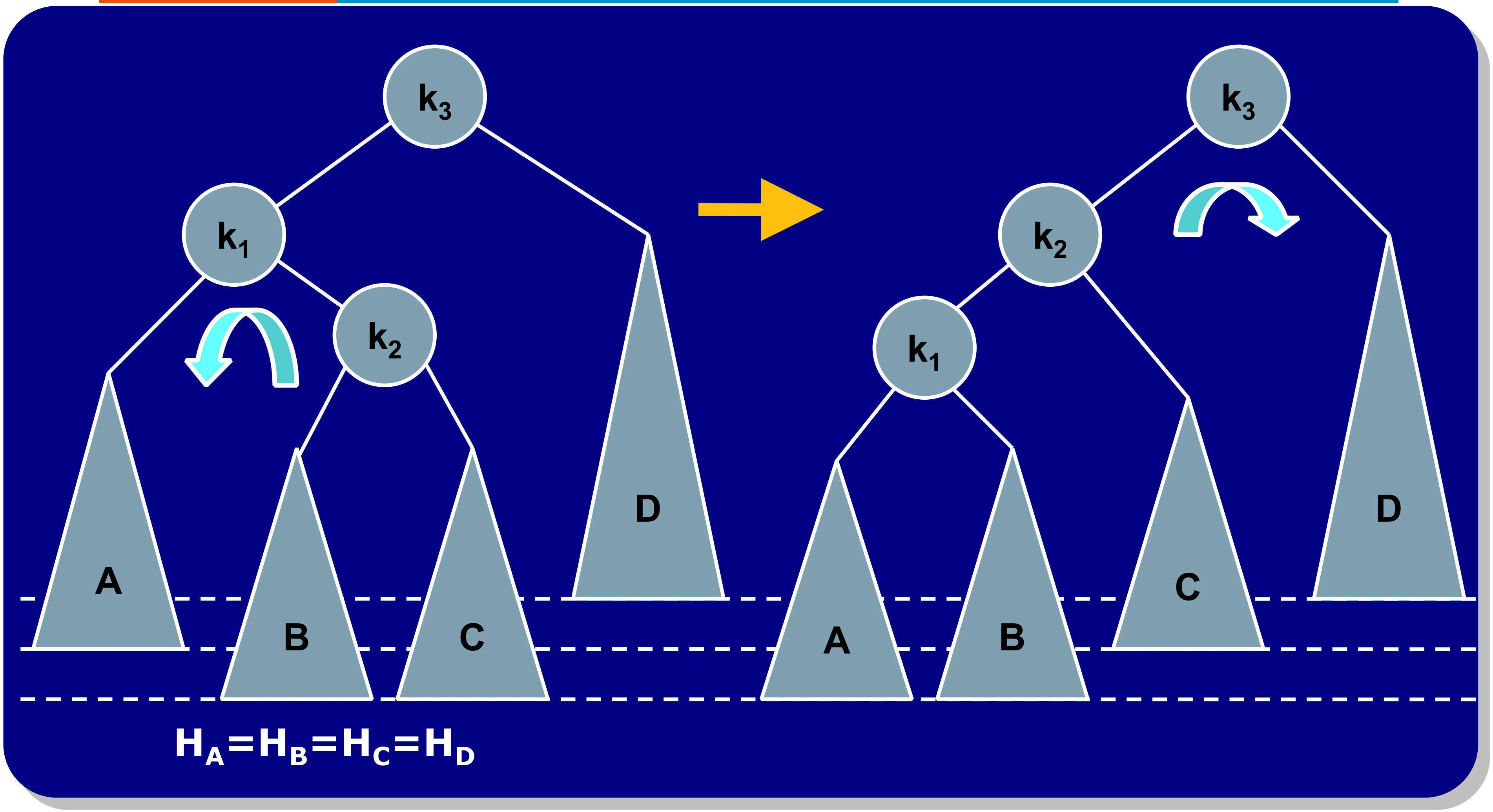
```
static < A extends Comparable <A>> AvlNode <A>
   singleRotateWithLeftChild(AvlNode <A> k2) {
      AvlNode <A> k1 = k2.left;
      k2.left = k1.right;
      k1.right = k2; // update tinggi kedua node. return k1;
}
```

Keterbatasan Single Rotation

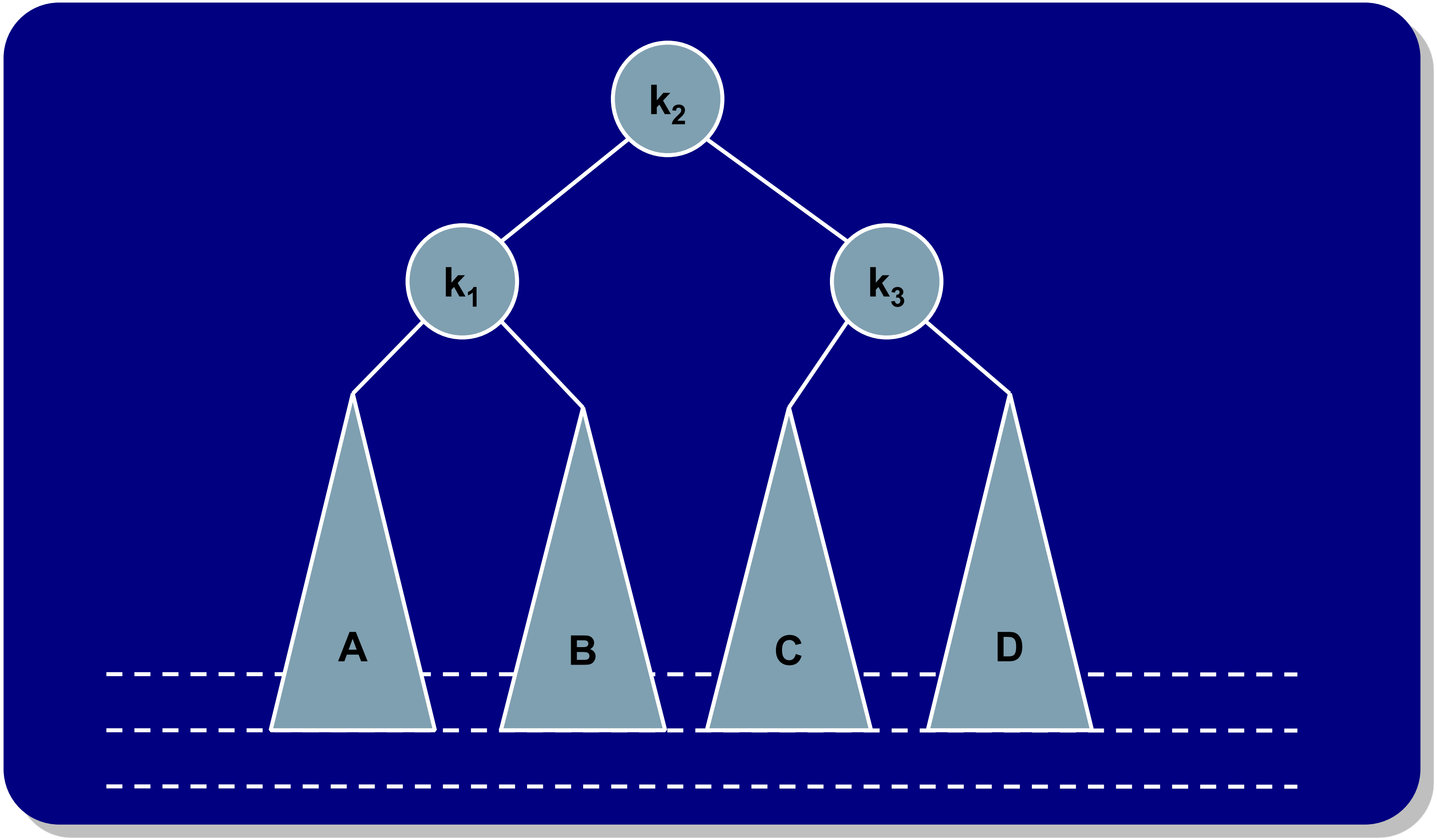
- Single rotation tidak bisa digunakan untuk kasus 3 dan 4 (inside case)



Double Rotation: Langkah



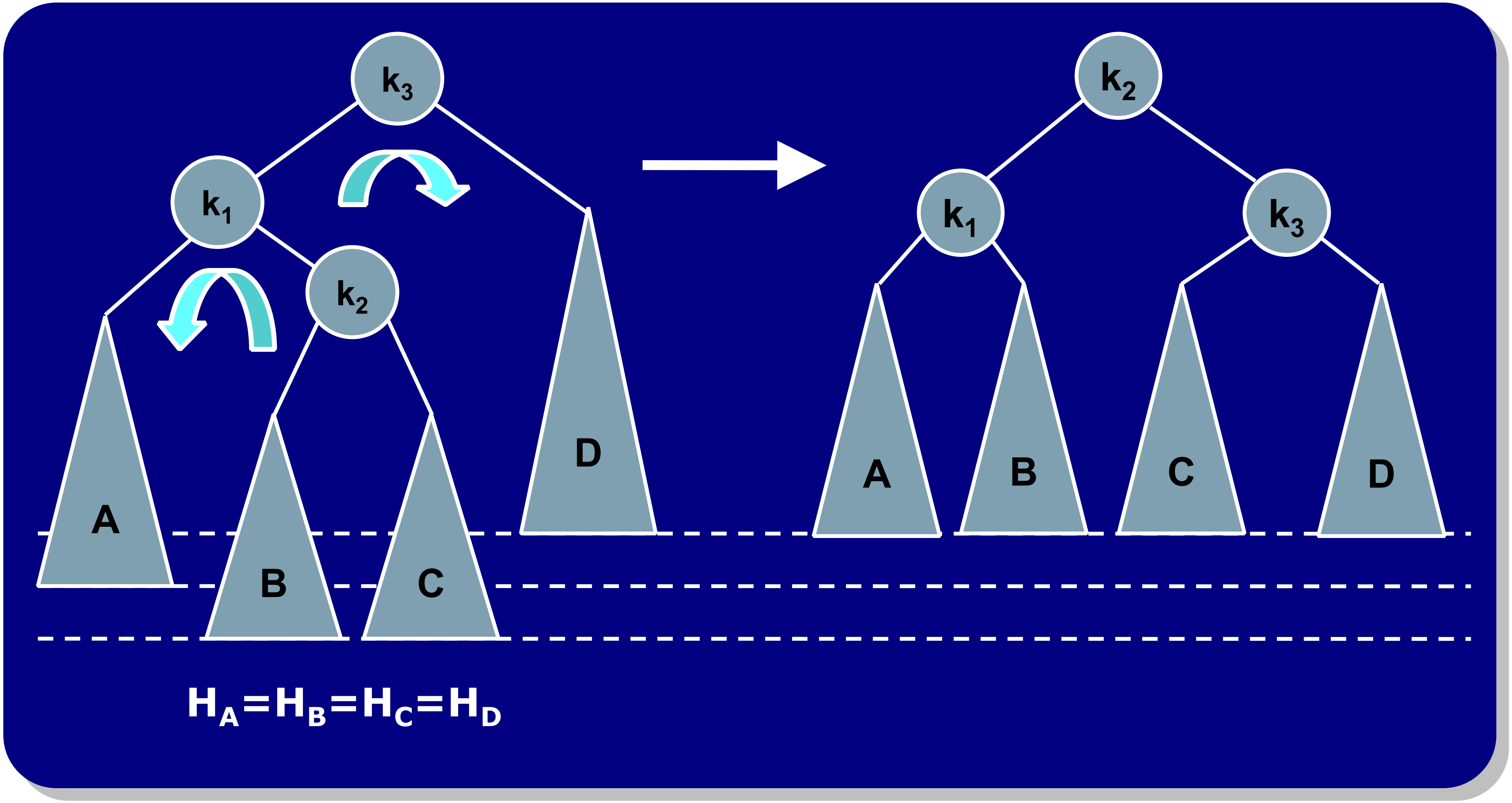
Double Rotation: Langkah



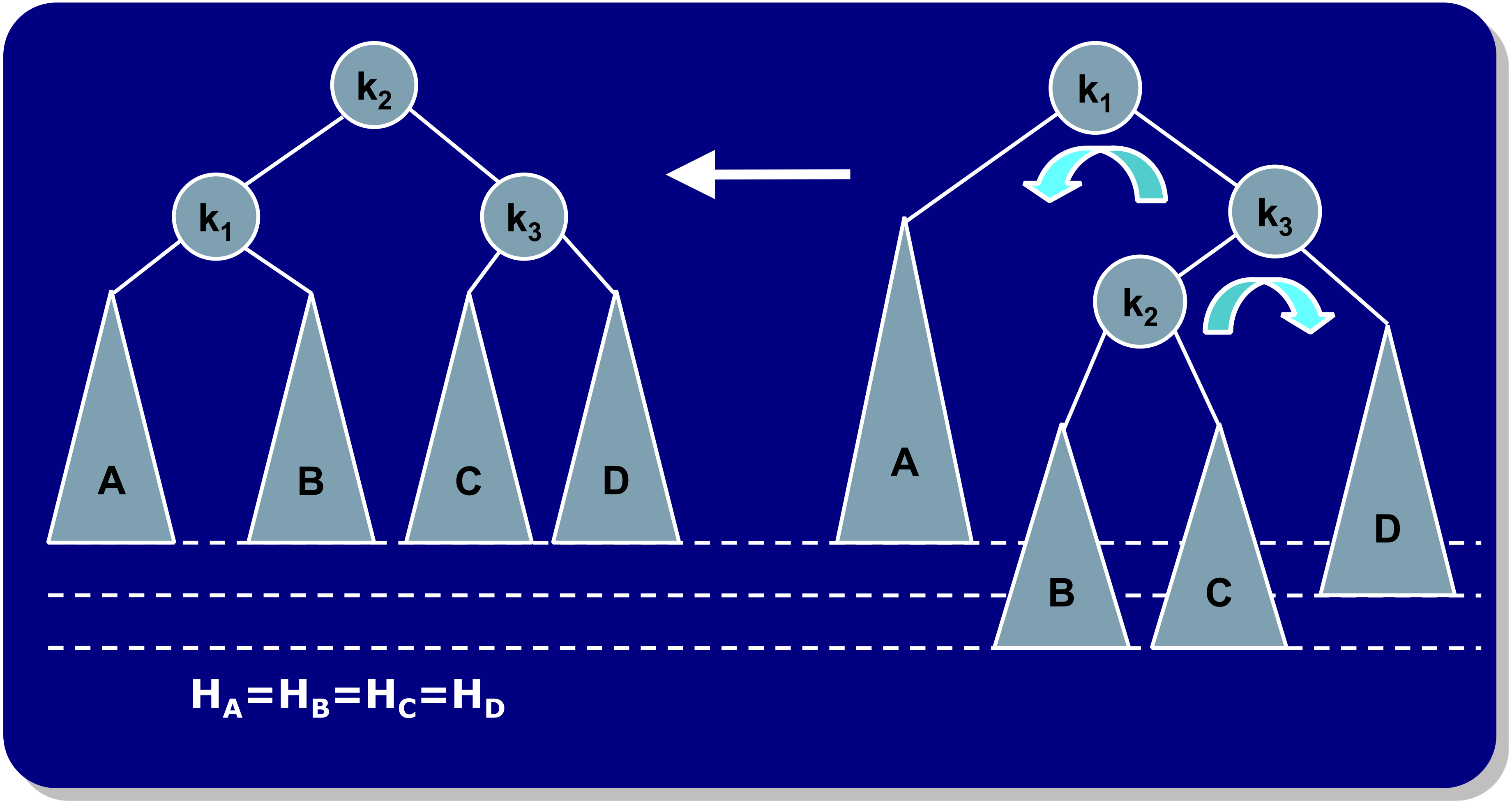
Code: Double Rotation

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

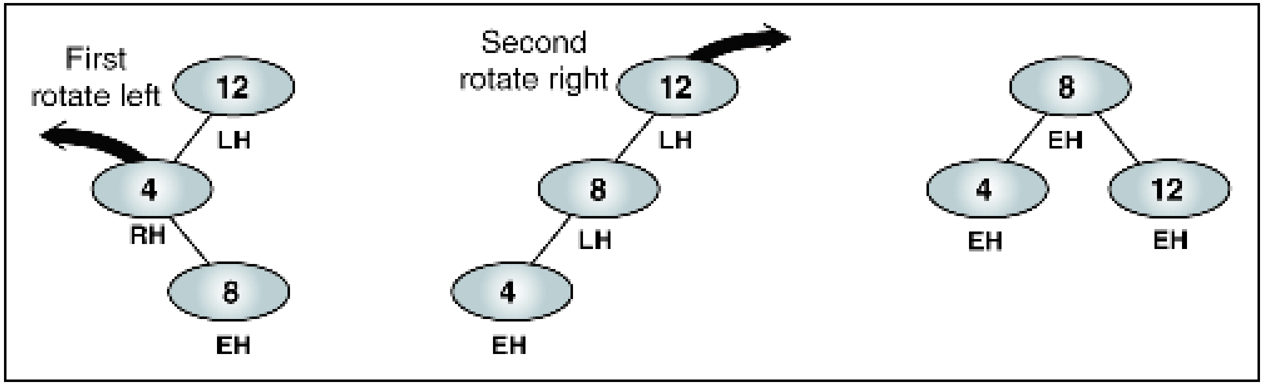
Double Rotation



Double Rotation

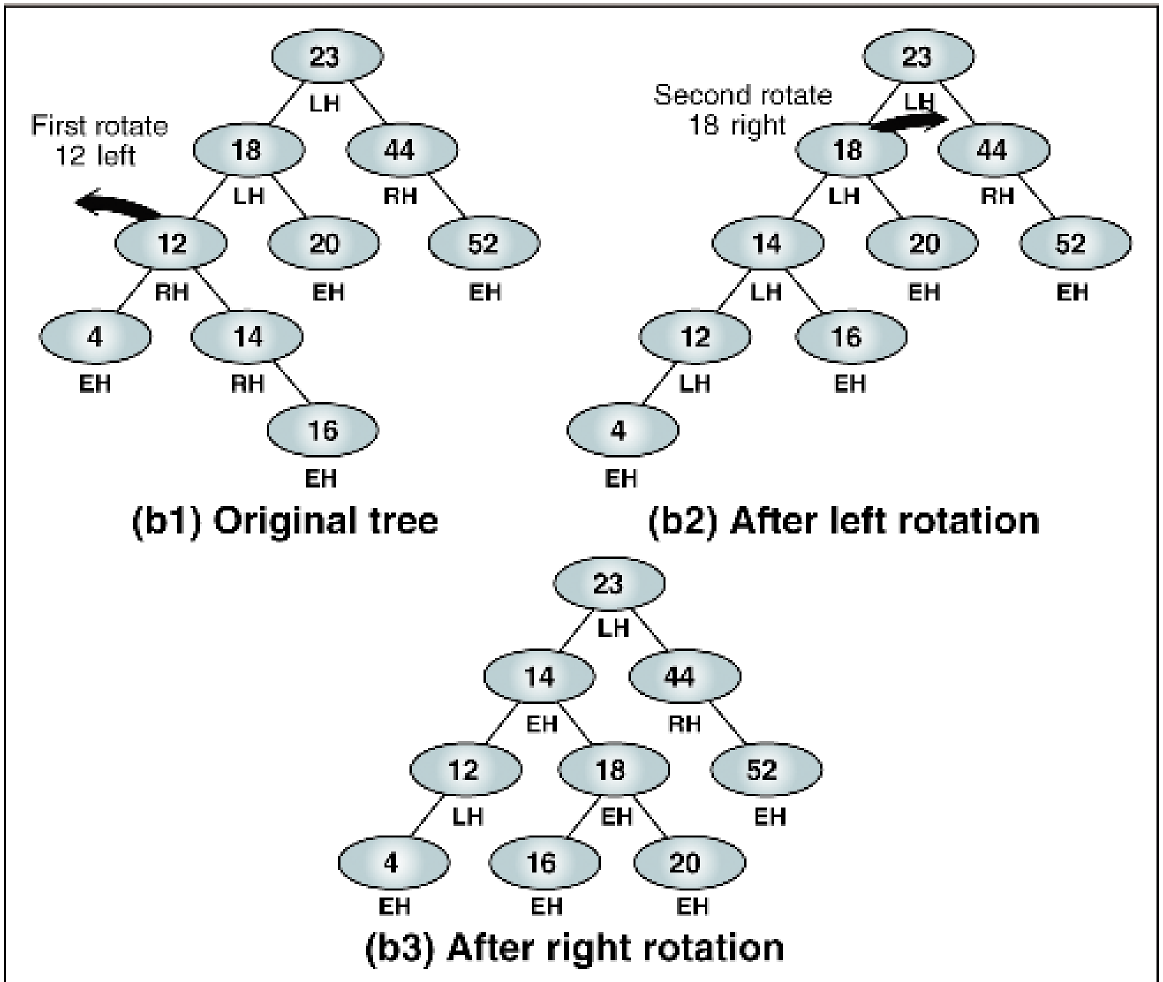


Case 3: Right of Left – Double Rotation Right (Simple)



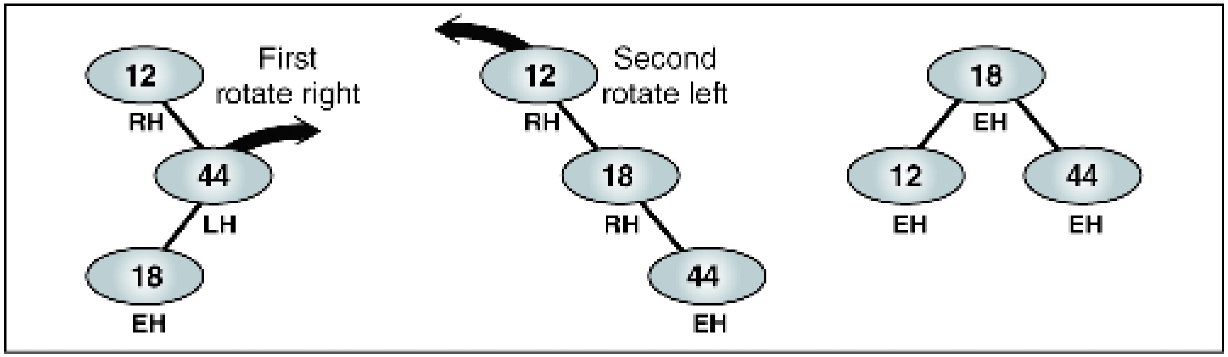
(a) Simple double rotation right

Case 3: Right of Left – Double Rotation Right (Complex)



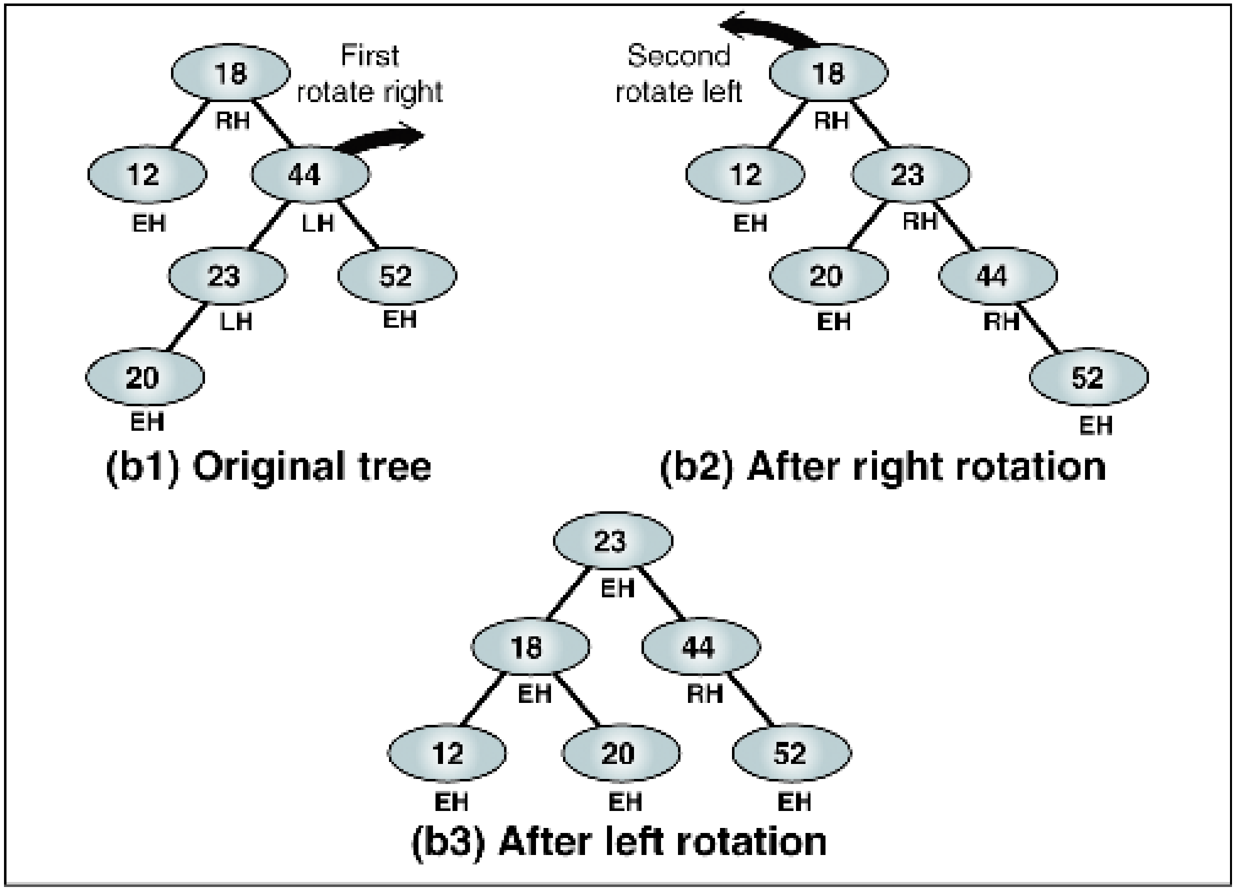
(b) Complex double rotation right

Case 4: Left of Right – Double Rotation Right (Simple)



(a) Simple double rotation right

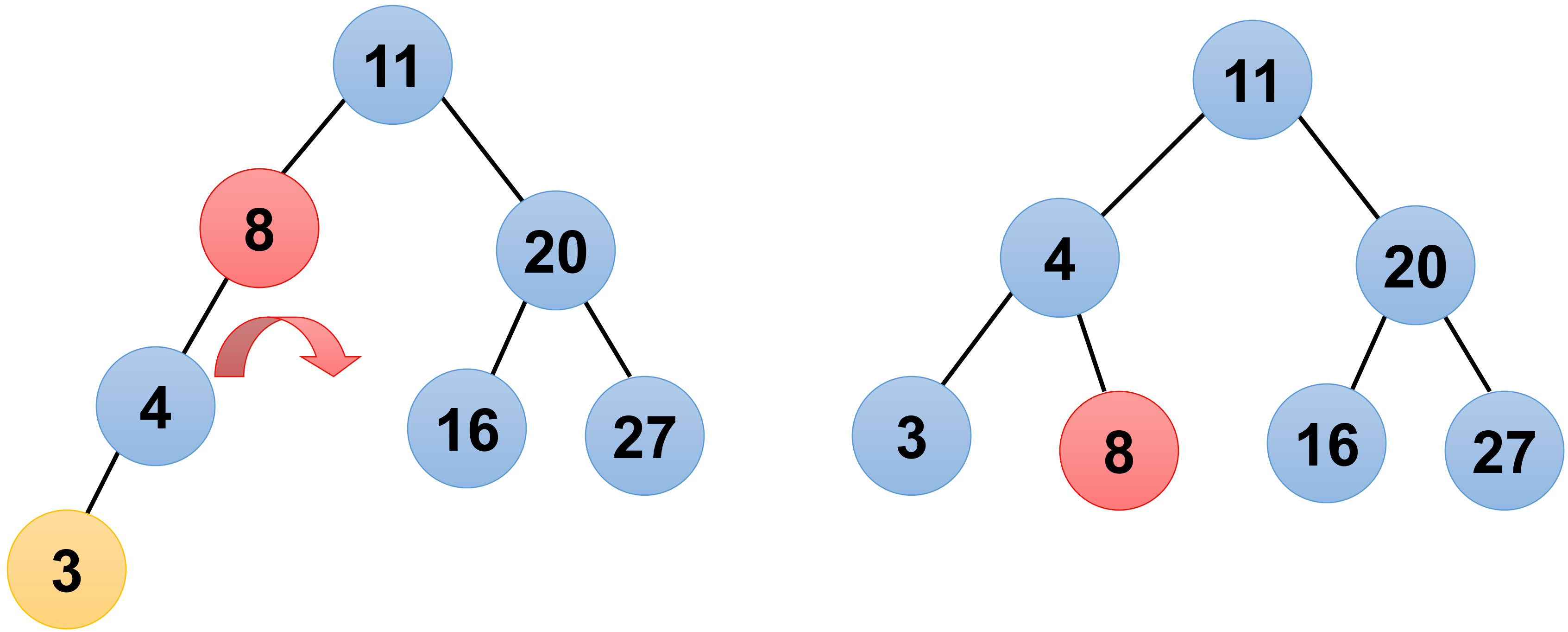
Case 4: Left of Right – Double Rotation Right (Complex)



(b) Complex double rotation right

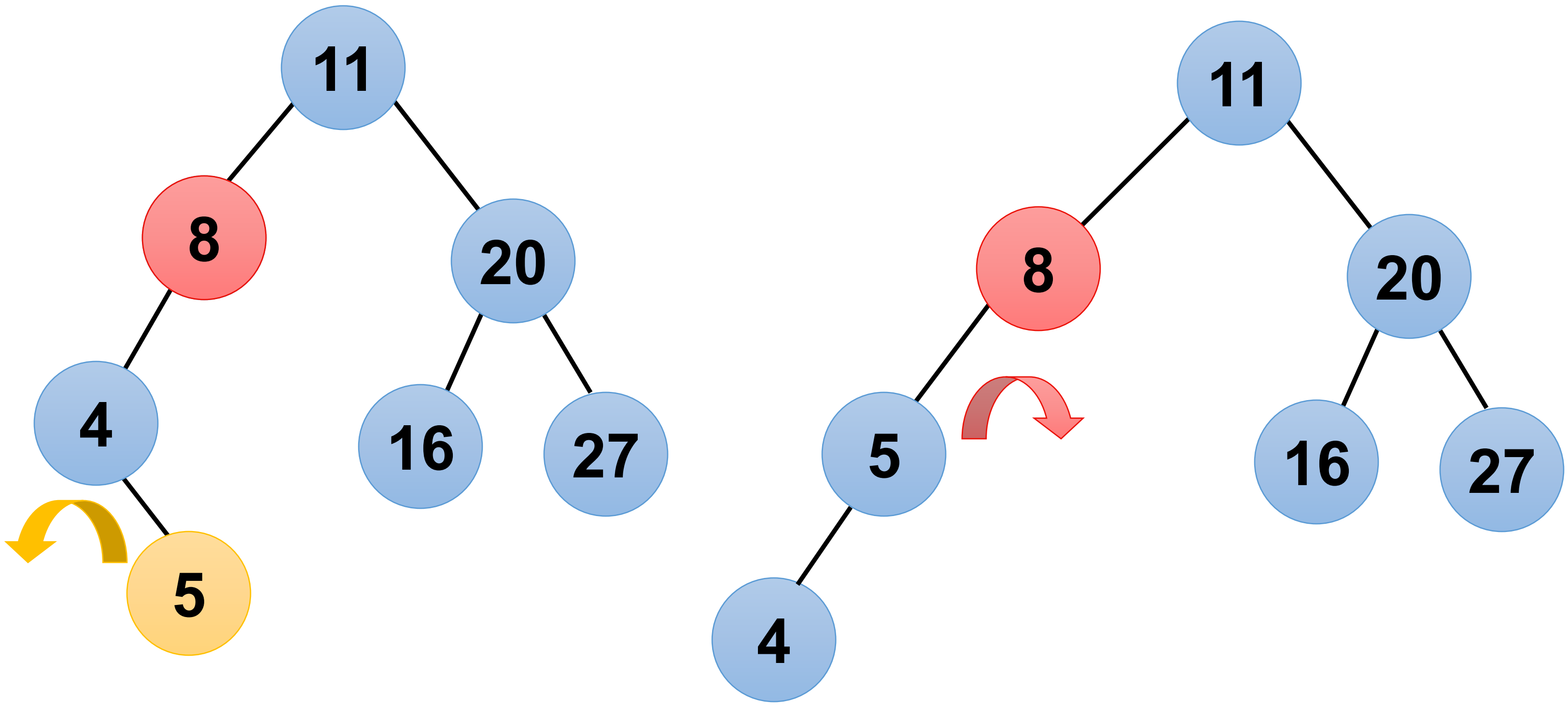
Contoh

- Sisipkan 3 ke AVL tree



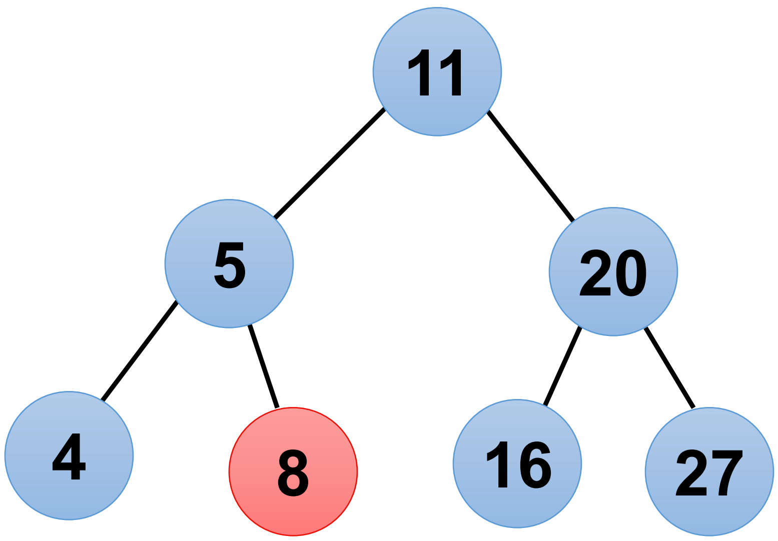
Contoh

- Penyisipan 5 ke AVL tree



Contoh

- Rotasi ke-2



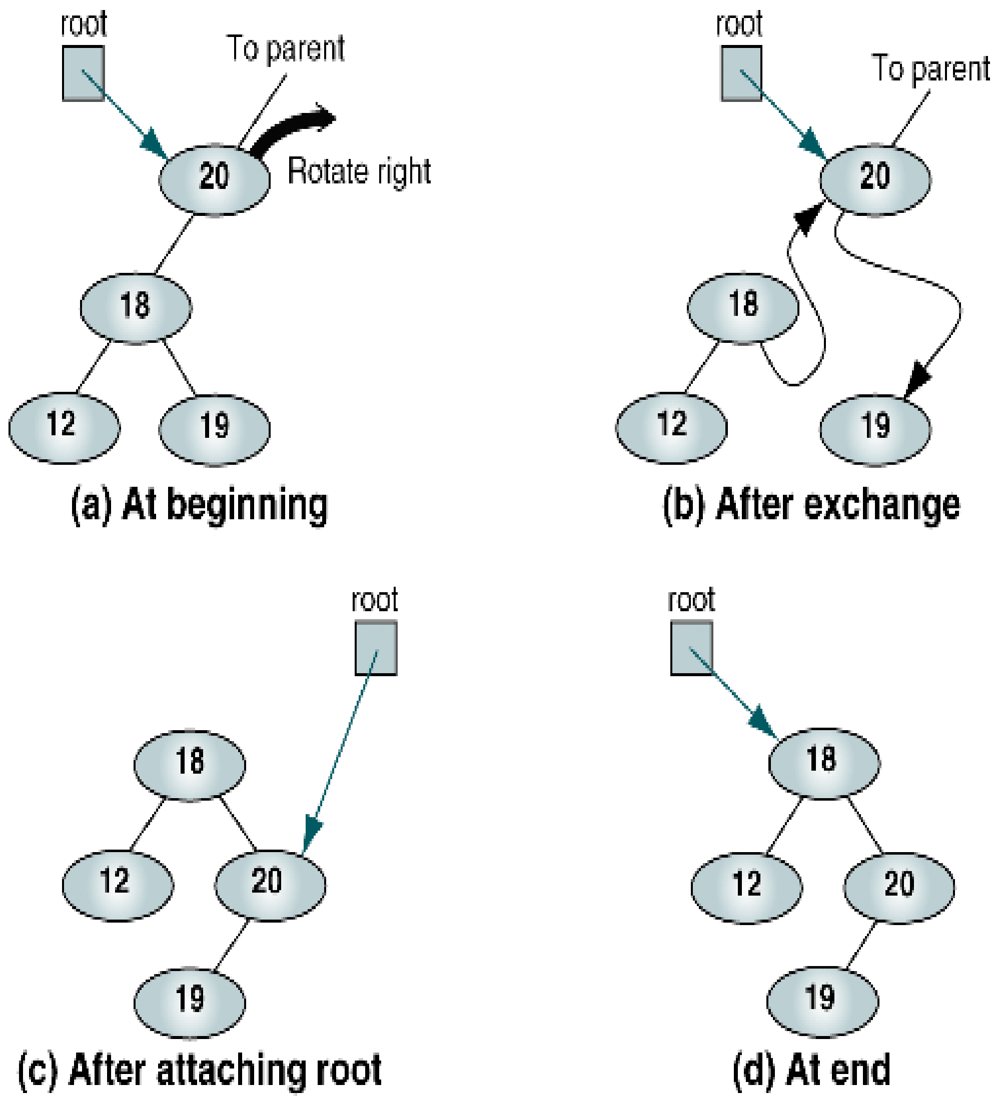
AVL Tree Insert

```
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.  leftBa1ance (root)
     3.  end if
4.  else
     New data >= root data
     1.  AVLInsert ( right subtree, newPtr)
     2.  if (right subtree taller)
          1.  1 rightBalance (root)
     3.  end if
5.  end if
6.  return root
end AVLInsert
```

AVL Tree Insert

```
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
4.  end leftBalance
```

AVL Tree Rotate Right



Single Rotation: Left & Right

```
Algorithm rotateRight (root)
This algorithm exchanges pointers to rot ate 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 rot ate 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
```

Operasi: Remove pada AVL Tree

- Menghapus node pada AVL Tree sama dengan menghapus BST procedure dengan perbedaan pada penanganan kondisi tidak balance.
- 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
- Terapkan single atau double rotation untuk menyeimbangkan tree.
- Bila Tree masih belum balance, ulangi lagi dari langkah 2.

Menghapus node X pada AVL Trees

- Deletion:
  - 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
- Rebalancing
  - Tahap menyeimbangkan node yang balance factornya tidak -1, 0, 1, dilakukan dari node yang dihapus menuju root.

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 I potential I 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)
   2.  if (tree shorter)
      1.  Set root to deleteRightBalance(root)
   3.  endif
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.  endif
```

AVL Tree Delete

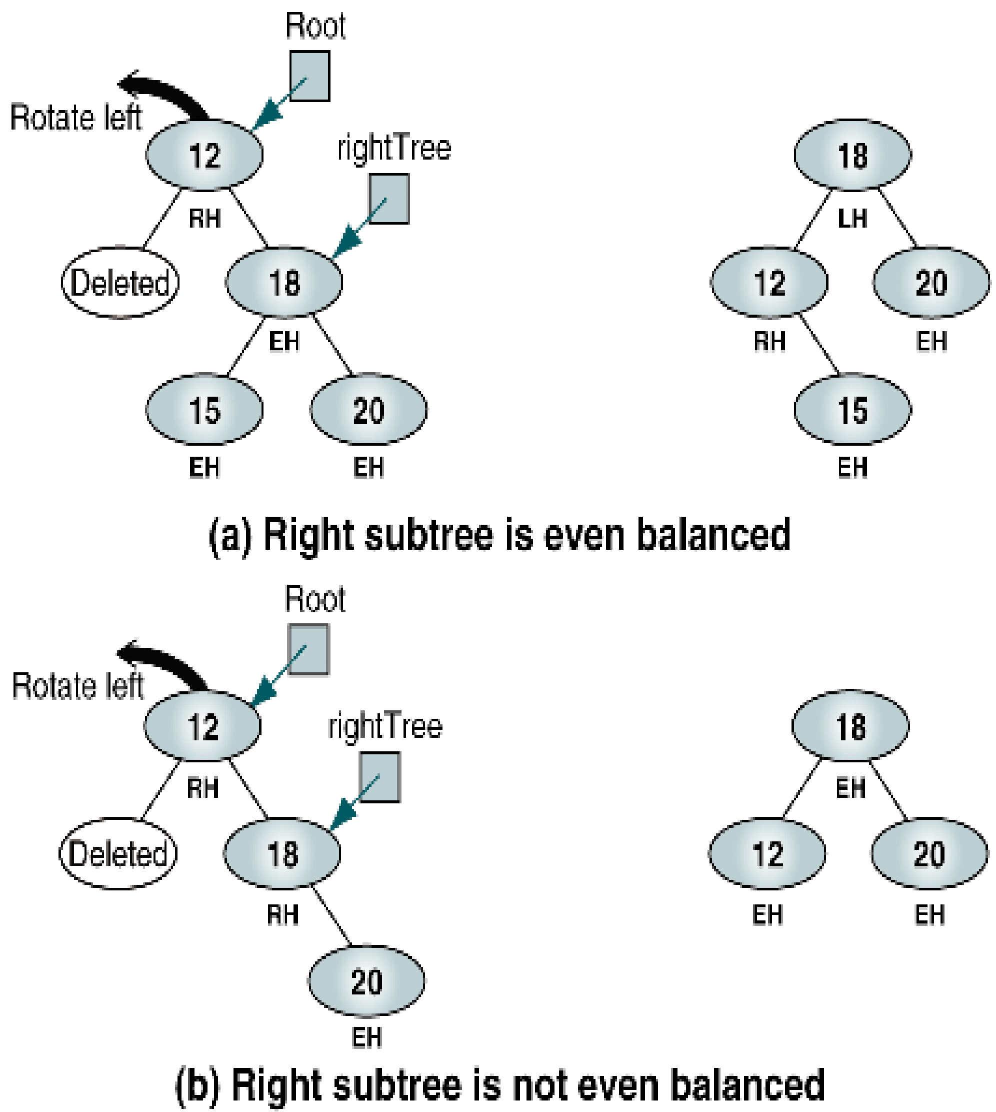
```
5. else
        Delete node found—test for leaf node
   1. save root
   2. if (no right subtree)
      1. set success to true
      2. return left subtree
   3. elseif (no left subtree)
         Have right but no left subtree
      1. set success to true
      2. return right subtree
   4. else
         Deleted node has two subtrees
      1. find substitute—largest node on left subtree
      2. find largest node on left subtree
      3. save largest key
      4. copy data in largest to root
      5. set left subtree to AVLDe1ete(1eft subtree, largest key, success)
      6. if (tree shorter)
          1. set root to dltRightBal (root)
      7. end if
   5. end if
6. end if
7. return root
end AVLDe1ete
```

AVL Tree Delete Right Balance

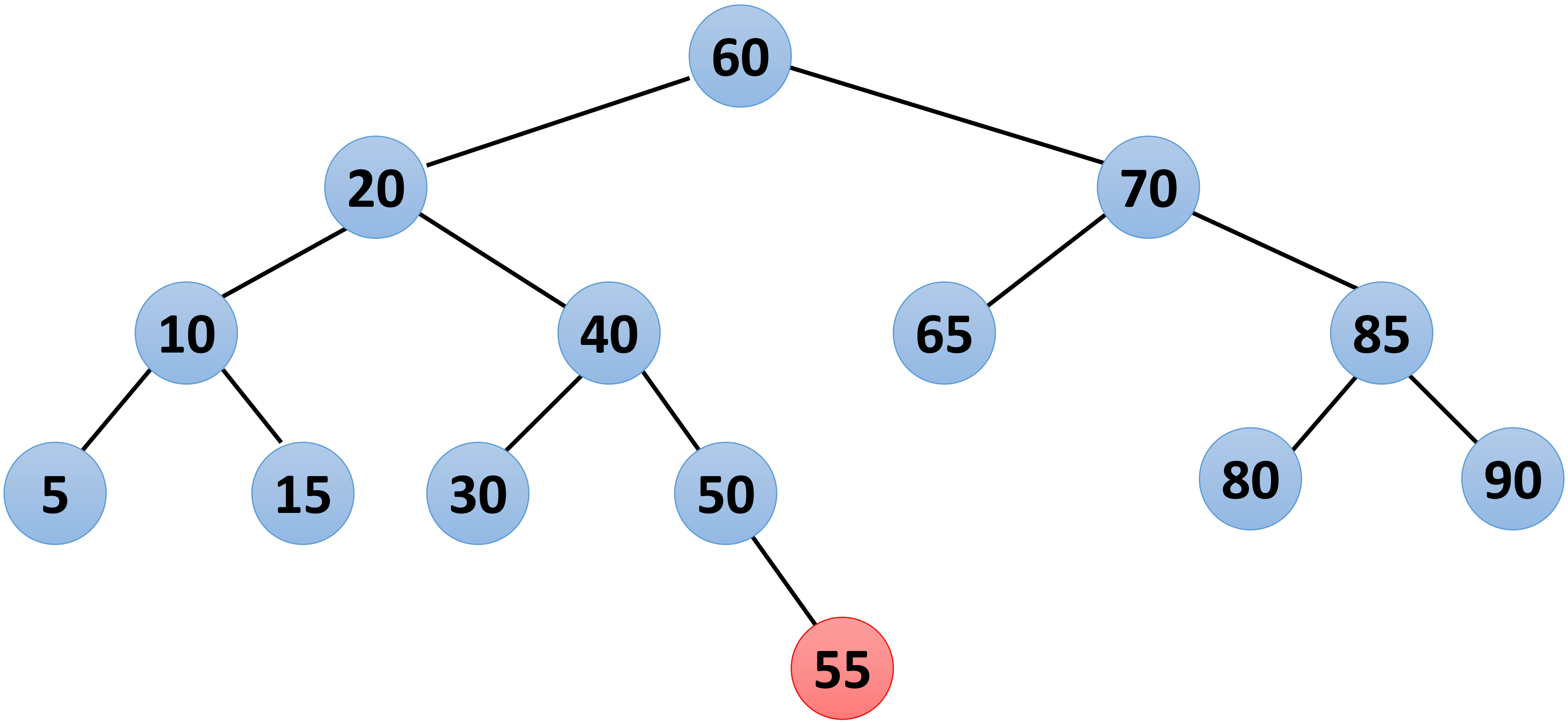
```
Algorithm deleteRightBa1ance (root)
The [sub] tree is shorter after a deletion on the left branch.
If necessary, balance the tree by rotating.
    Pre         tree is shorter
    Post        balance restored
    Return      new root

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
          1.  right subtree  = rotateRight ( rightOfRight)
          2.  root           = rotateLeft (root)
     3.  else
          Single rotation required
          1.  set root to rotateLeft ( root)
     4.  end if
2.  end if
3.  return root
4.  end deleteRightBa1ance
```

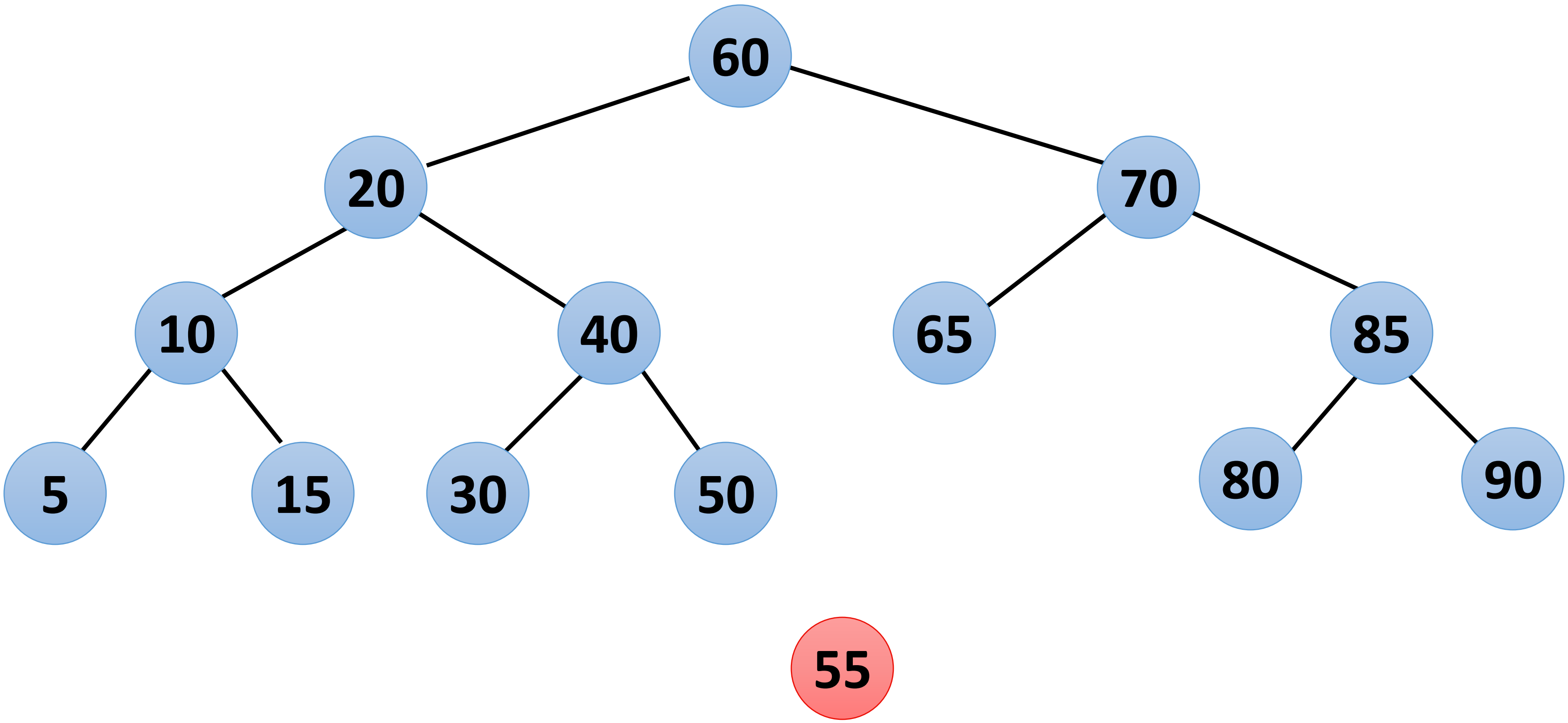
AVL Tree Delete Balancing



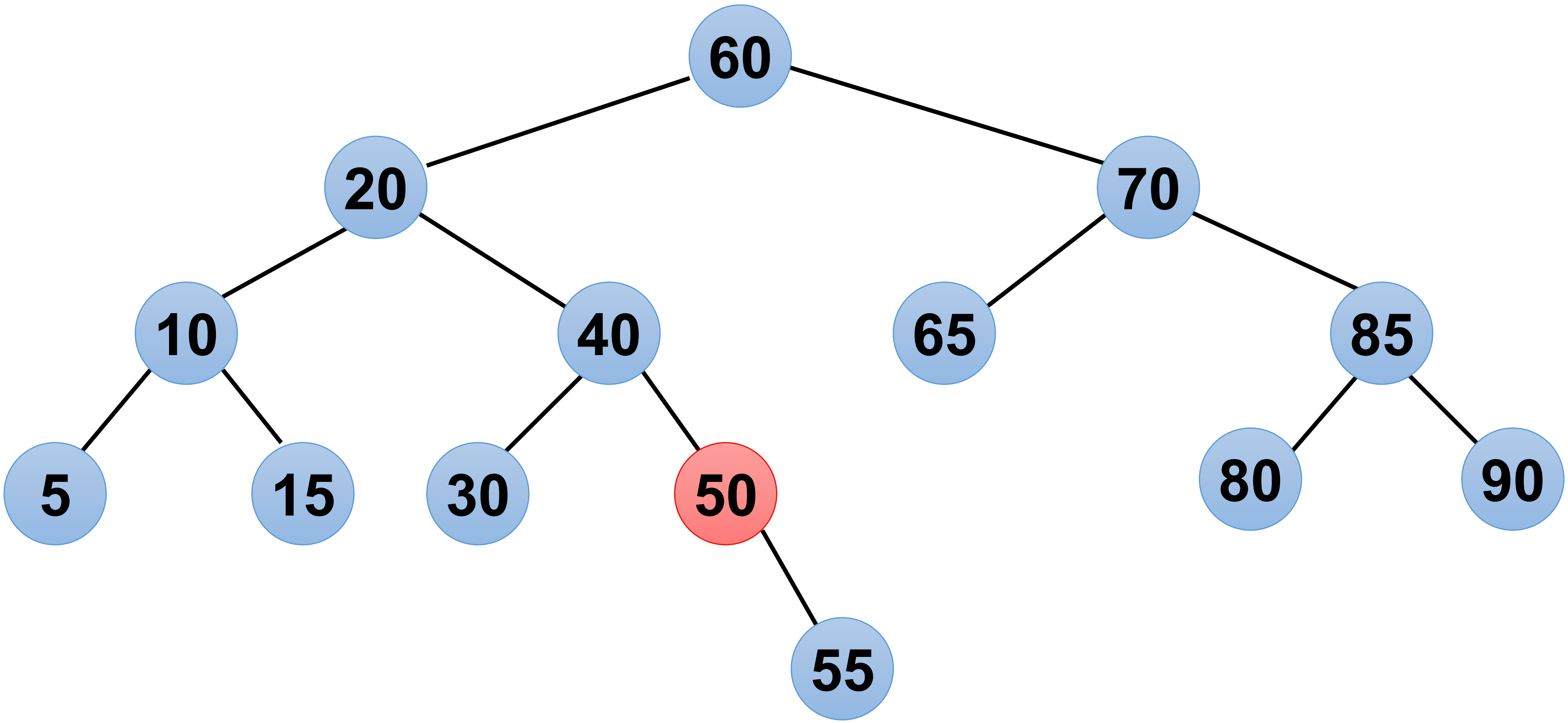
Delete 55 (case 1)



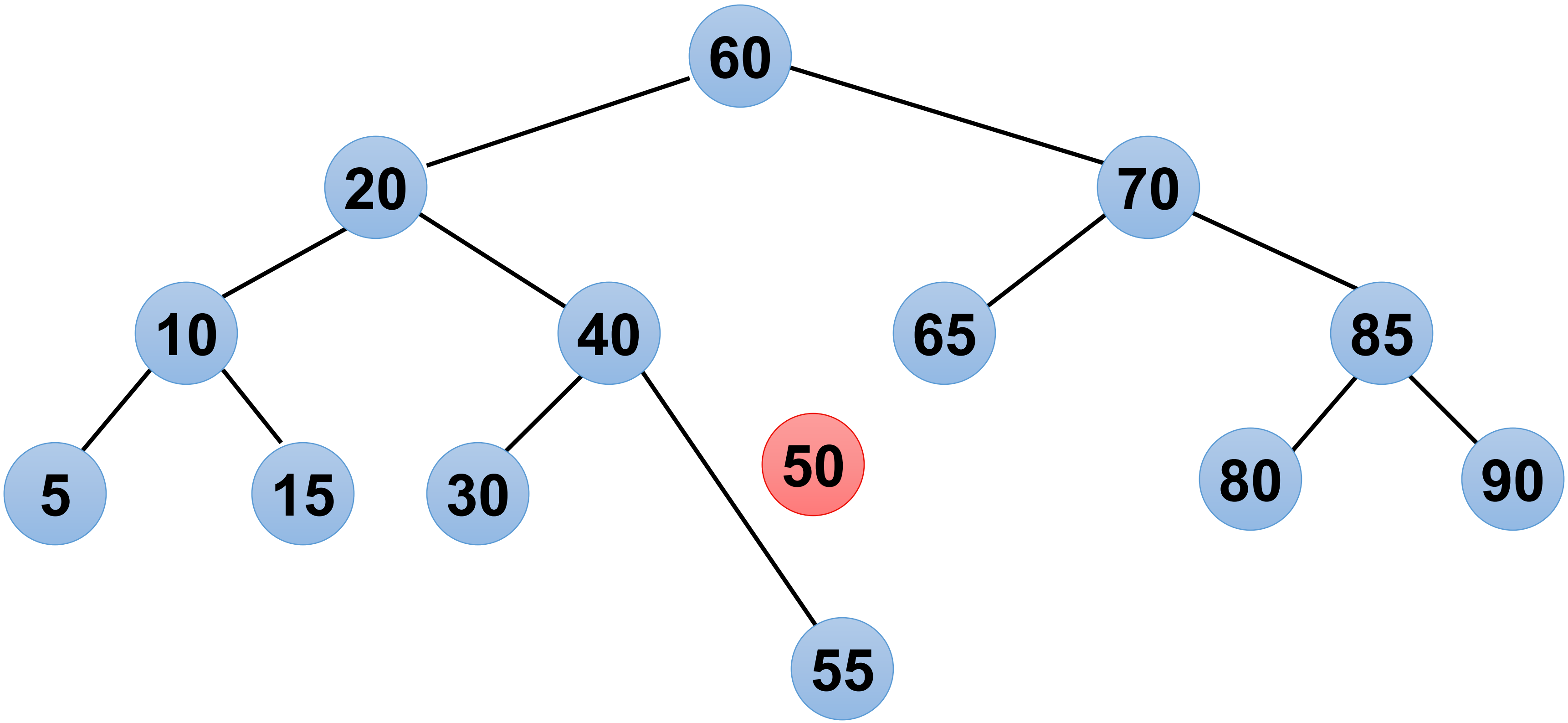
Delete 55 (case 1)



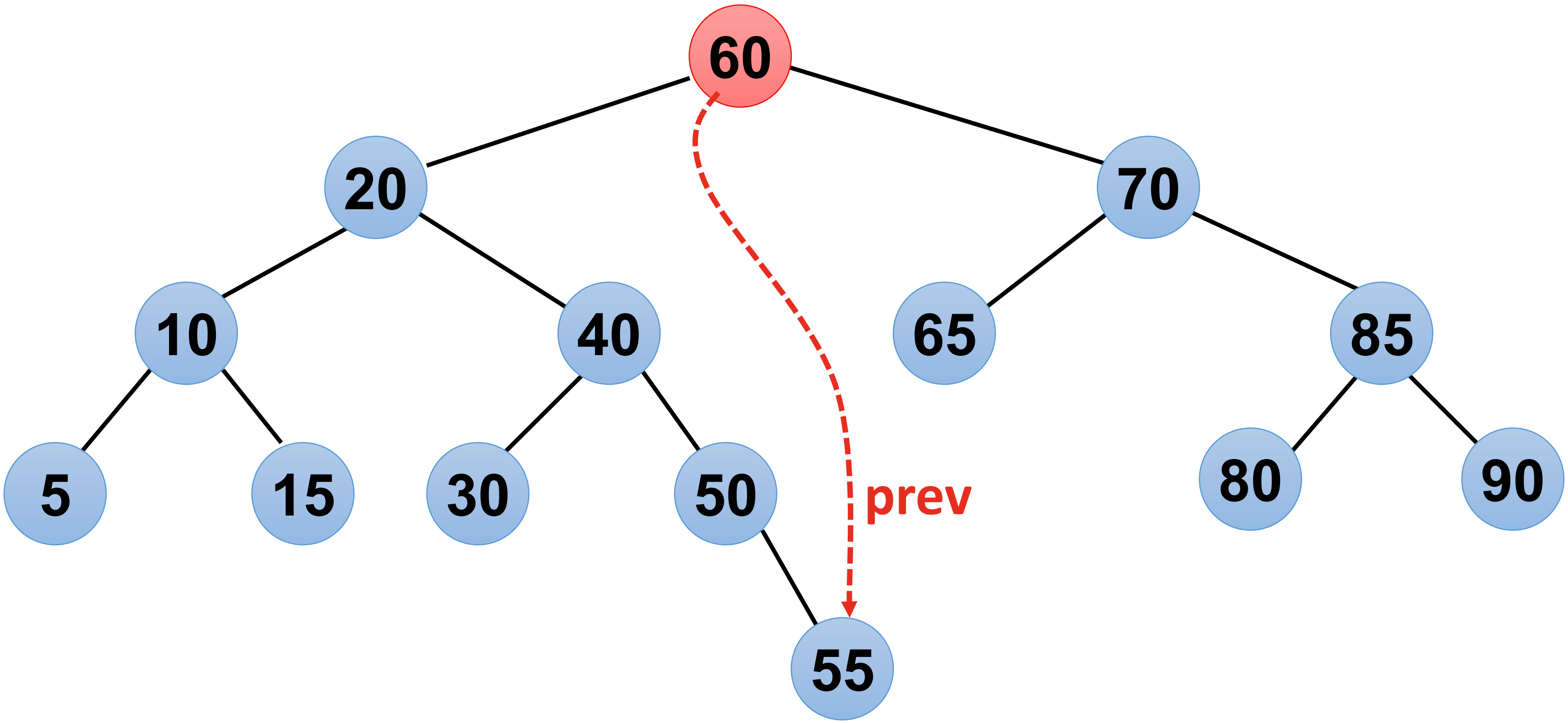
Delete 50 (case 2)



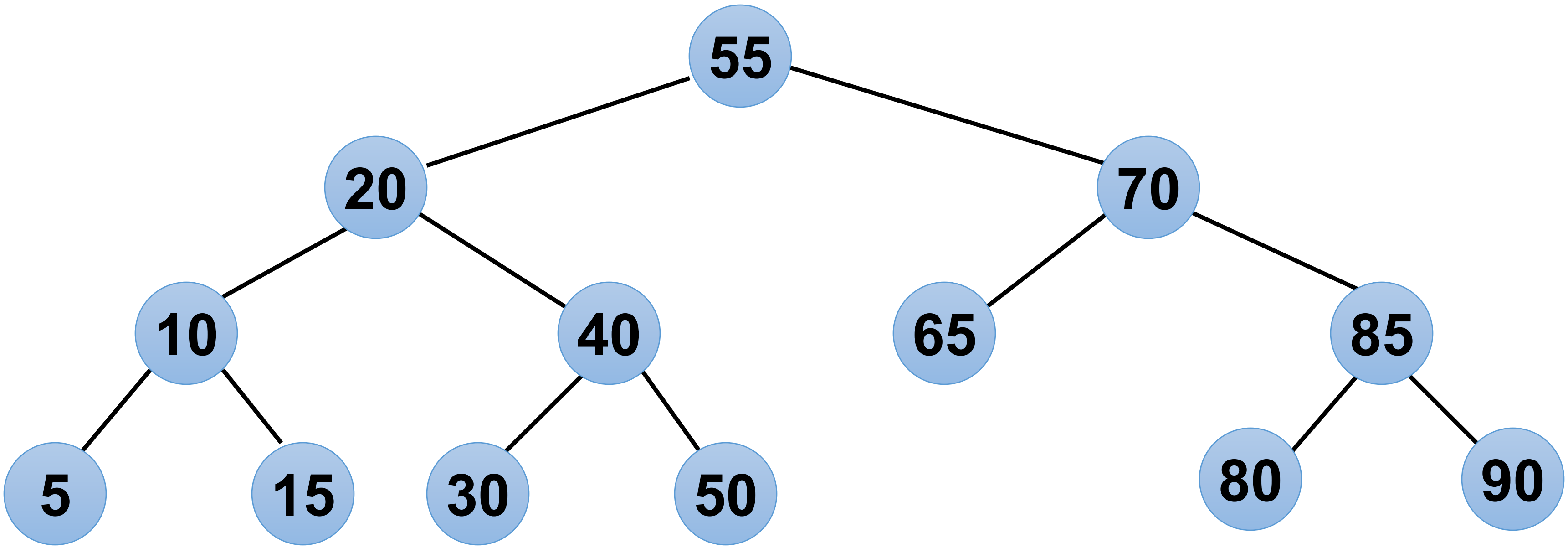
Delete 50 (case 2)



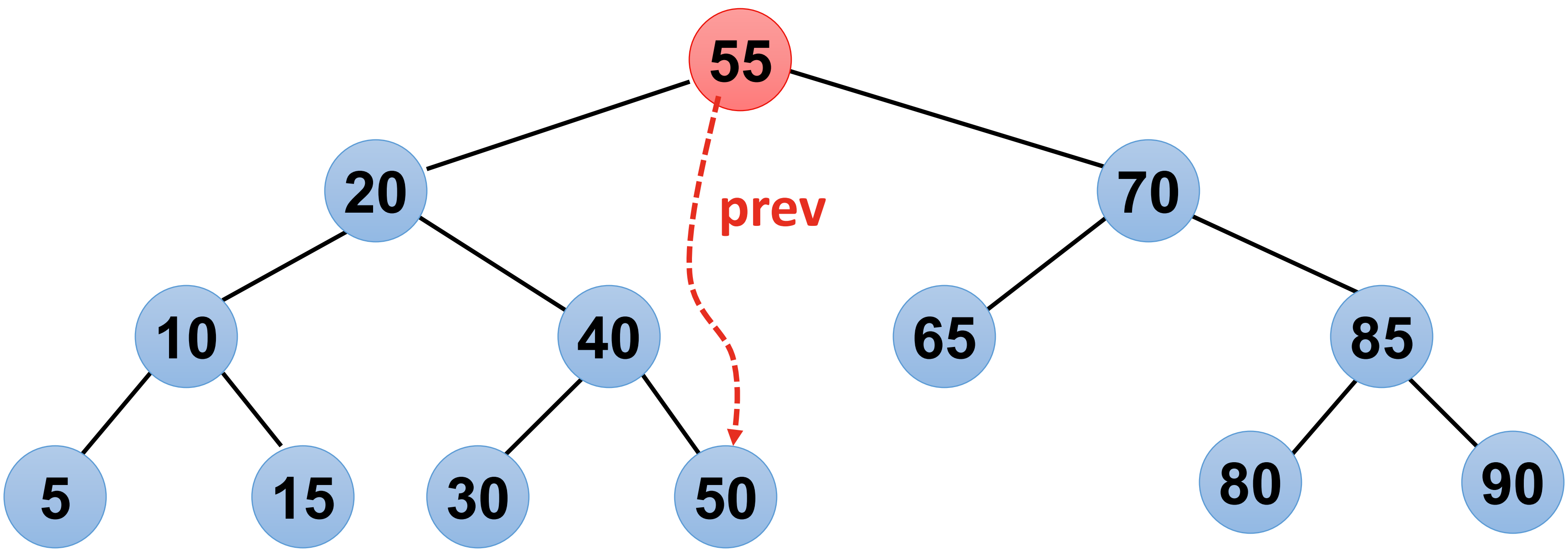
Delete 60 (case 3)



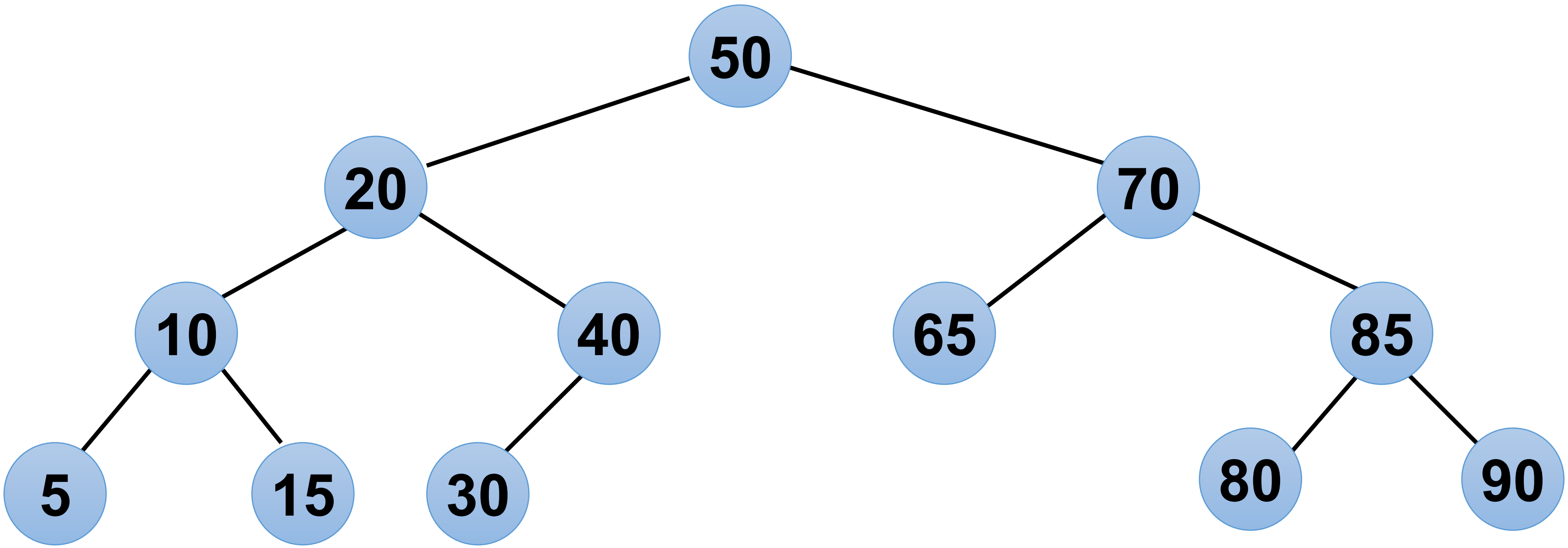
Delete 60 (case 3)



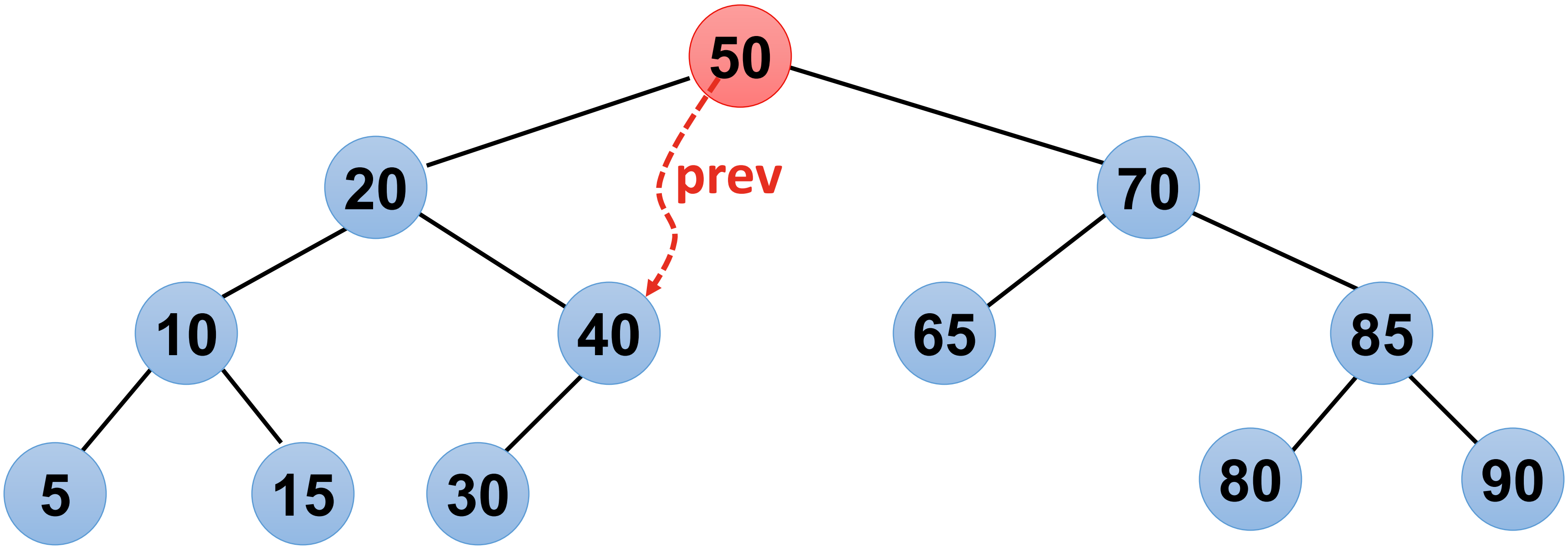
Delete 55 (case 3)



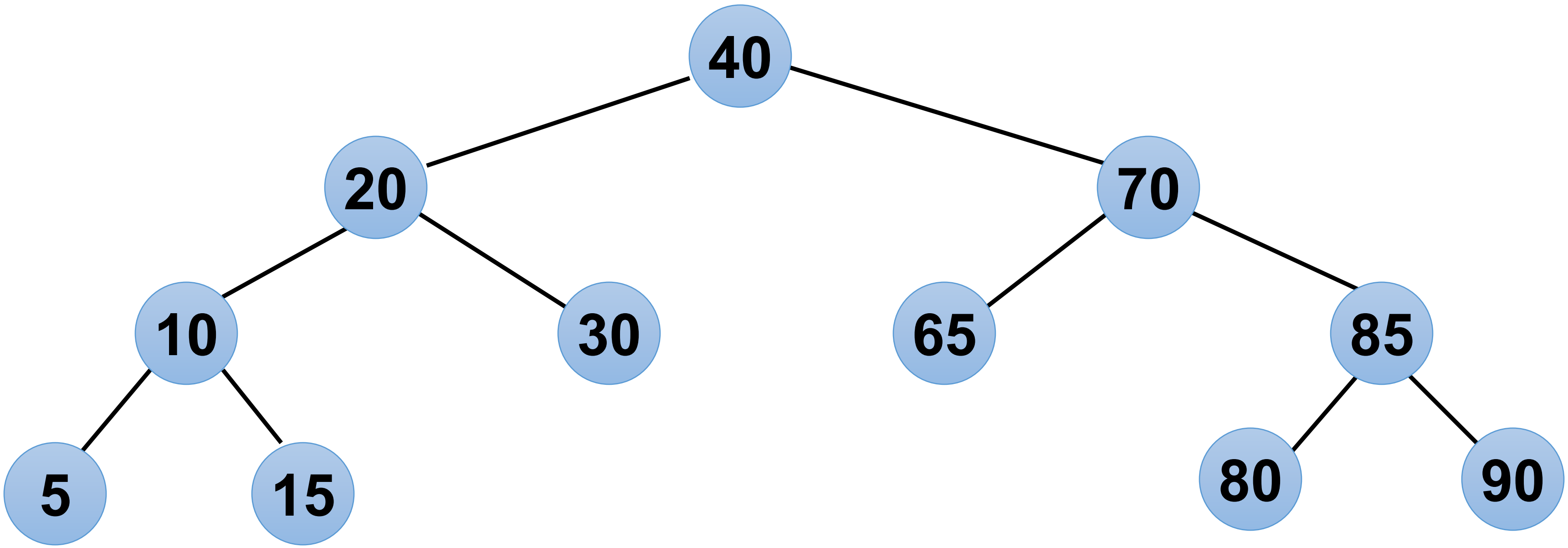
Delete 55 (case 3)



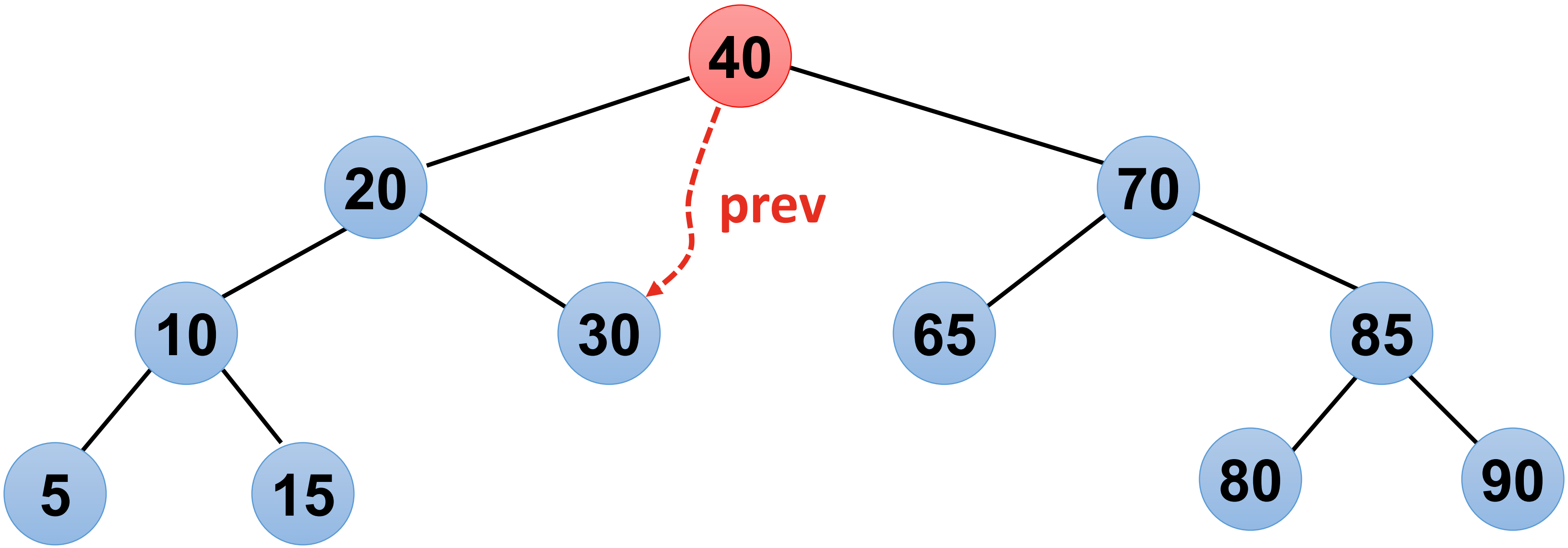
Delete 50 (case 3)



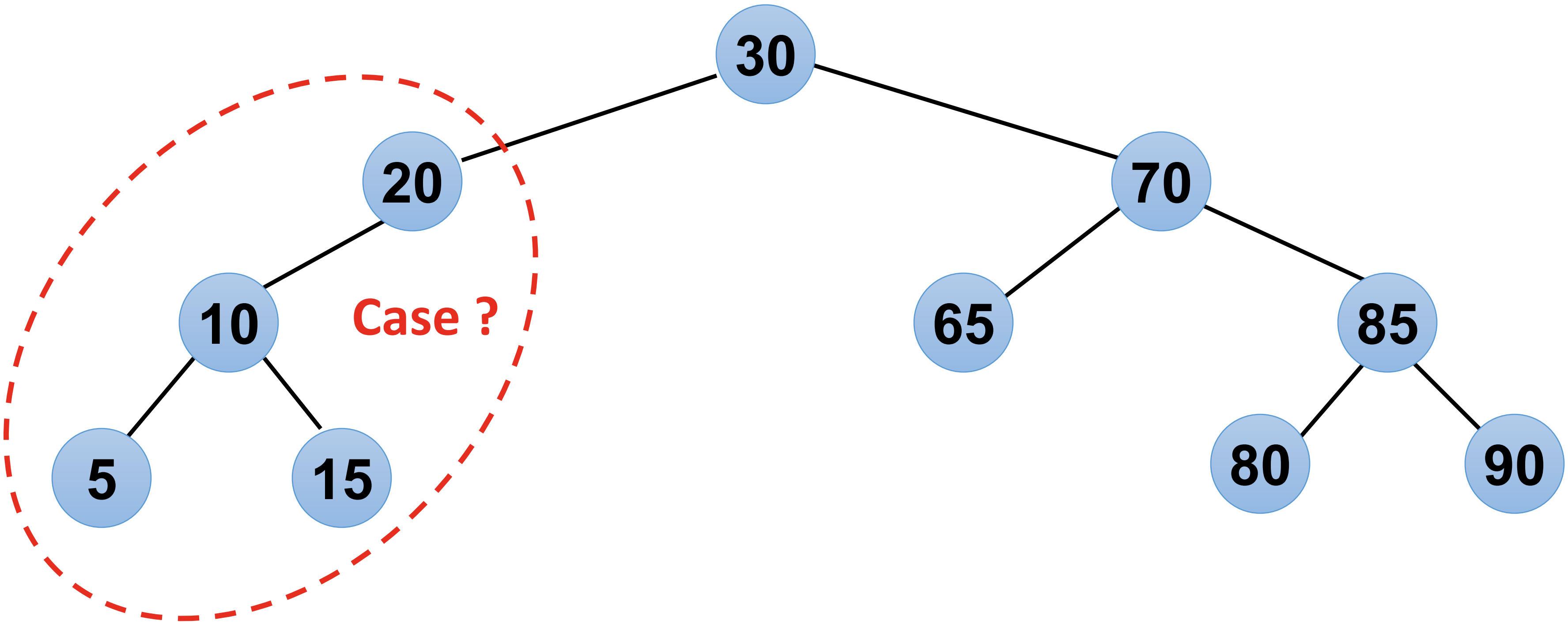
Delete 50 (case 3)



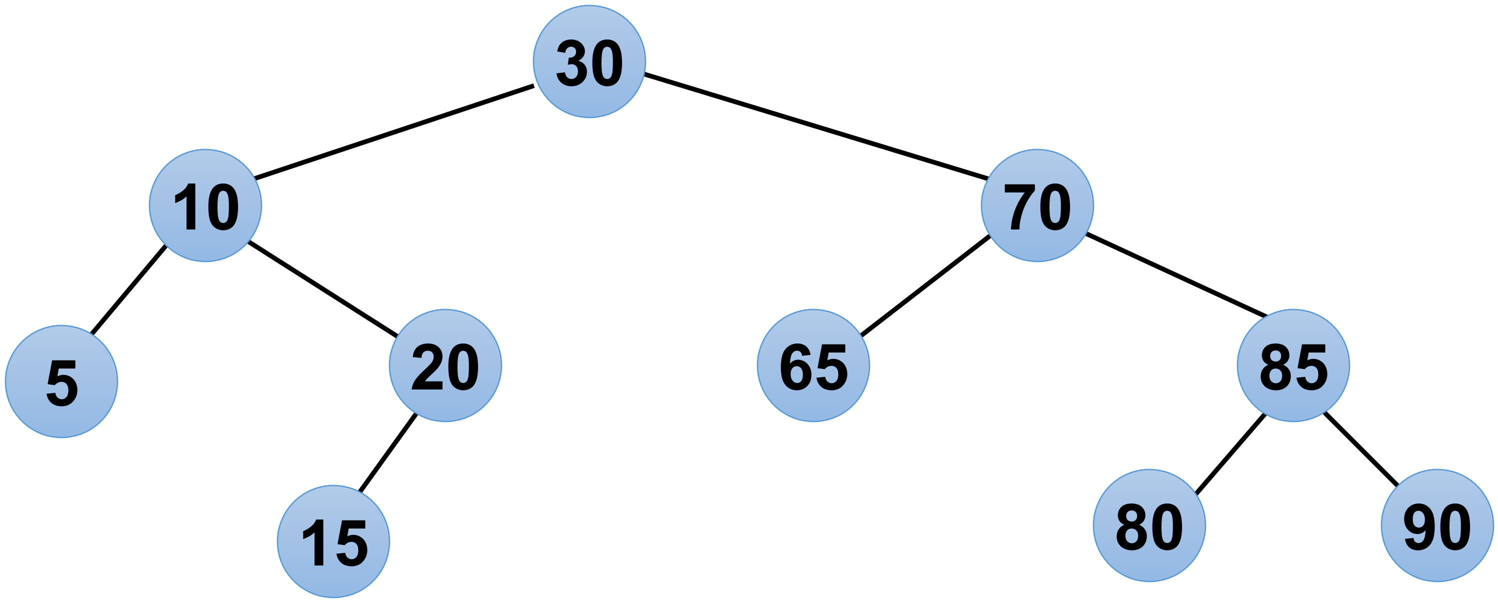
Delete 40 (case 3)



Delete 40: Rebalancing



Delete 40: after rebalancing



Single rotation is preferred!

Latihan

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

- Ingat AVL tree adalah BST, penambahan harus sesuai kaidah pada BST

Implementasi AVL Tree

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

Ide

- Setiap kali melakukan insert, perlu mengecek 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*.

Algoritme 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.

Algoritme 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” ?

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** !

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;
}
```

Code

```
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 = 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

Code: Single Rotation

```
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;
}
```

Code: Double Rotation

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