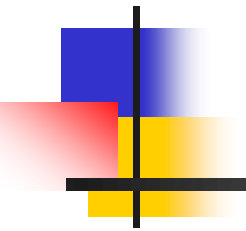


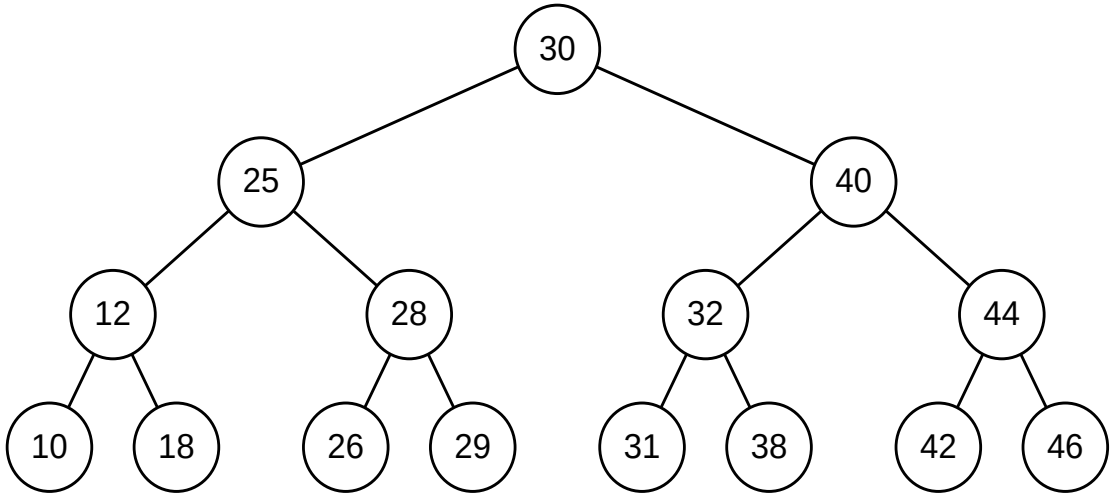
Chapter 06

Self-balancing binary search tree
AVL Trees

Binary Search Trees

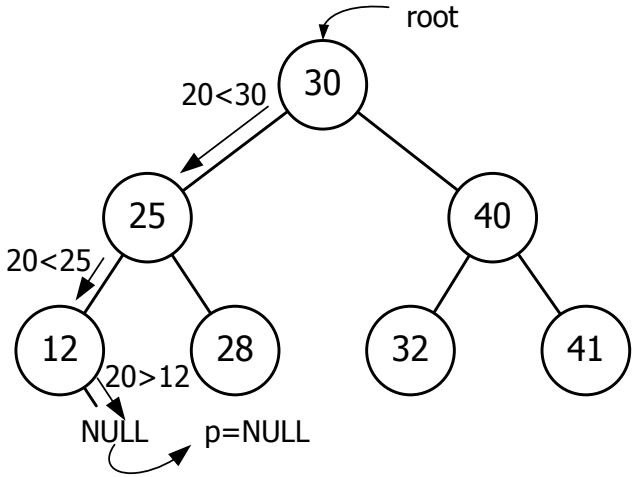
- **Binary search trees**
  - ค่าของโหนดลูกทุกๆโหนด ที่อยู่ทางซ้าย จะต้องมีค่าน้อยกว่าค่าของโหนดลูกทุกๆโหนดที่อยู่ทางขวา
  - โดยปกติค่าคีย์ที่ใช้สร้างโหนดจะต้องไม่ซ้ำกัน (แต่ถ้ายอมให้ซ้ำได้ จะให้ตัวที่ซ้ำไปอยู่ทางขวา และจะต้องเสียเวลาในการตรวจสอบเพิ่ม)

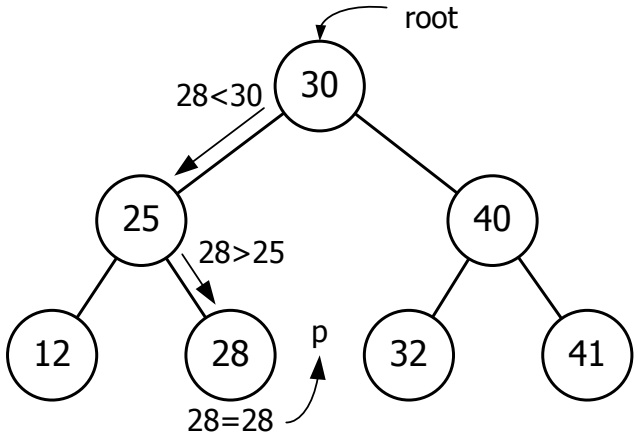
value in **left child < node < right child**



Binary Search in Tree

Example **Find the value 20 & 28 from the following tree structure**





Entry to node 30 **compare with key=20** **20 < 30 Entry to the left child(25)** **20 < 25 Entry to the left child(12)** **20 > 12 right Entry to right child** **right child =NULL** **return (Not Found)**	**Entry to node 30** **compare with key=28** **28 < 30 Entry to left child(25)** **28 > 25 Entry to right child(28)** **28 =28** **return (Found)**

Programming in Java

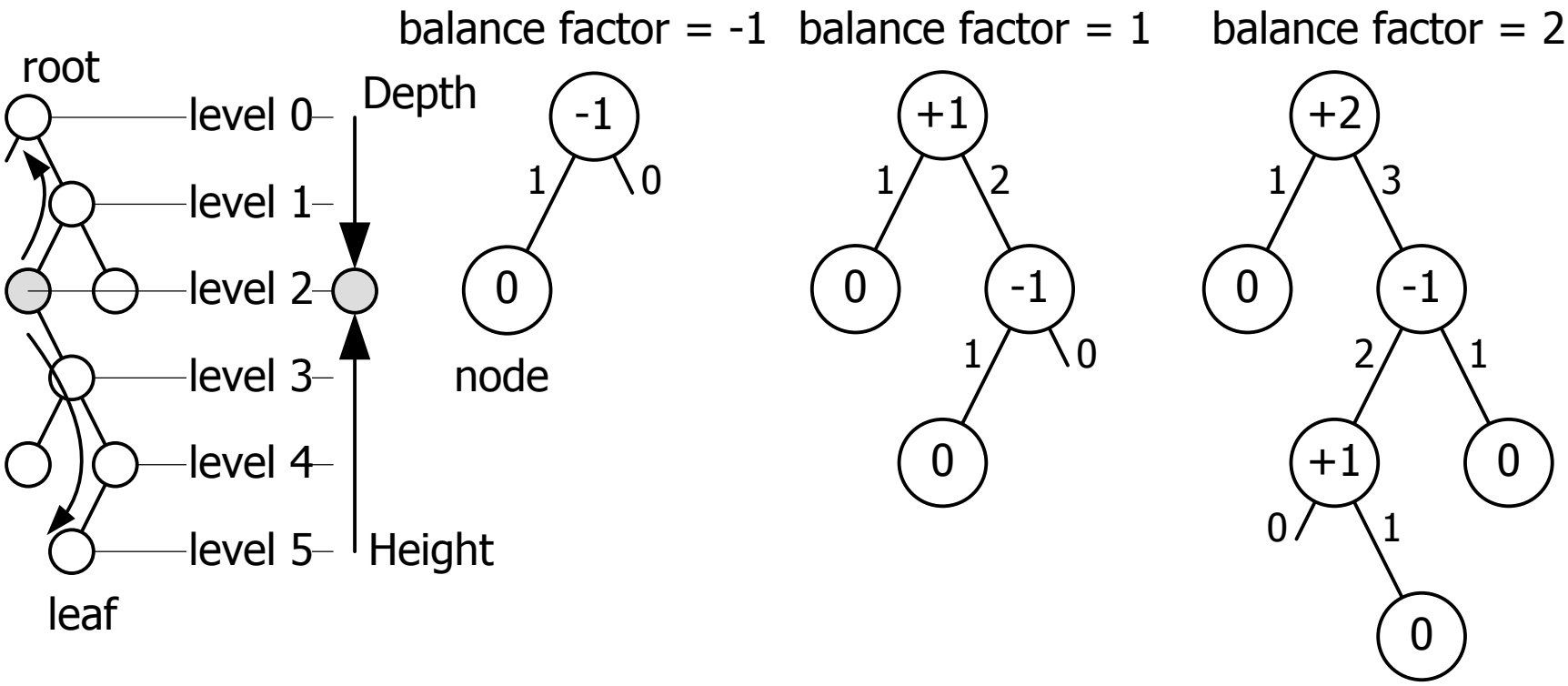
```
Node  binary_search(Node  root,  long key)
{Node  current = root;
   while (current != null)
   {  if (current.info == key)
         return current ;
      else  if (current.info > key)
               current = current.left ;
            else
               current = current.right;
    }
     return  null ;
}
```

Balanced Trees

- **Balanced Trees : A tree which the height of left and right subtrees of every node differ by at most one (Absolute).**
- **Balance Factor: The differences between the heights of the left and right subtree.**

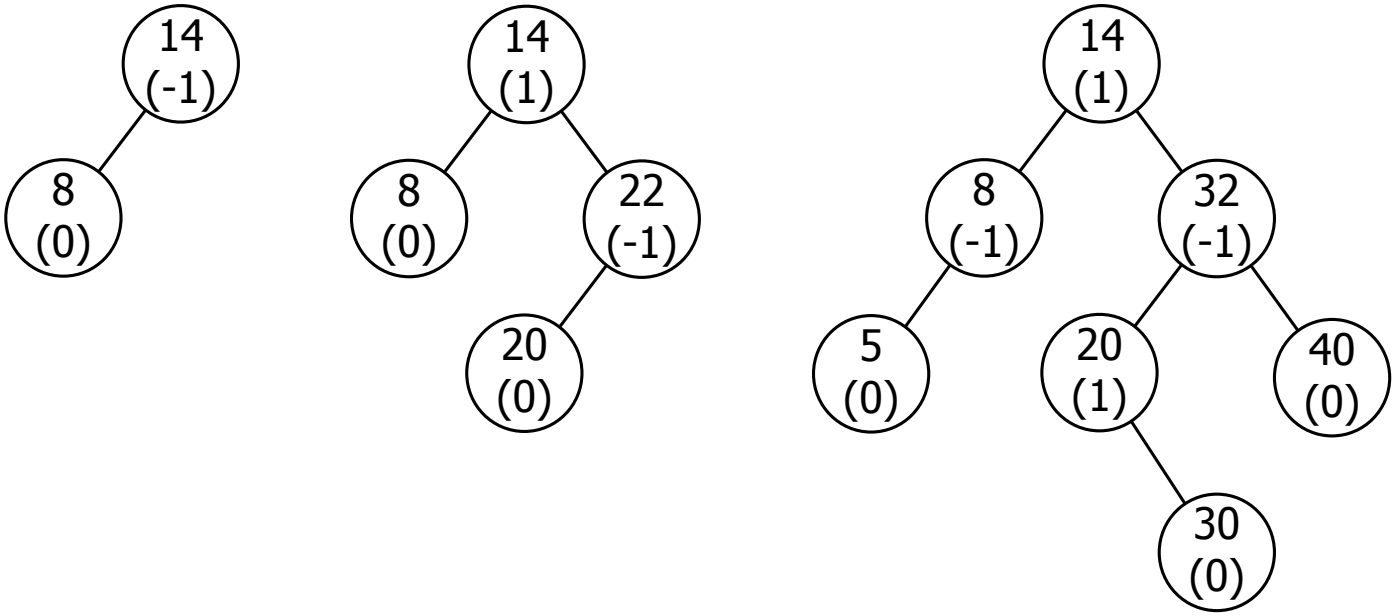
Balance Factor = height(right subtree) - height(left subtree)

Height of subtree = 1+max(height of leftsub, height of rightsub)



AVL Tree

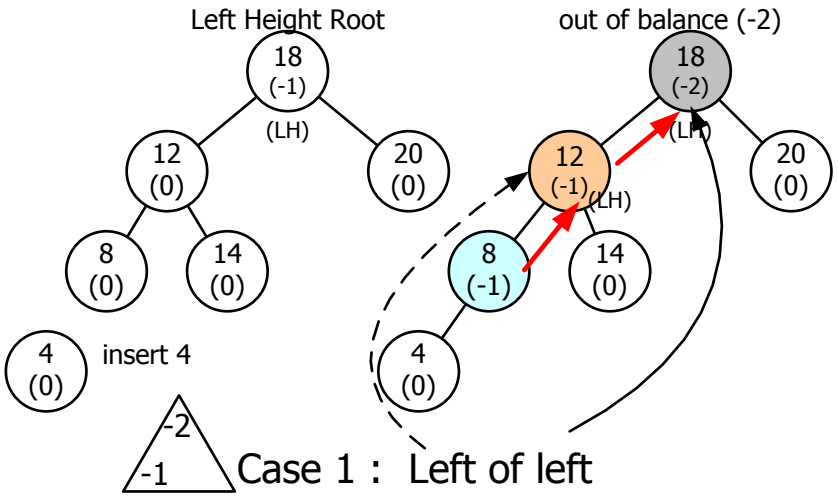
- *AVL Tree (Adel'son-Vel'skii and Landis)*
  - ต้นไม้ไบนารี ที่ปรับสมดุลโดยกำหนดความสูงของ sub-tree ทั้งสองข้าง ของทุกๆโหนด ให้มีความแตกต่างกันได้ไม่เกิน 1 |HR - HL| <= 1
  - วัดด้วยการใช้ balance factor (ความสูงทางขวา-ความสูงทางซ้าย)
  - AVL is balance binary search tree

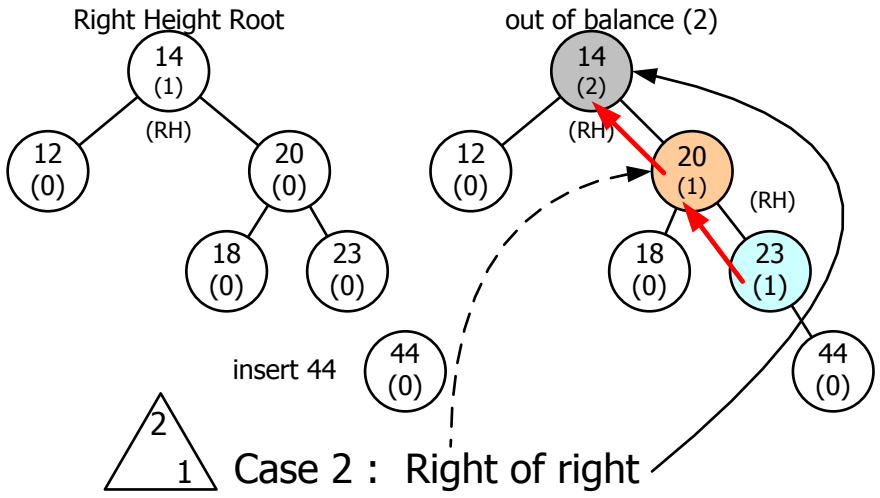


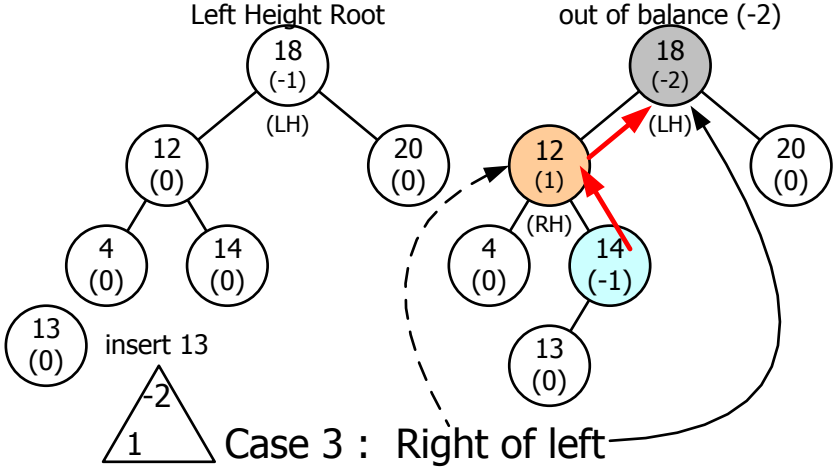
Class Node & Binary Tree

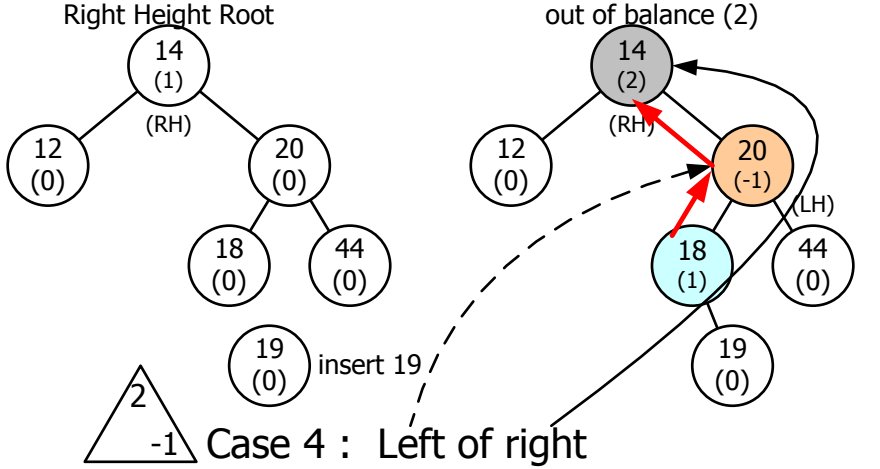
```
class  Node {  long info ;
               int height , balance ;     // if have height or balance ;
               Node  left, right;
      int nodeHeight(Node n) {
        if (n==null) return -1;
        else return 1+ Math.max(nodeHeight(n.left), nodeHeight(n.right)) ;
        }
      int balanceFactor(Node n) {
          return nodeHeight(n.right) - nodeHeight(n.left)  ;
        }
      void reCalculate(Node n) { // postorder traversing
      int  leftHeight=-1, rightHeight=-1;
           if (n.left != null) {reCalculate(n.left); leftHeight = n.height ;}
           if (n.right != null) {reCalculate(n.right); rightHeight = n.height;}
           height = 1+Math.max(leftHeight,rightHeight);
           balance = rightHeight - leftHeight;

        }
}
```

Out of Balance AVL trees









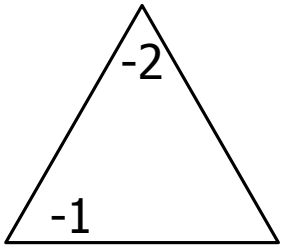
Rotation in Binary Trees

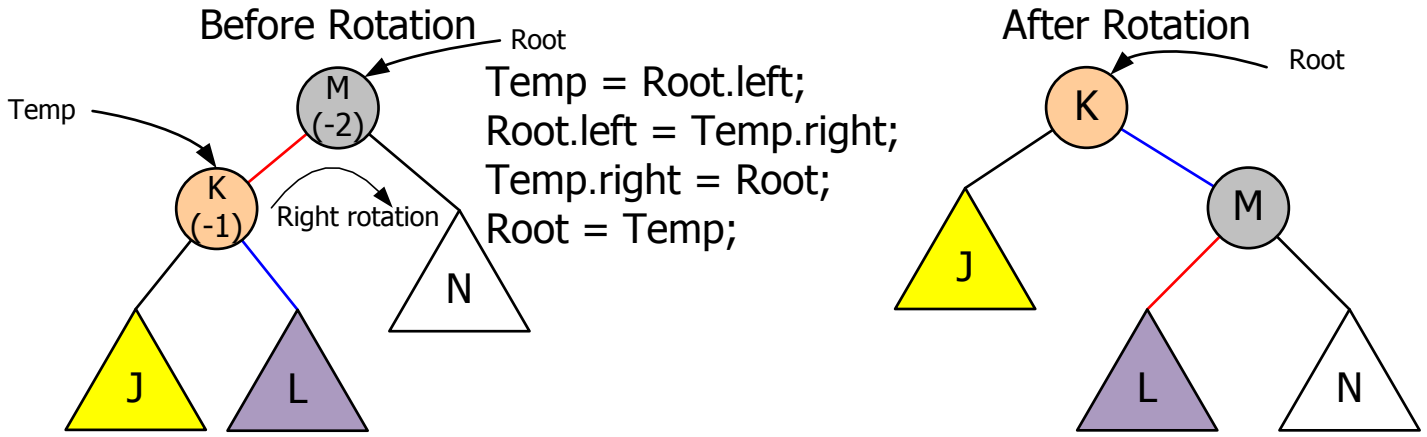
- *Rotation in Binary Trees*
  - Single Rotation
    - Right Rotation (for Left of left Case 1)
    - Left Rotation (for Right of right Case 2)
  - Double Rotation
    - Left to Right Rotation (for Right of left Case 3)
    - Right to Left Rotation (for Left of right Case 4)

Right Rotation

- ***Right Rotation for left of left***

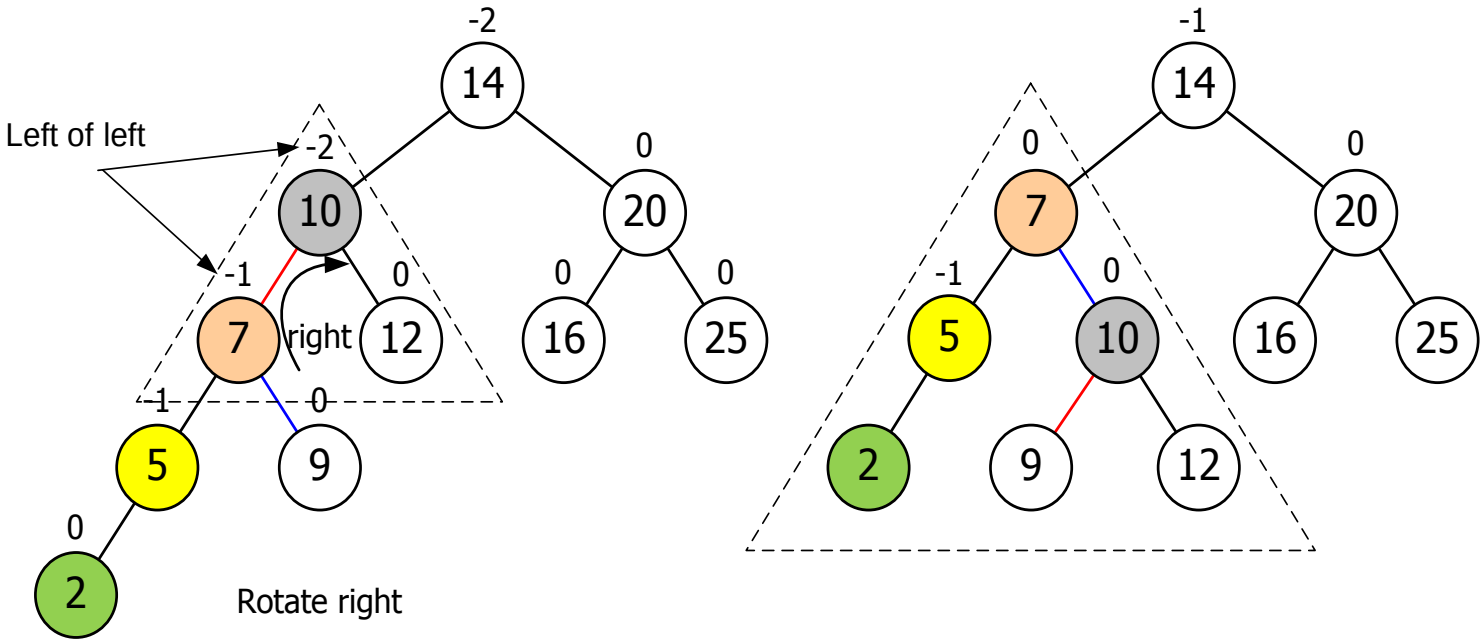
```
Node  rotateRight (Node  root)
{Node temp ;
     temp  = root.left;
     root.left = temp.right;
     temp.right = root;
    root = temp;
     return(root);
}
```





Adjust AVL Tree(Rotate right)

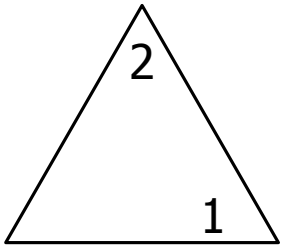
- **14, 20, 10, 7, 25, 12, 16 , 5, 9, 2**
  - **เมื่อเพิ่มโหนด 2 จะทาให้โหนด 10 ไม่สมดุลทางด้ายซ้าย (Balance factor = -2)**
  - **ทางซ้ายของโหนด 10 มีค่าบาลานซ์เป็น -1**
  - **ต้องแก้ด้วยการหมุนโหนดไปทางขวา (Balance factor เป็น -2 กับ -1)**
  - **หมุนโหนดทางซ้ายของโหนด 10(โหนด 7) ขึ้นมาทางขวา แล้วย้ายโหนด 10 ลง**

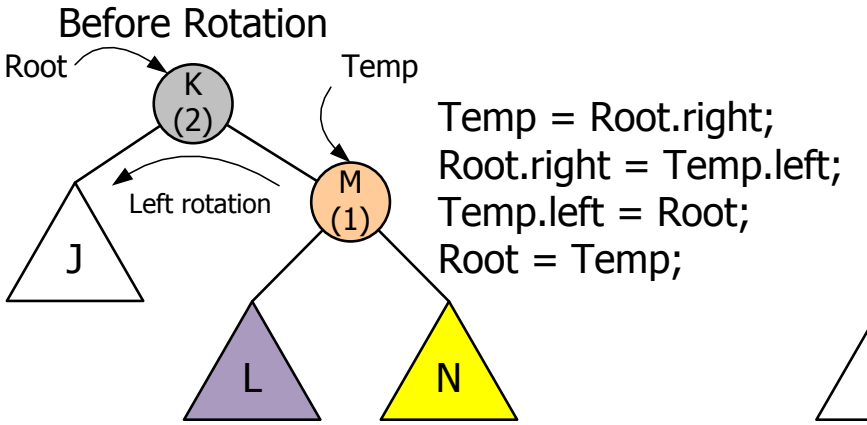


Left Rotation

Left Rotation for right of right

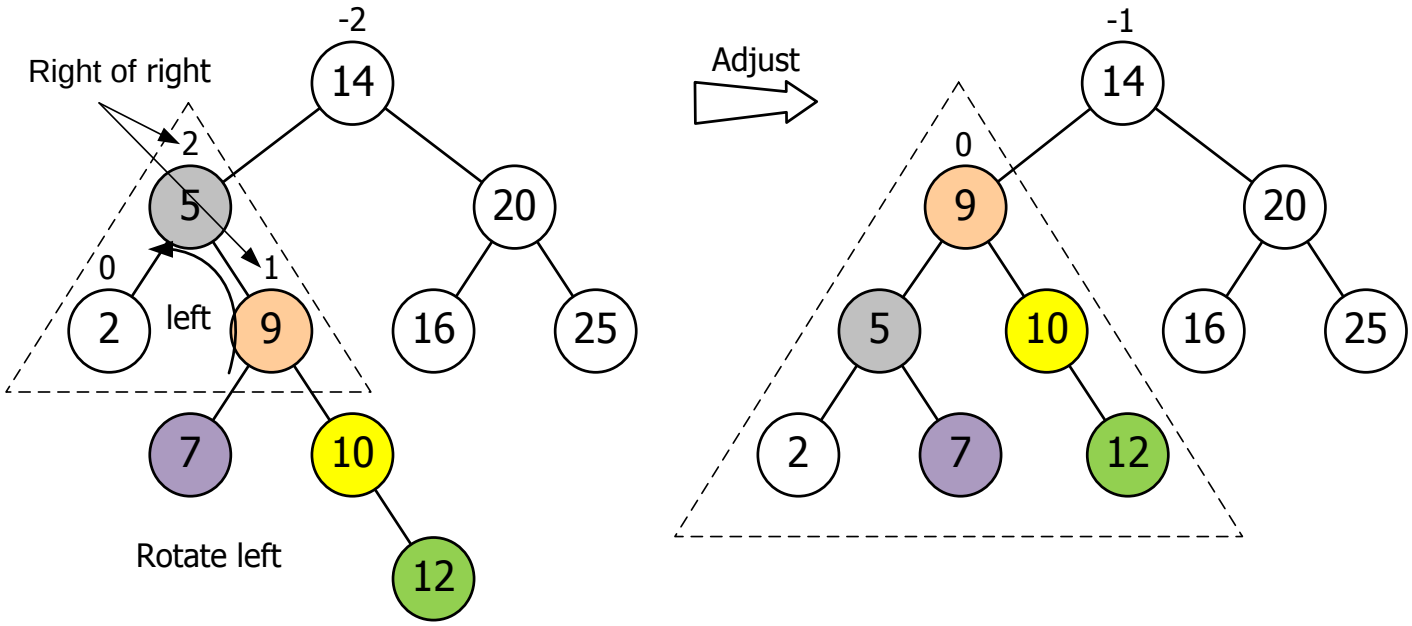
```
Node rotateLeft (Node root)
{Node temp;
    temp  = root.right;
    root.right = temp.left;
    temp.left = root;
    return(root);
}
```





Adjust AVL Tree(Rotate left)

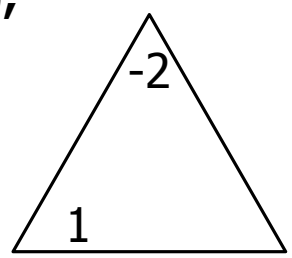
- **14, 5, 20, 25, 16, 2, 9, 10, 7, 12**
  - **เมื่อเพิ่มโหนด 12 จะทาให้โหนด 5 ไม่สมดุลทางขวา (Balance factor = 2)**
  - **ทางขวาของโหนด 5มีค่าบาลานซ์เป็น 1**
  - **ต้องแก้ด้วยการหมุนโหนดไปทางขวา (Balance factor เป็น 2 กับ 1)**
  - **หมุนโหนดทางขวาของ 5 (โหนด 9) ขึ้นมาทางซ้ายแล้วย้ายโหนด 5 ลง**

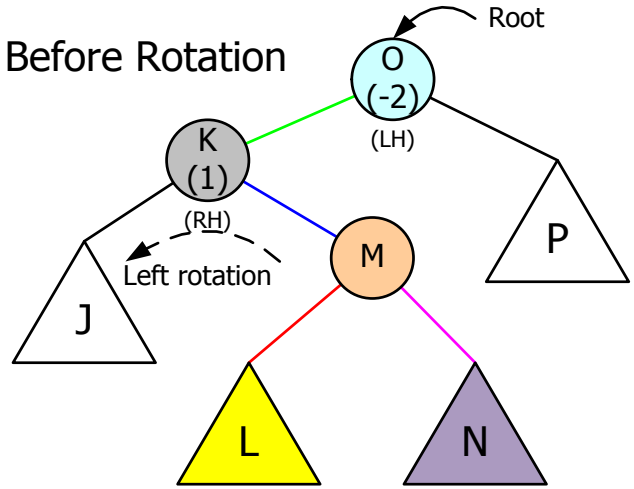


Left to Right Rotation

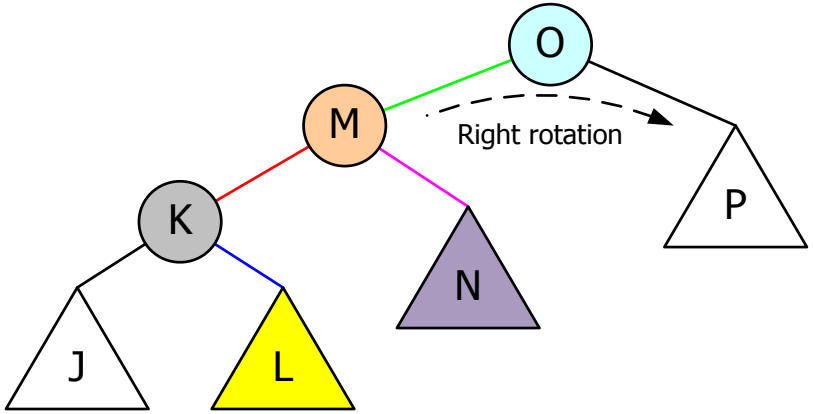
- ***Left to Right Rotation for right of left***

```
Node  rotateLeftToRight (Node root)
{ root.left  = rotateLeft (root.left);
  root = rotateRight(root);
  return(root);
}
```



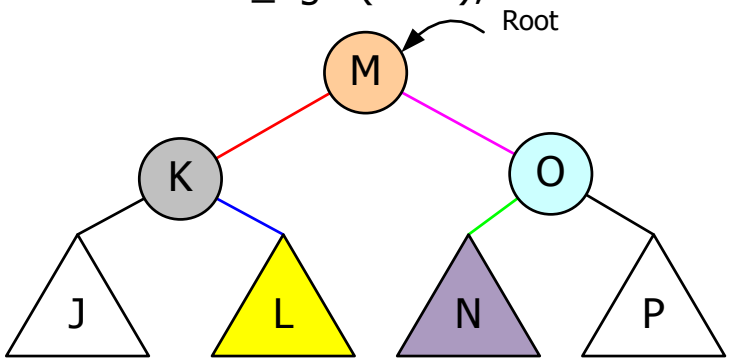


Root->left = rotate_left(Root->left);



After 1st Rotation(Rotate Left)

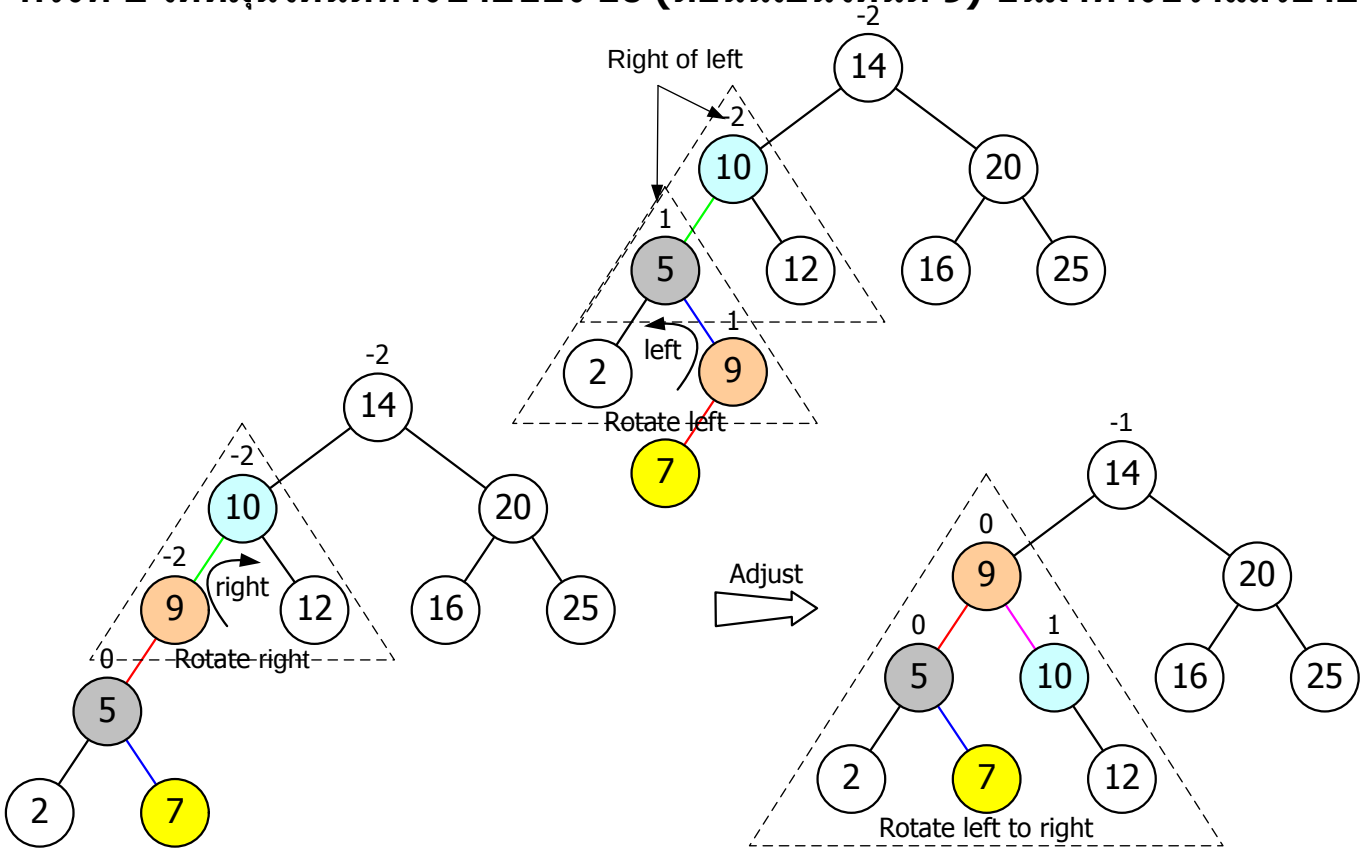
Root = rotate_right(Root);



After 2nd Rotation(Rotate Right)

Adjust AVL Tree(left to right)

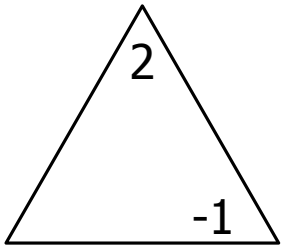
- **14, 10, 20, 5, 25, 16, 12, 2 , 9, 7**
  - **เมื่อเพิ่มโหนด 7 จะทาให้โหนด 10 ไม่สมดุลทางซ้าย (Balance factor = -2)**
  - **ทางซ้ายของโหนด 10 มีค่าบาลานซ์เป็น 1**
  - **ต้องแก้ด้วยการหมุน 2 ครั้ง (Balance factor เป็น -2 กับ 1)**
  - **ครั้งที่ 1 ให้หมุนโหนดทางขวาของ 5(โหนด 9) ขึ้นมาทางซ้ายแล้วย้ายโหนด 5ลง**
  - **ครั้งที่ 2 ให้หมุนโหนดทางซ้ายของ 10 (ตอนนี้เป็นโหนด 9) ขึ้นมาทางขวาแล้วย้ายโหนด 10 ลง**

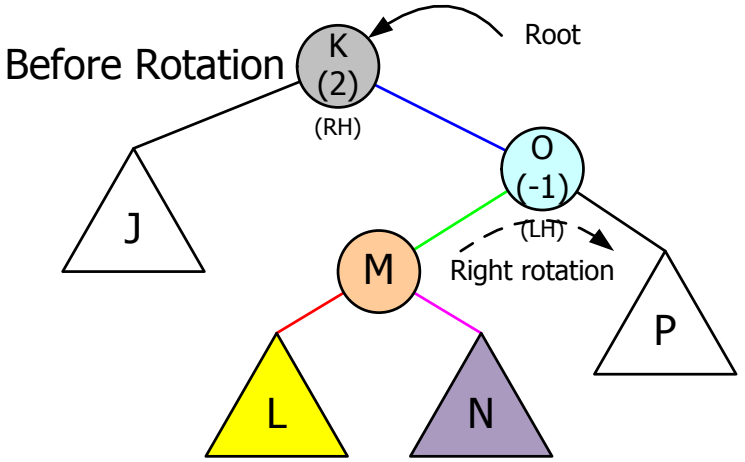


Right to Left Rotation

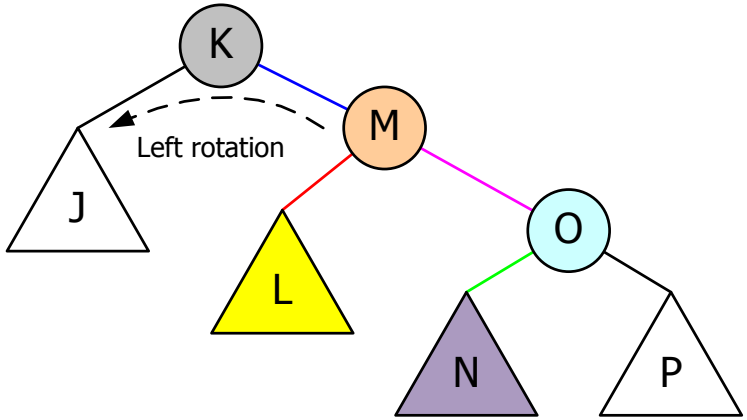
- ***Right to Left Rotation for left of right***

```
Node  rotateRightToLeft (Node root)
{ root.right  = rotateRight (root.right);
  root = rotateLeft(root);
  return (root);
}
```



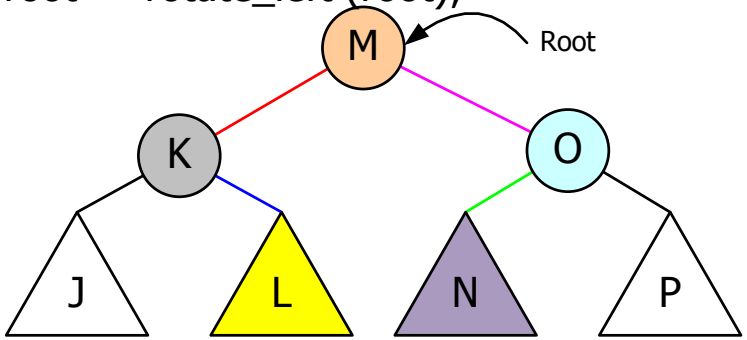


root->right = rotate_right (root ->right);



After 1st Rotation(Rotate_Right)

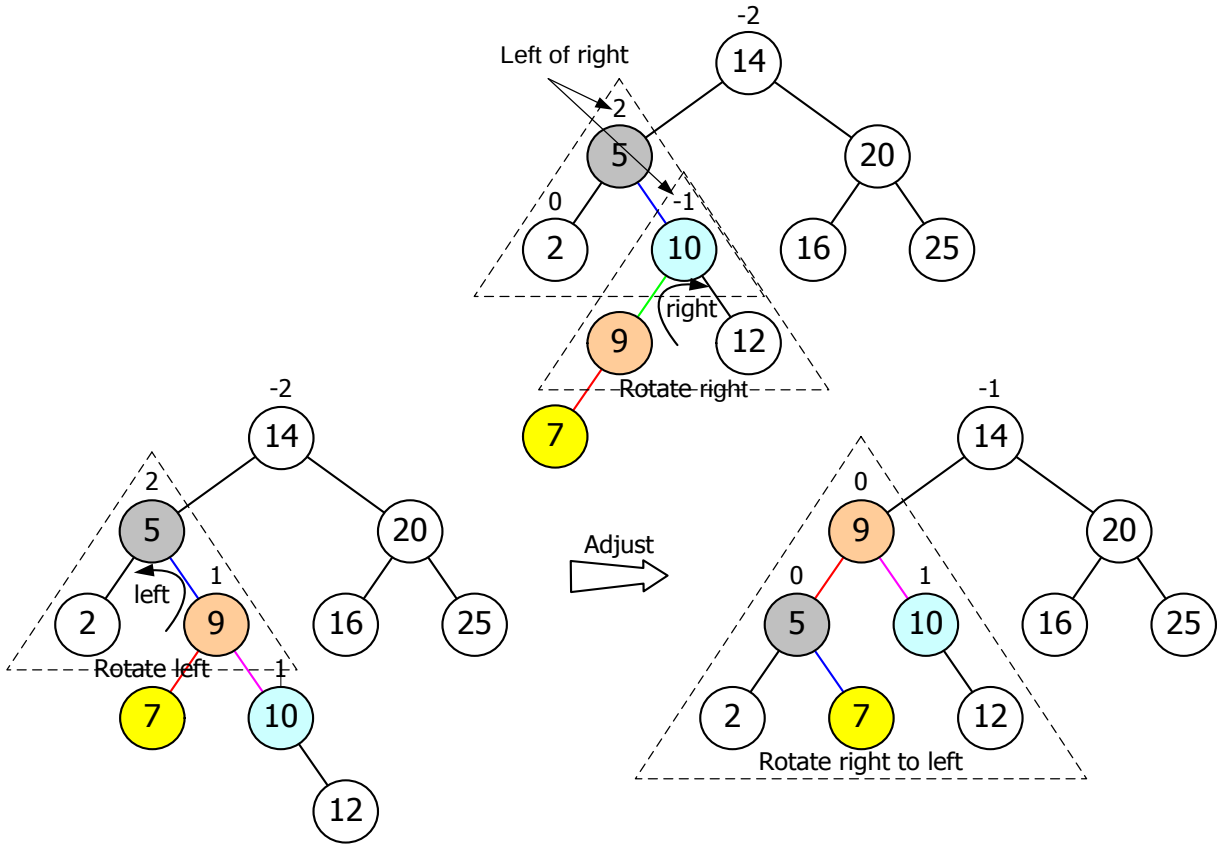
root = rotate_left (root);



After 2nd Rotation (Rotate_Left)

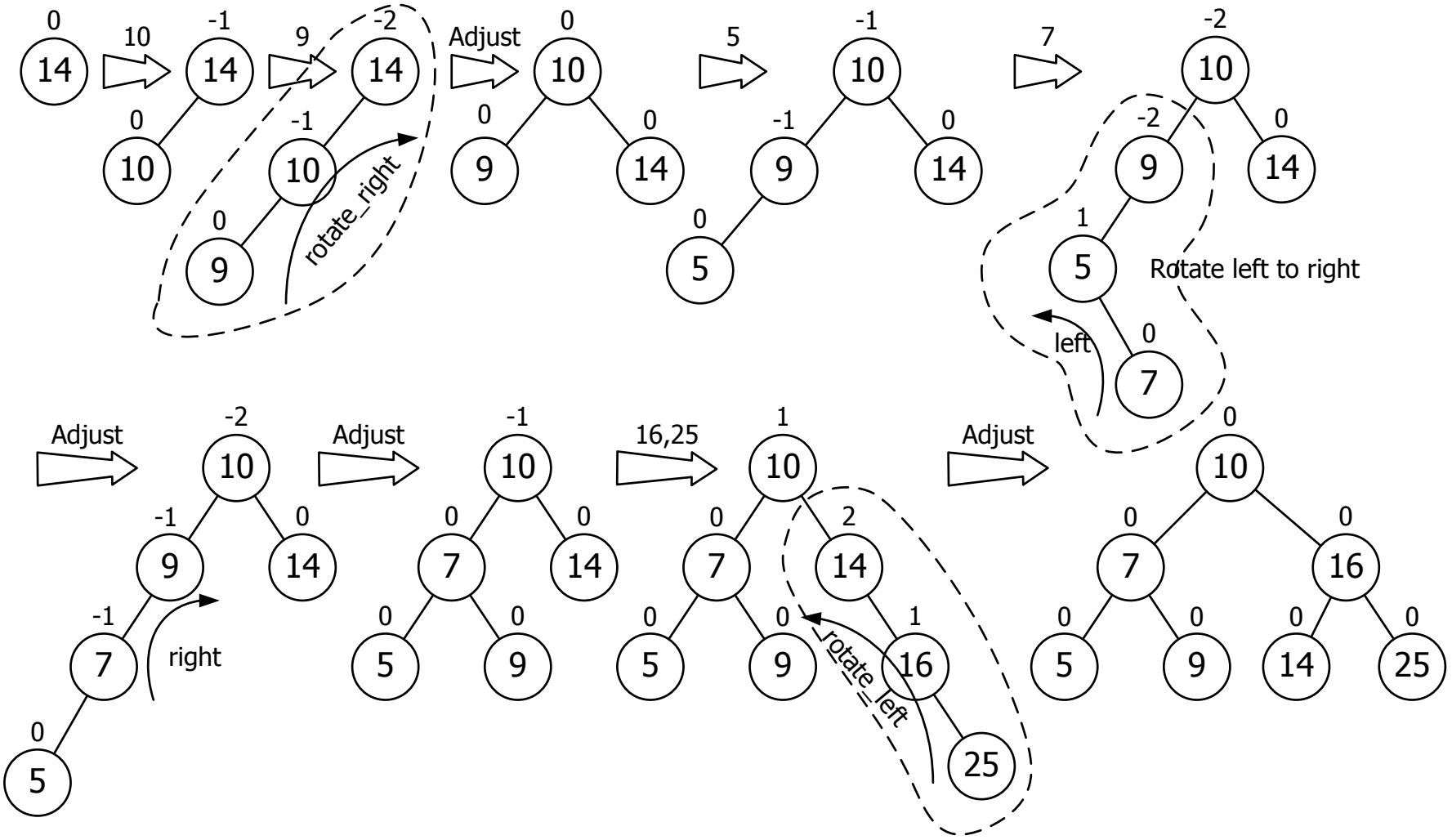
Adjust AVL Tree (right to left)

- **14, 5, 20, 2, 10, 16, 25, 9, 12, 7**
  - **เมื่อเพิ่มโหนด 7 จะทาให้โหนด 5 ไม่สมดุลทางขวา (Balance factor = 2)**
  - **ทางขวาของโหนด 5 มีค่าบาลานซ์เป็น -1**
  - **ต้องแก้ด้วยการหมุน 2 ครั้ง (Balance factor เป็น 2 กับ -1)**
  - **ครั้งที่ 1 ให้หมุนโหนดทางซ้ายของ 10 (โหนด 9) ขึ้นมาทางขวาแล้วย้ายโหนด 10ลง**
  - **ครั้งที่ 2 ให้หมุนโหนดทางขวาของ 5 (ตอนนี้เป็นโหนด 9) ขึ้นมาทางซ้ายแล้วย้ายโหนด 5 ลง**



Making AVL Tree

- **14, 10, 9, 5, 7, 16, 25**



Multiway Trees

- **Binary Search tree มีตัวเปรียบเทียบ 1 ตัว มีโหนดลูก 2 โหนด**
- **Multiway Search tree order m หรือเรียกว่า m-way search tree คือ multiway tree ที่มีคุณสมบัติคือ**

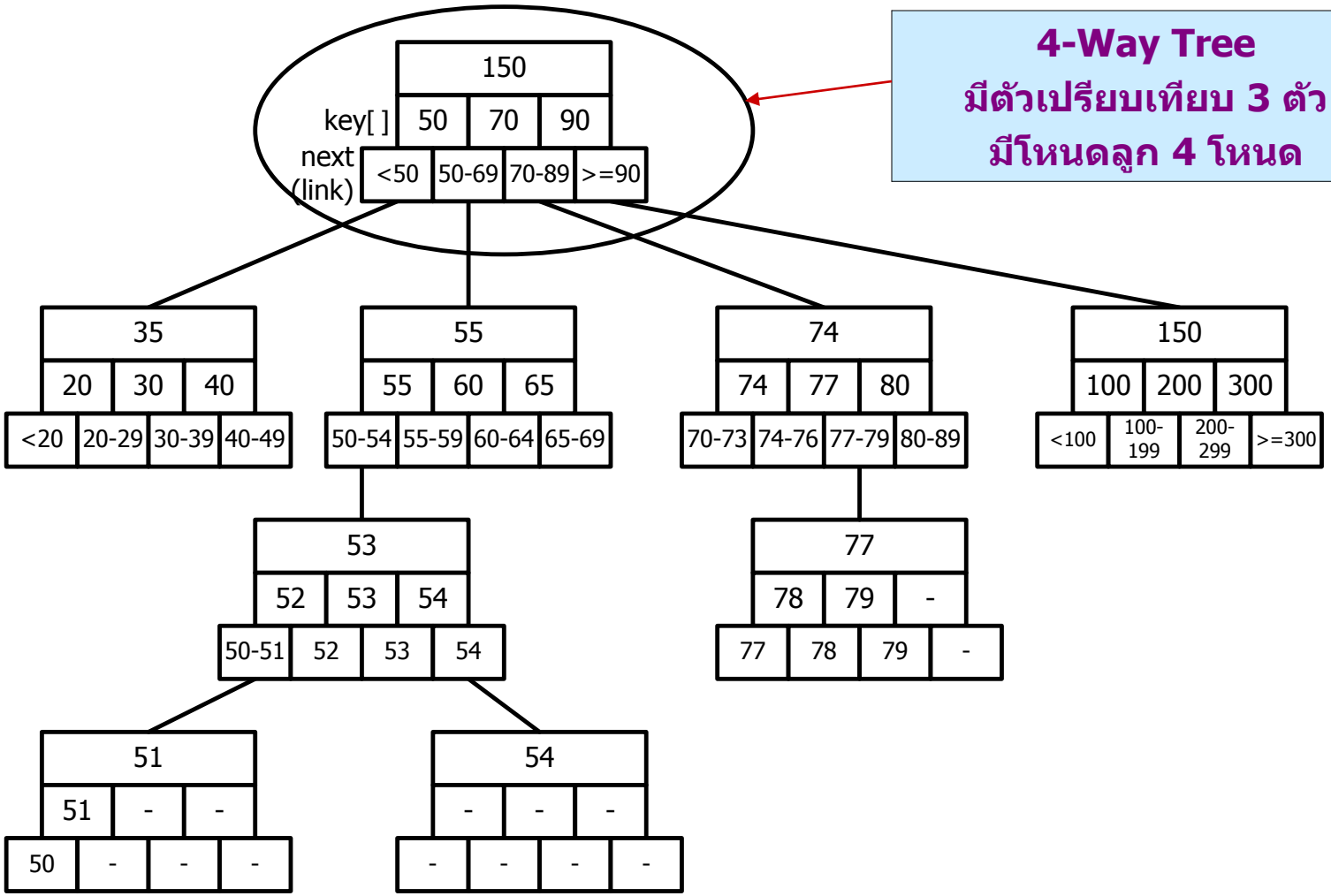
  1. มีโหนดลูก m ตัว และมีข้อมูลเปรียบเทียบ m-1 ตัว

  2. ค่าคีย์ของแต่ละโหนดจะต้องเรียงลำดับจากน้อยไปมาก

  3. ค่าคีย์ของโหนดแรก จะต้องน้อยกว่าค่าคีย์ของโหนดอื่น

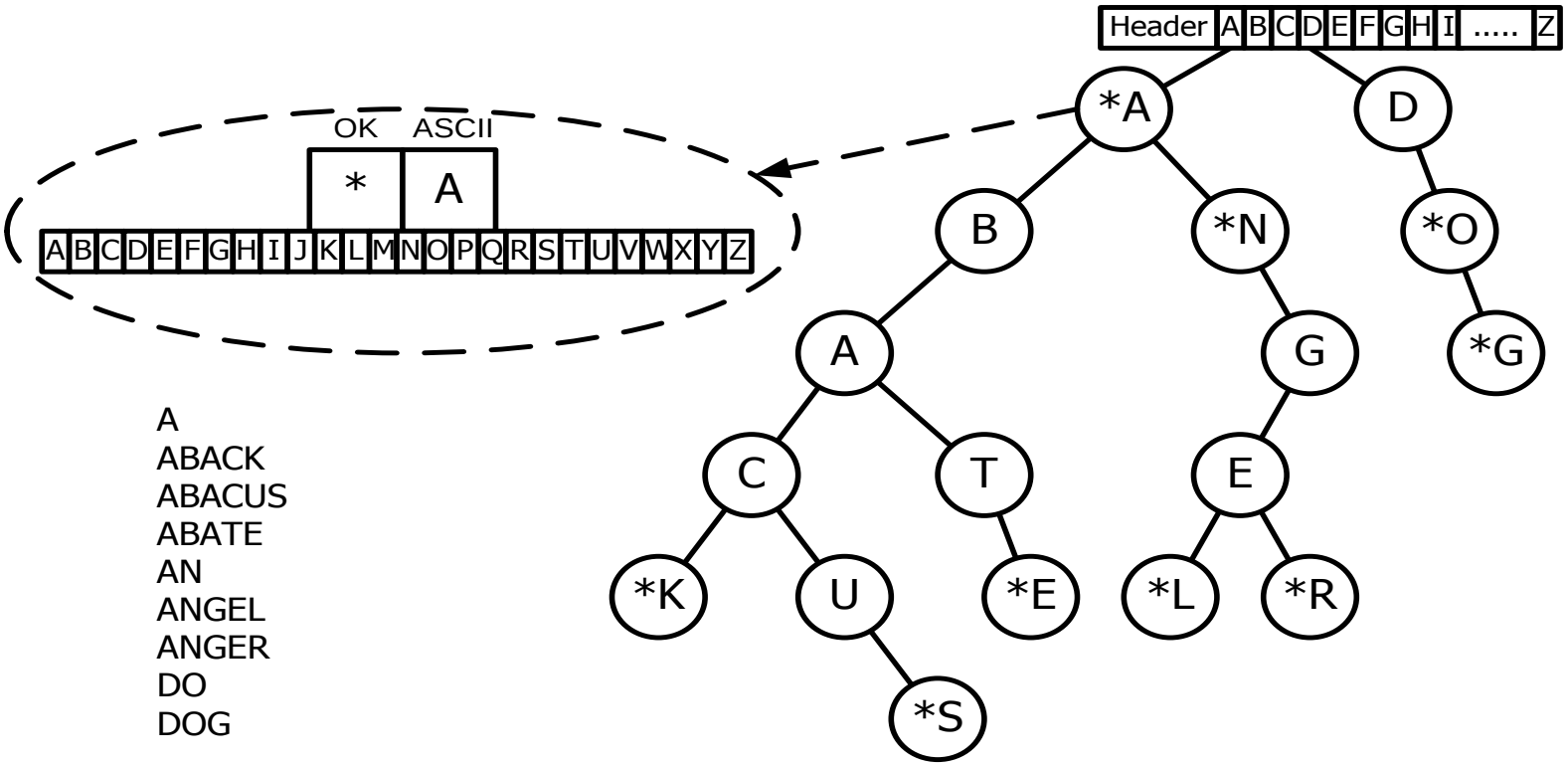
  4 ค่าคีย์ของโหนดสุดท้าย จะต้องมากกว่าค่าคีย์ของโหนดอื่น
- **B -Trees คือ multiway search tree ที่จัดการข้อมูลอยู่ใน External Storage (มีปริมาณข้อมูลใหญ่มาก เกินกว่าจะเก็บในหน่วยความจำหลัก สามารถสร้างให้มีโหนดลูกได้ไม่จำกัด)**
- **B+ -Trees แต่ละโหนดจะต้องมีข้อมูลอย่างน้อยครึ่งหนึ่ง**
- **B* -Tree แต่ละโหนดจะต้องมีข้อมูลอย่างน้อย 2/3**

Example 4-Way Tree



Lexicographic Search Trees :TRIES

- **Table lookup to information retrieval from a tree by using a key or part of a key to make a multiway branch**
- **In a computer , Tree will become to large**



RADIX TREE : TRIES

```
class TrieNode {
        char ascii;
        boolean ok;
        String mean;   //ArrayList <String> mean = new ArrayList<String>();
        TrieNode []  font;
    public TrieNode (char x) {
            ascii = x;
            ok = false;
            mean = null;
            font = new TrieNode[26];
            for (int i=0; i<26; i++)
                font[i]=null;
    }
}
class Trie { TrieNode root;
    public Trie() {
    root = new TrieNode('*');
    }
```

Root เป็นอะไรก็ได้

Link to TRIES

```
void linkTrie(String word, String meaning) {
TrieNode p;
      int index;
      String x = word.toUpperCase();
        p = root;
        for (int i=0; i < x.length(); i++) {
           index = x.charAt(i)-65;
           if (p.font[index] != null)
              p = p.font[index];
           else { p.font[index] = new TrieNode(x.charAt(i));
                p = p.font[index]; }
        }
        p.ok = true;
        p.mean = meaning;
}
```

เปลี่ยนคำศัพท์ให้เป็นอักษรตัวใหญ่

เปลี่ยนค่า 'A' เป็น index หมายเลข 0
..... 'B' 1

Set ให้รู้ว่าคำศัพท์สมบูรณ์ที่โหนดนี้

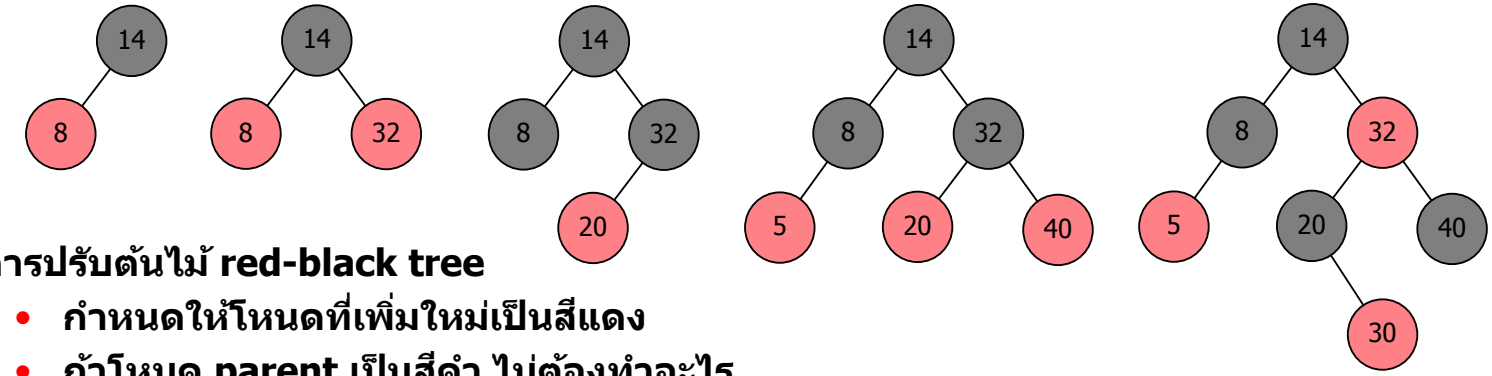
ความหมายคำศัพท์ที่สมบูรณ์ในโหนดนี้

Searching in TRIES

```
String searchTrie(String word) {
        TrieNode p;
        int i,k;
        String    str = word.toUpperCase();
                  p = root;
                  i = 0;
                  k = str.charAt(i)-65;
                  while ((p.font[k] != null) && (i < word.length())) {
                          p = p.font[k];
                          i++;
                          if (i<str.length())
                             k = str.charAt(i)  - 65;
                  }
                  if ((p.ok) && (i == str.length()))
                       return p.mean;
                  else
                       return null;
}
```

Red-Black Tree

- ต้นไม้ไบนารี ที่คิดขึ้นโดย **Rudolf Bayer** ซึ่งกำหนดสีของของโหนดให้เป็นสีแดงหรือสีดำ และมีการกำหนดเงื่อนไขเพื่อใช้ในการปรับสมดุล
  - โหนดของต้นไม้ ถูกกำหนดให้เป็นสีแดง หรือสีดำ เท่านั้น
  - ราก(Root) ต้องเป็นสีดำ
  - ลูกของโหนดที่เป็นสีแดง ต้องเป็นสีดำ (ถ้าลูกของโหนดสีแดงเป็นสีแดงจะต้องปรับสมดุลใหม่)
  - ทุกๆทางเดินจาก root ไปยัง null node จะต้องผ่านโหนดสีดำเท่ากัน

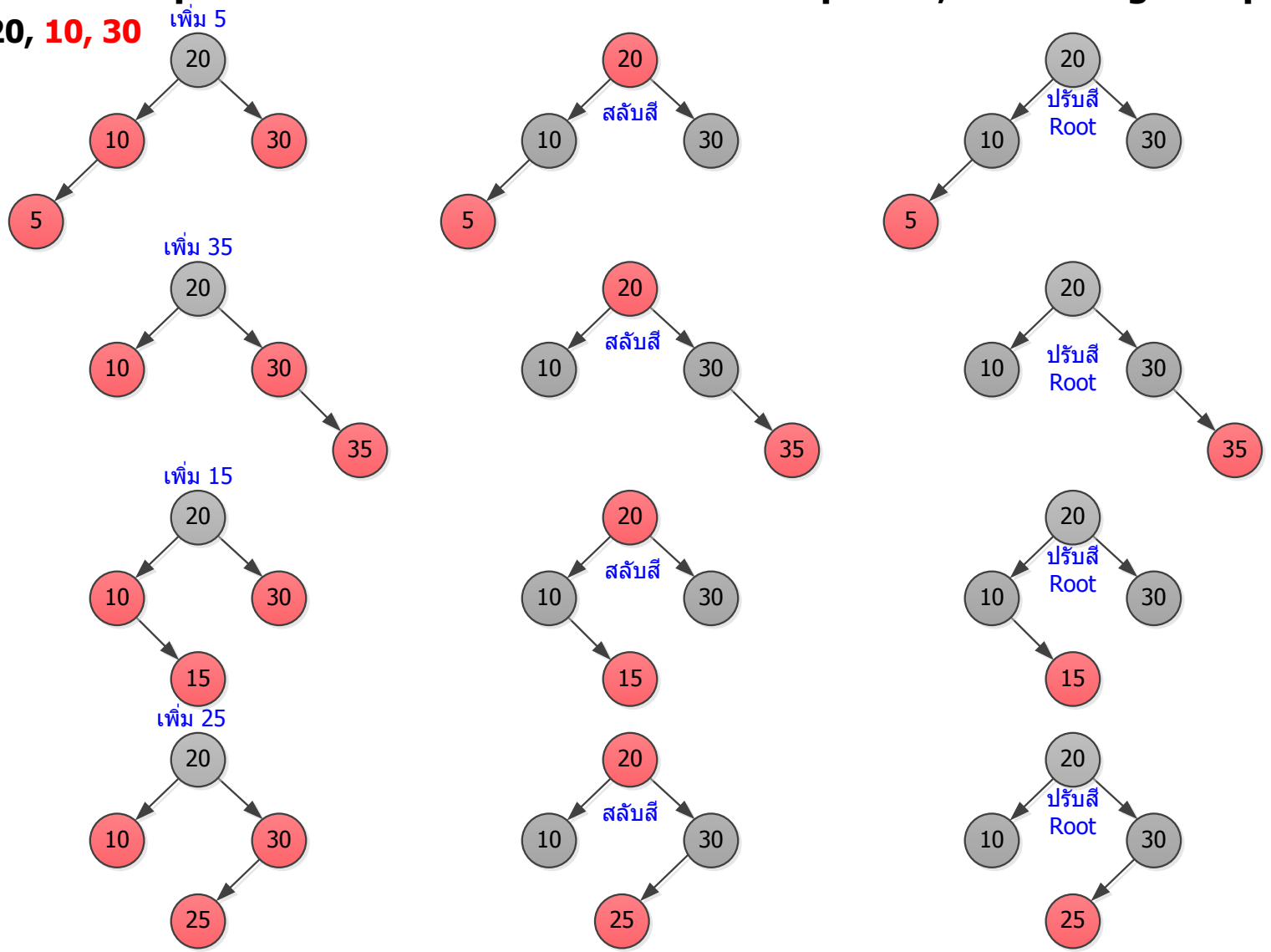


- การปรับต้นไม้ red-black tree
  - กำหนดให้โหนดที่เพิ่มใหม่เป็นสีแดง
  - ถ้าโหนด parent เป็นสีดำ ไม่ต้องทำอะไร
  - ถ้าโหนด parent เป็นสีแดง แสดงว่าไม่เป็นไปตามกฎ(ไม่สมดุล) ให้พิจารณาดังนี้
    - กรณีที่ 1 ถ้ามีโหนด uncle สีแดง ให้สลับสีกับ grand parent
    - กรณีที่ 2 ถ้าไม่มีโหนด uncle (หรือ uncle สีดำ) และเป็นโหนดด้านนอก ให้หมุนต้นไม้ 1 ครั้ง แล้วปรับสี
    - กรณีที่ 3 ถ้าไม่มีโหนด uncle (หรือ uncle สีดำ) และเป็นโหนดด้านใน ให้หมุนต้นไม้ 2 ครั้ง แล้วปรับสี
  - ถ้าปรับสีแล้วทำให้ Root เป็นสีแดง ให้เปลี่ยน Root เป็นสีดำ
  - ถ้าปรับสีแล้วทำให้เกิดสีแดงซ้อนกันอีก ให้ปรับสมดุลใหม่อีก

Red-Black Tree

- **Case 1 parent และ uncle สีแดง ให้สลับสีโหนด parent, uncle กับ grand parent**

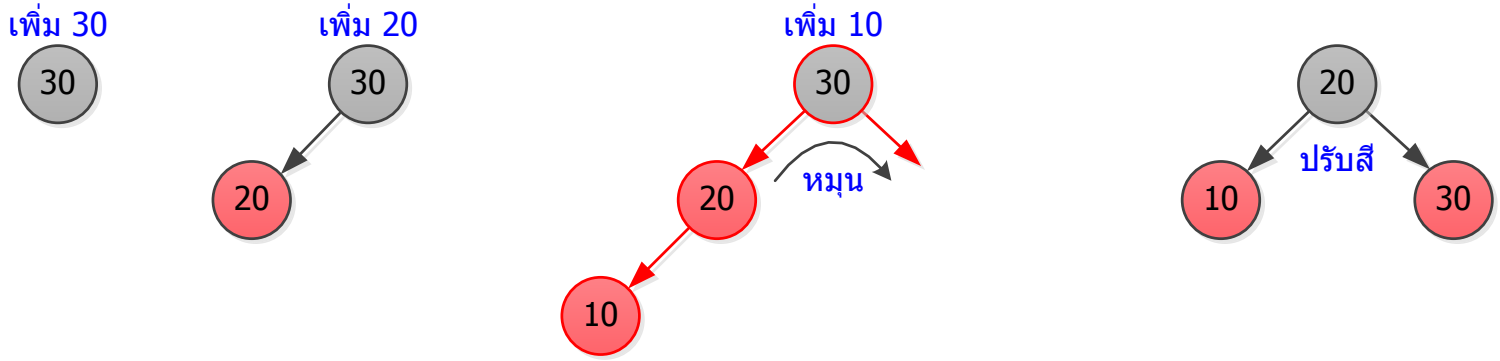
20, 10, 30



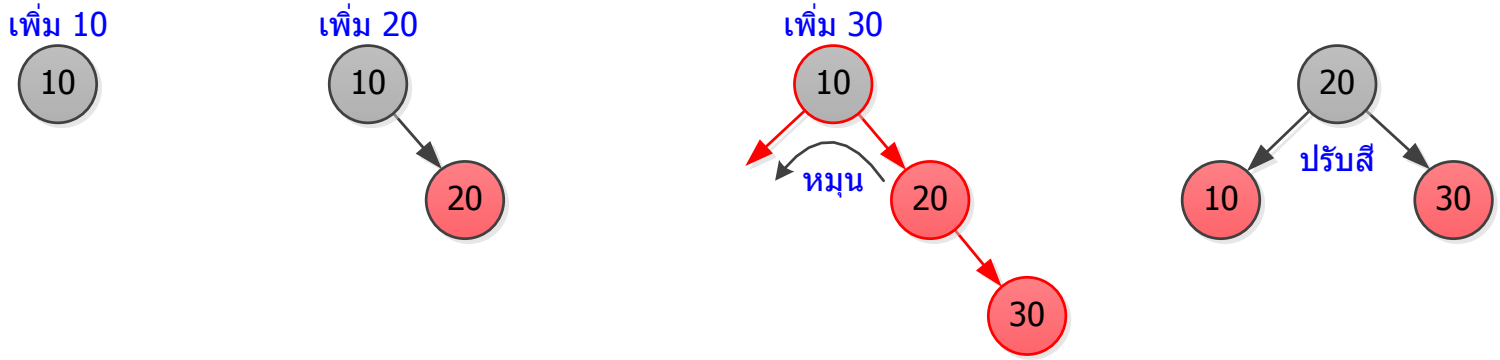
Red-Black Tree

- **Case 2 โหนดด้านนอก ไม่มี uncle(หรือมีสีดำ) ให้หมุน 1 ครั้ง ปรับสี**

30, 20, 10

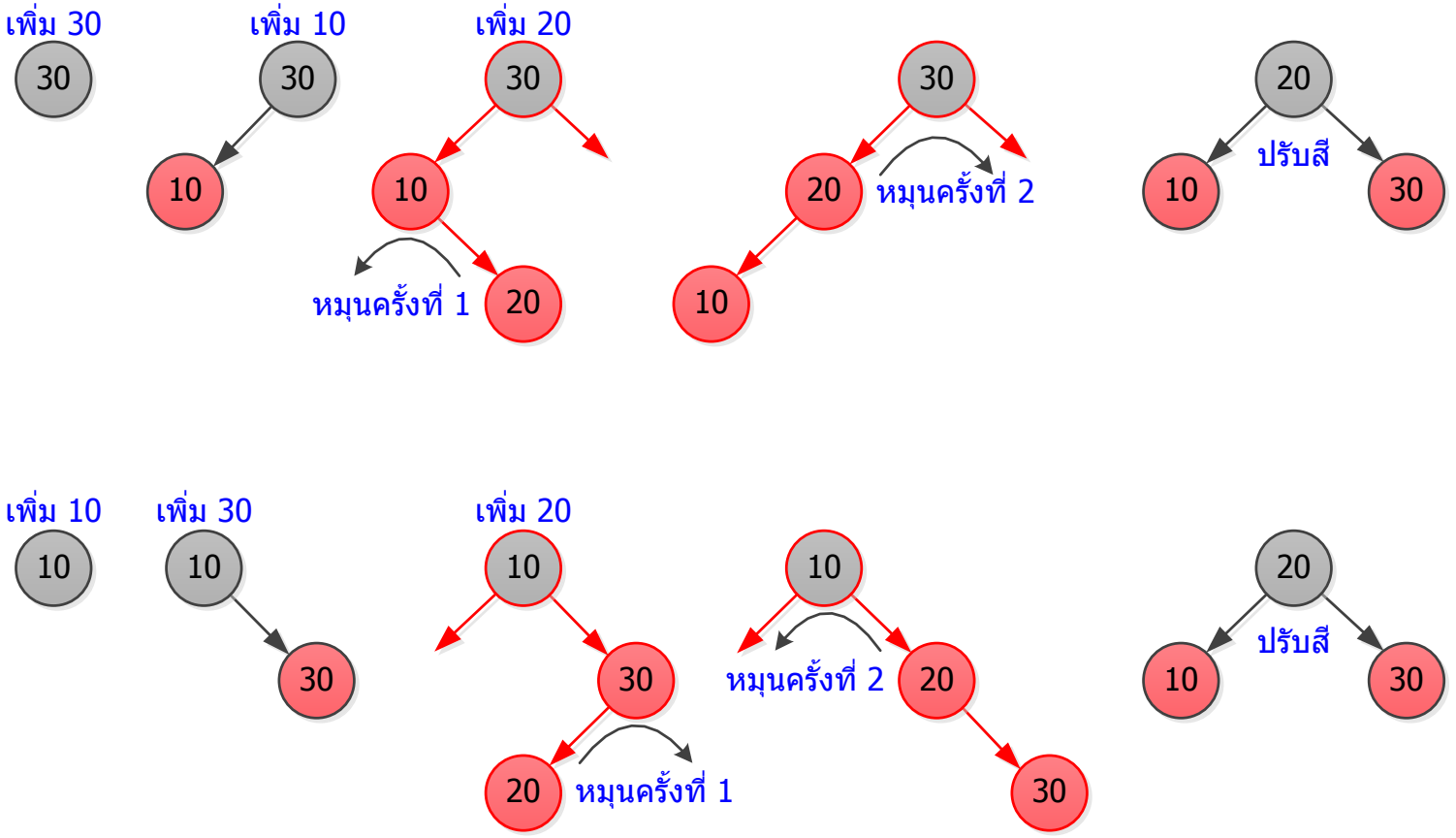


10, 20, 30



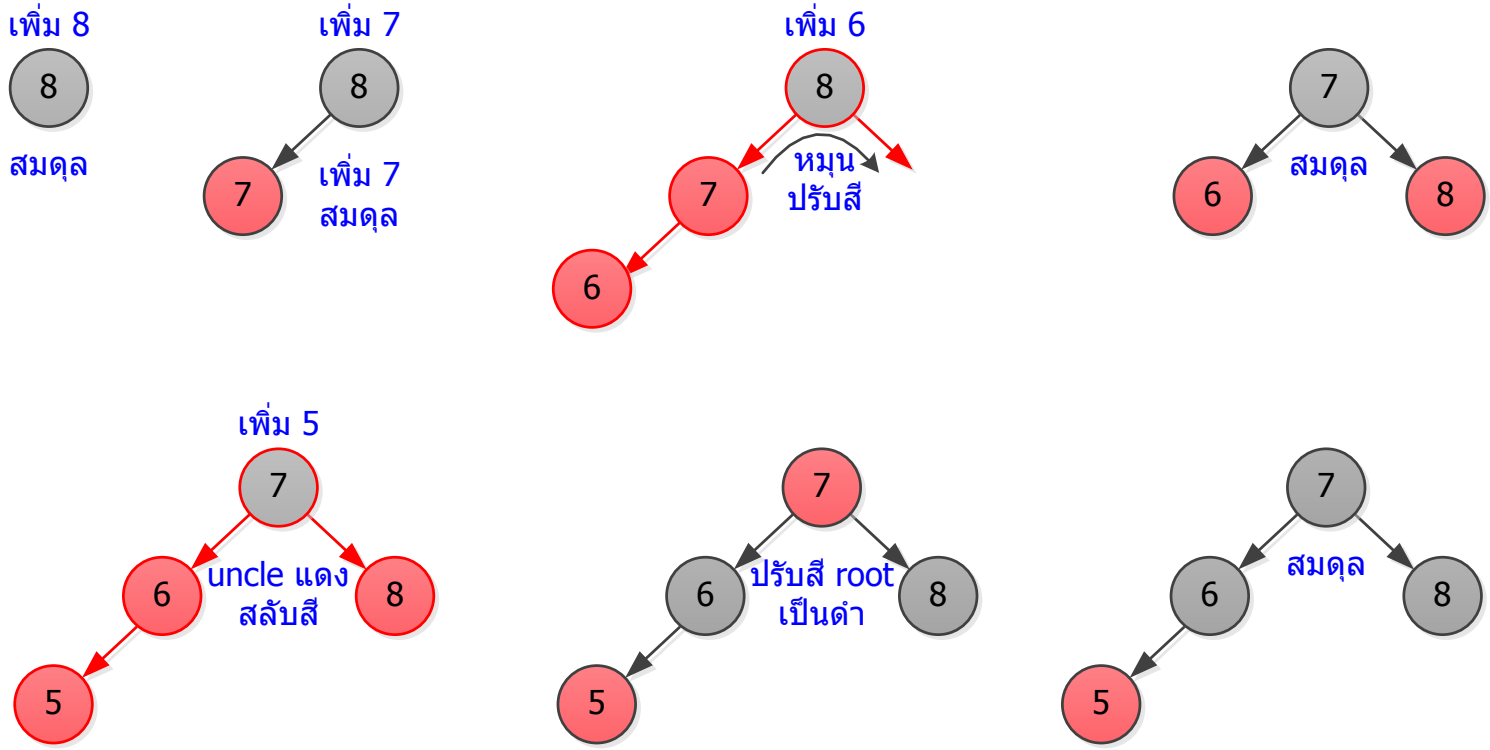
Red-Black Tree

- **Case 3 โหนดด้านใน ไม่มี uncle(หรือมีสีดำ) ให้หมุน 2 ครั้ง แล้วปรับสี**



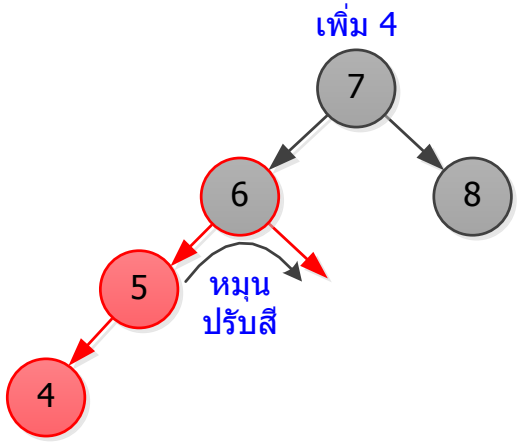
Red-Black Tree

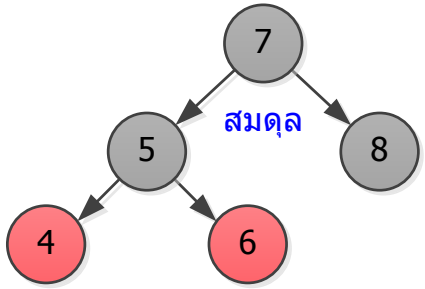
8, 7, 6, 5

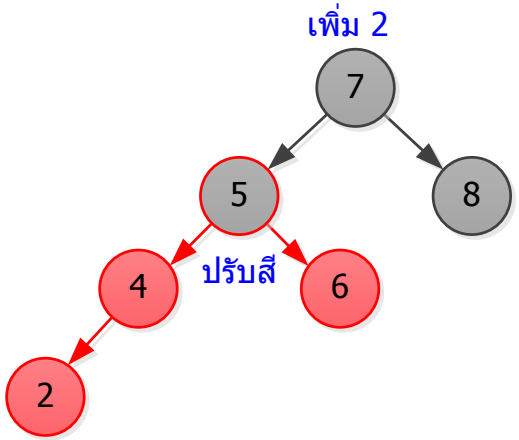


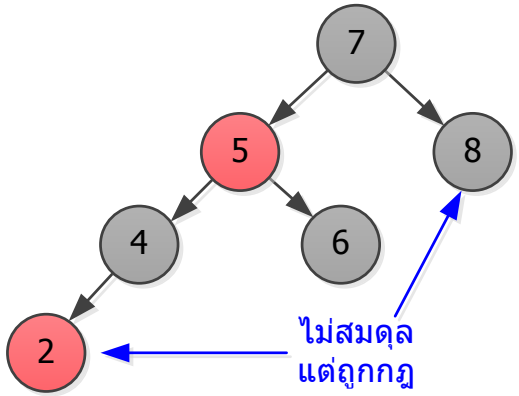
Red-Black Tree

8, 7, 6, 5, 4, 2



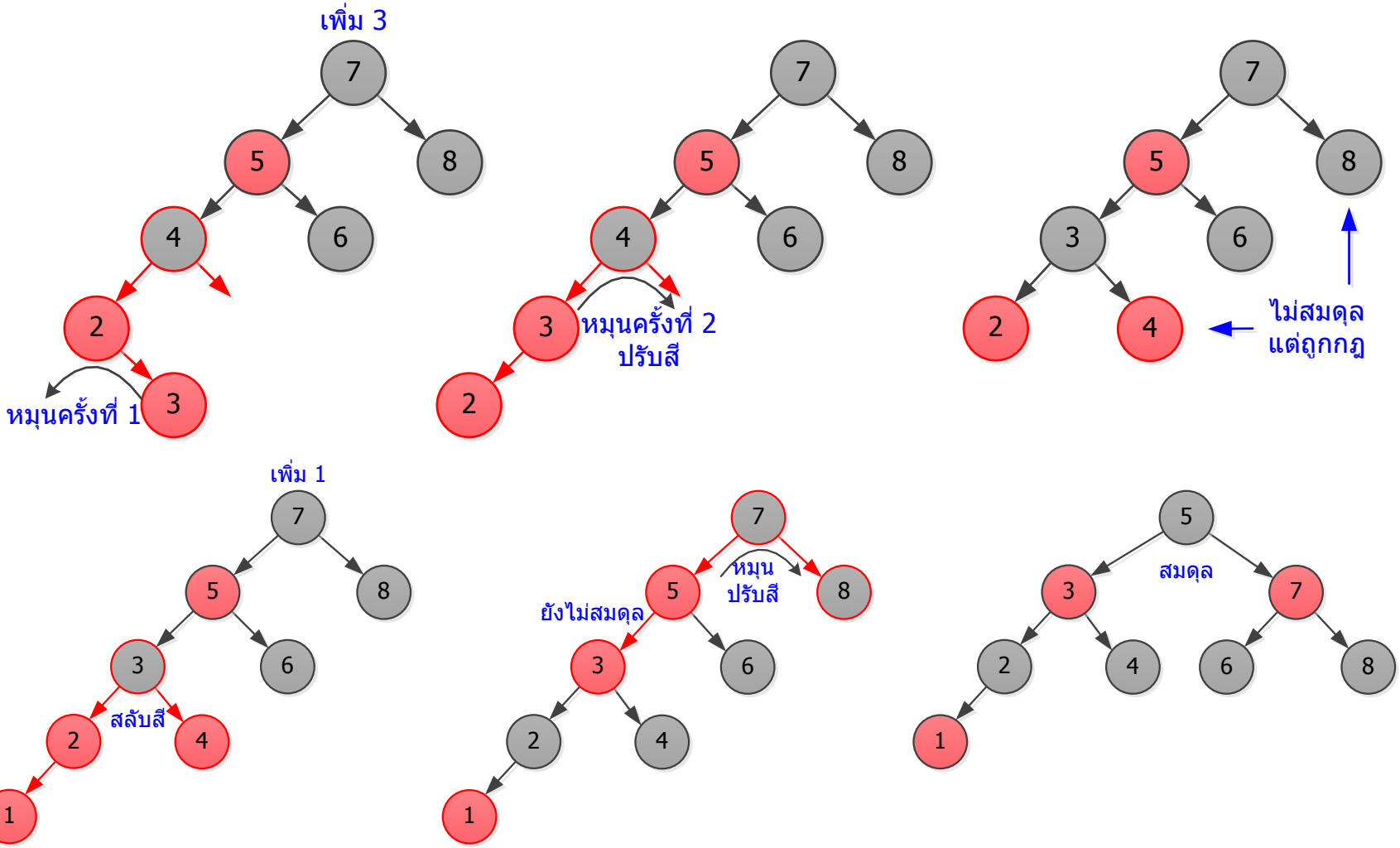






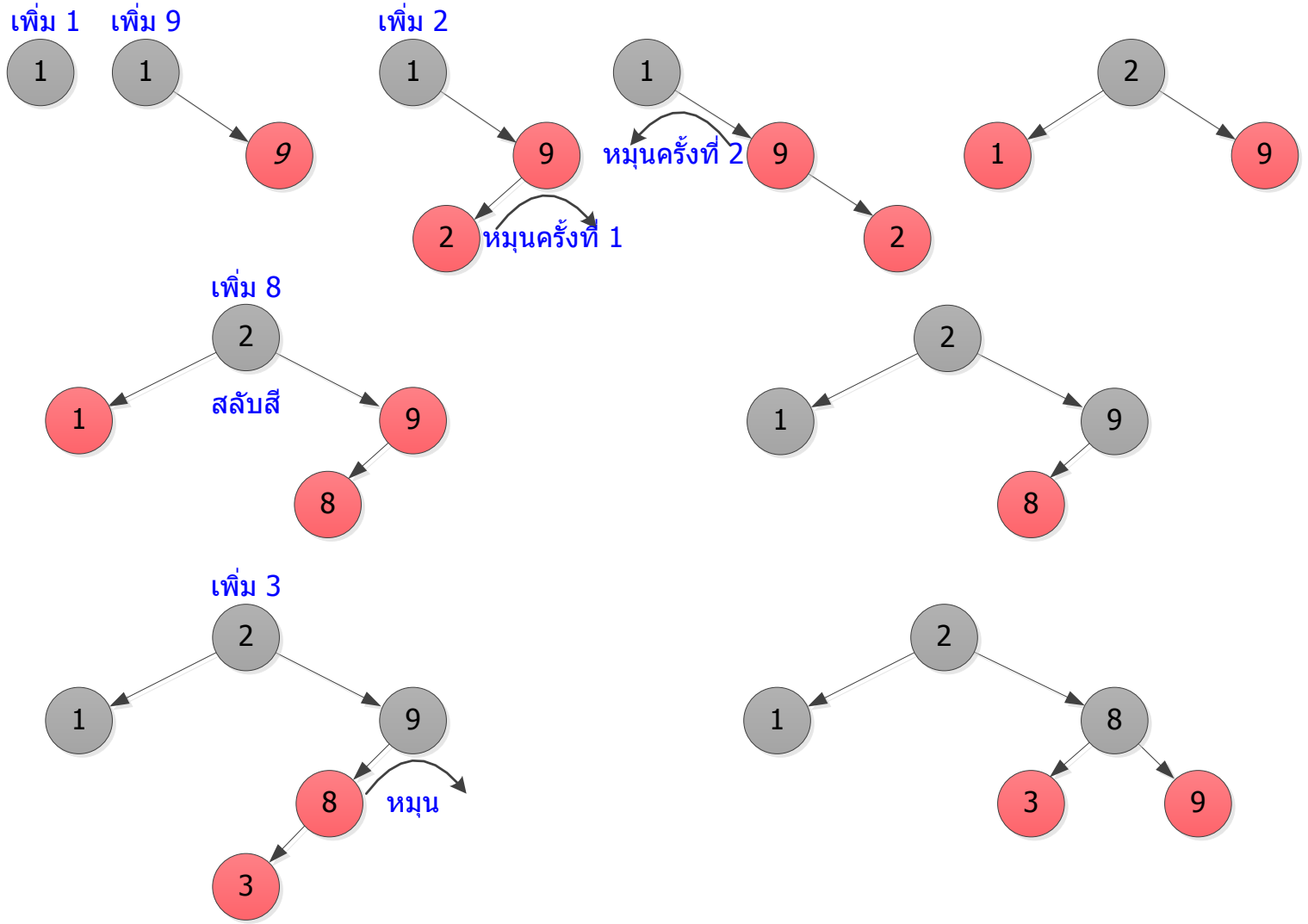
Red-Black Tree

8, 7, 6, 5, 4, 2, 3, 1



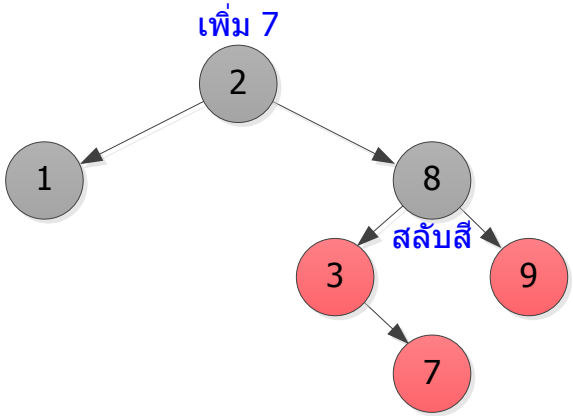
Red-Black Tree

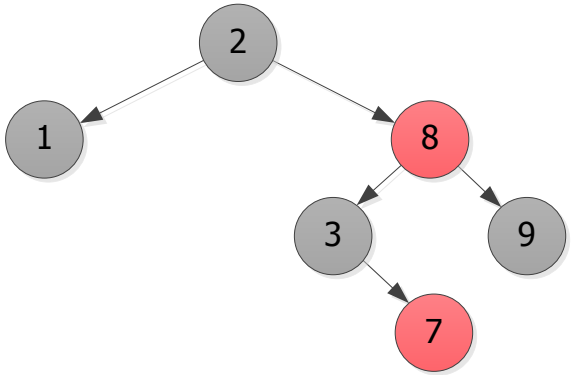
1, 9, 2, 8, 3

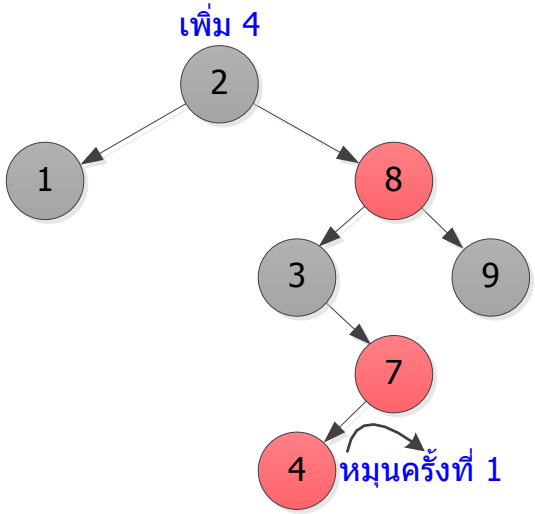


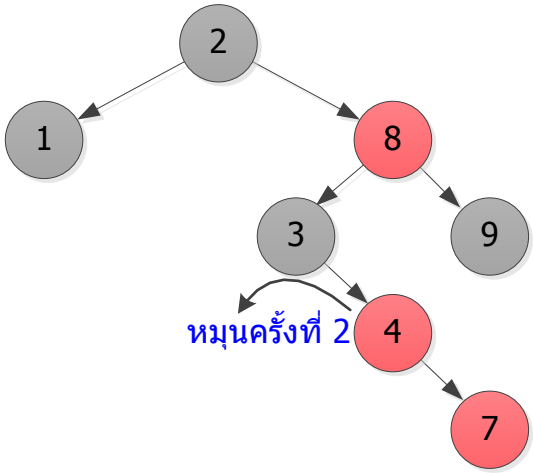
Red-Black Tree

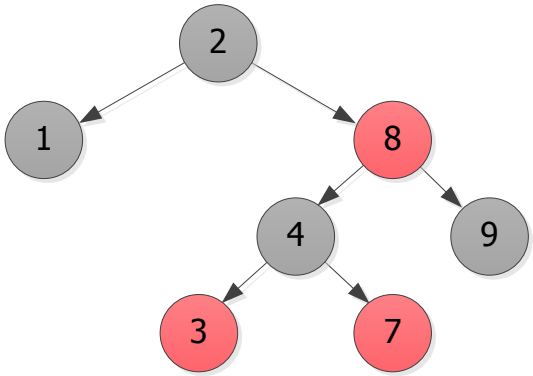
1, 9, 2, 8, 3, 7, 4





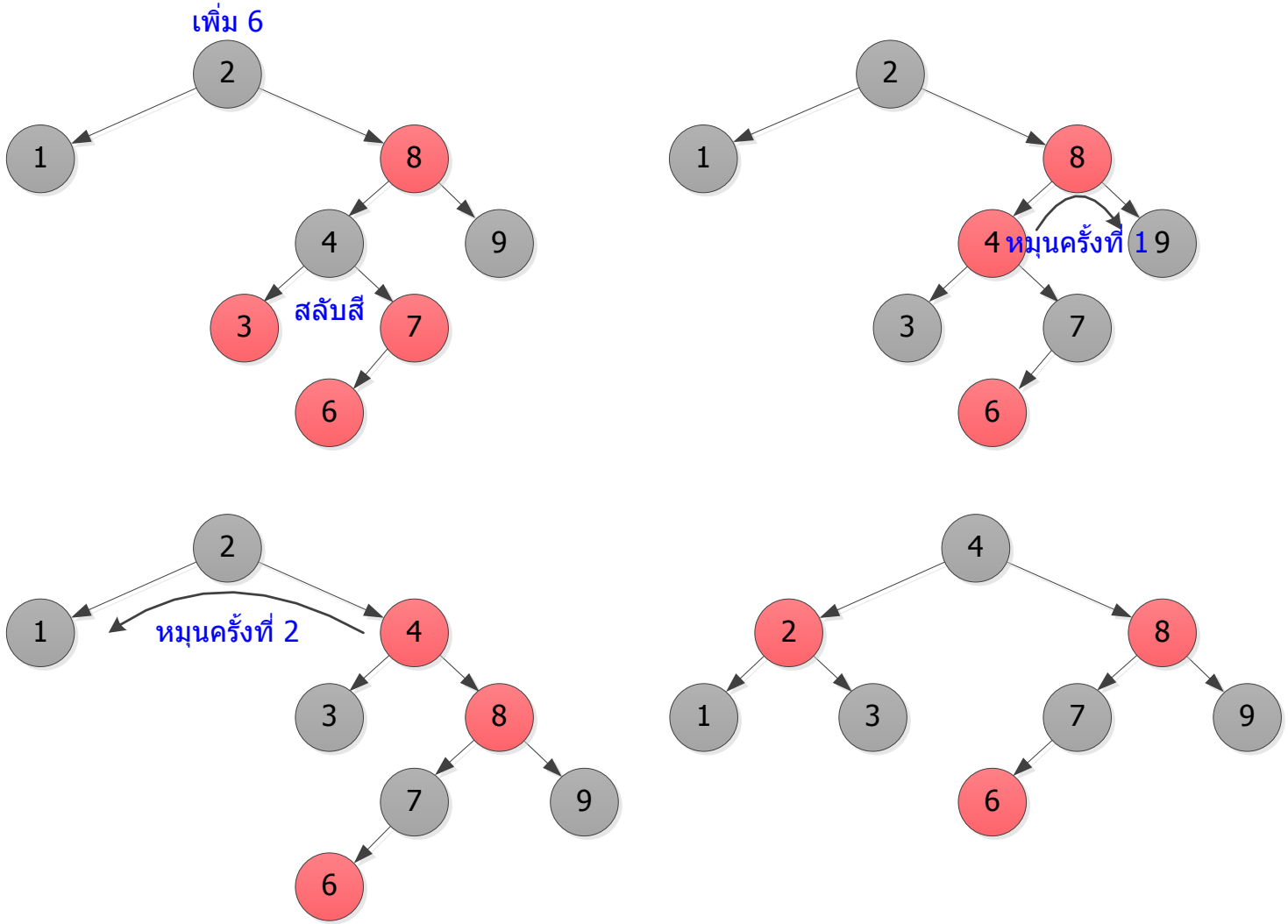






Red-Black Tree

1, 9, 2, 8, 3, 7, 4, 6



Java : TreeSet

- TreeSet เป็นคลาสที่ใช้เก็บ set ข้อมูล โดยจัดโครงสร้างแบบ Binary Tree (Red-Black)

  TreeSet<Object> Mytree = new TreeSet<Object> ();

  - ข้อมูลที่ใส่จะถูกกำหนดตำแหน่งใน Tree และเรียงลำดับจากน้อยไปมากอัตโนมัติ
  - ไม่สามารถใส่ object ที่มีคีย์ที่ซ้ำได้ (ถ้าซ้ำจะทับตัวเดิม)
  - ใช้ for .. each หรือ Iterator ในการเข้าถึงข้อมูลทีละตัวตามลำดับ

```
for ( Object  itr : Mytree )
{ ..... operate itr   .....}
```

- ตัวอย่างคำสั่งที่เกี่ยวข้องกับคลาส
  - int size() return จำนวนข้อมูลที่เก็บอยู่
  - void clear() ลบข้อมูลทุกตัว
  - boolean add(Object o) เพิ่มข้อมูลลงใน Tree
  - boolean addAll(Collection c) เพิ่มข้อมูลทีละกลุ่ม
  - boolean contains(Object o) เช็คว่ามีข้อมูลอยู่หรือไม่
  - Object First() return ข้อมูลตัวแรก(น้อยที่สุด)
  - Object Last() return ข้อมูลตัวสุดท้าย(มากที่สุด)
  - boolean remove(Object o) ลบข้อมูลตัวที่ระบุ
  - SortedSet headSet(Object toElement) return set ตั้งแต่ตัวแรกจนถึงตัวที่น้อยกว่า <
  - SortedSet tailSet(Object fromElement) return set ตั้งแต่ตัวที่มากกว่าทั้งหมด >=
  - SortedSet subSet(Object fromElement, Object toElement)

    return ตั้งแต่ fromElement <= subset < toElement
  - SortedSet subSet(Object fromElement, boolean tF, Object toElement , boolean tE)

    return subset โดยสามารถกำหนดว่าจะรวมเอาตัว fromElement และ toElement ด้วยหรือไม่ก็ได้ ถ้า boolean เป็น true จะรวมตัวที่ระบุ แต่ถ้าเป็น false จะไม่รวม

```
Iterator <Object> itr = Mytree.iterator();
while (itr.hasNext())
{ ..... operate  itr.next() ....}
```