2110211

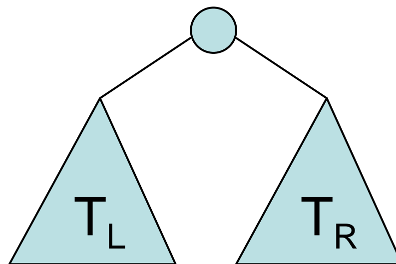
AVL Trees และ Red-Black Trees

สมชาย ประสิทธิ์จูตระกูล

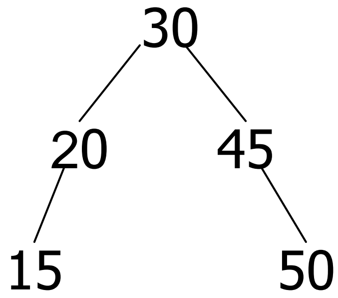
AVL Tree

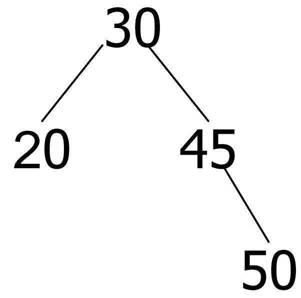
- AVL = Adel'son-Vel-skii + Landis (1962)
- AVL Tree = Binary Search Tree + กฎ
- AVL Tree
  - empty tree
  - a node with two AVL subtrees (T_L , T_R)

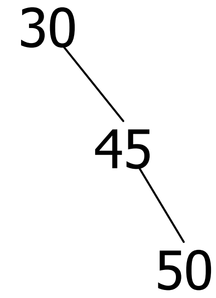
$$| \text{height}(T_L) - \text{height}(T_R) | \leq 1$$

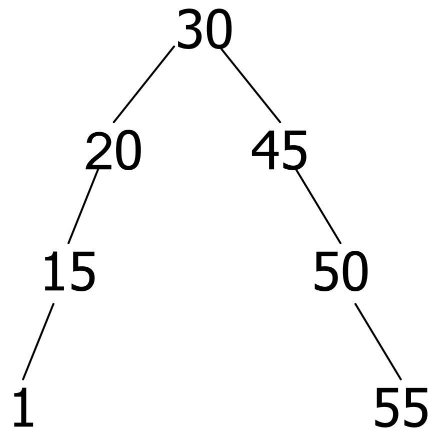


ตัวอย่าง

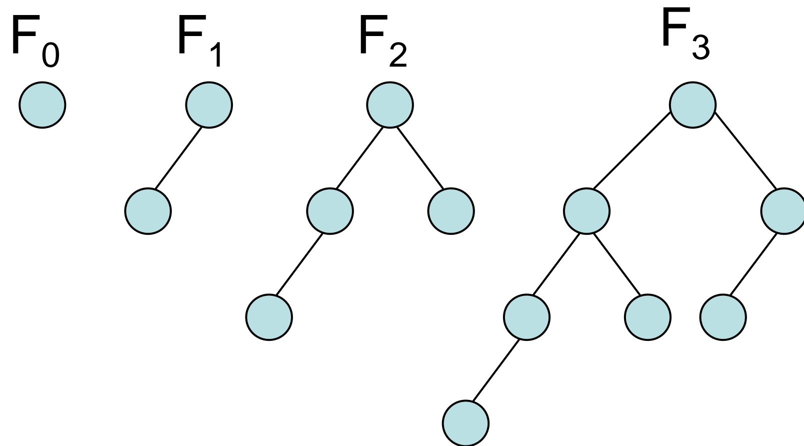








Fibonacci Trees



$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1, \quad |F_0| = 1, |F_1| = 2$$

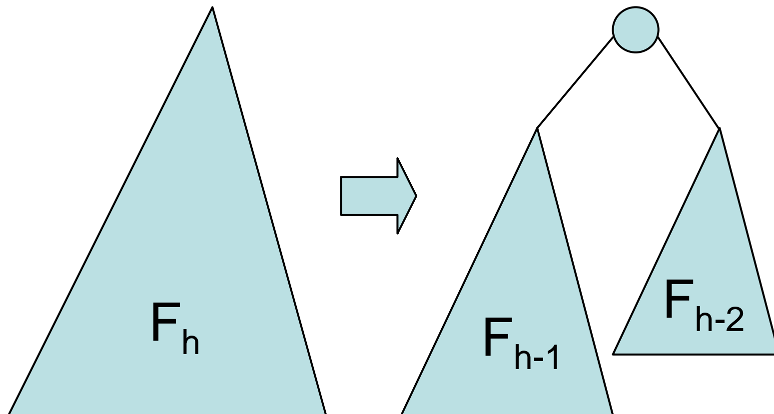
$$g_h = |F_h| + 1$$

$$g_h = g_{h-1} + g_{h-2}, \quad g_0 = 2, g_1 = 3$$

$$\approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3}$$

$$h \approx 1.44 \log(|F_h| + 1)$$

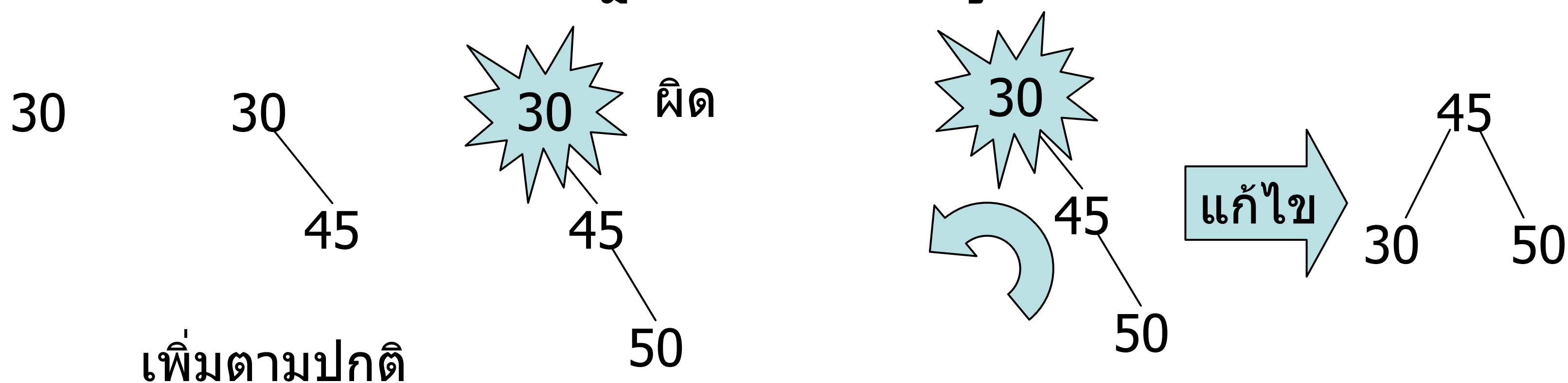
$$h \approx 1.44 \log n$$



ต้นไม้ AVL ที่มี n nodes มีความสูงเป็น O(log n)

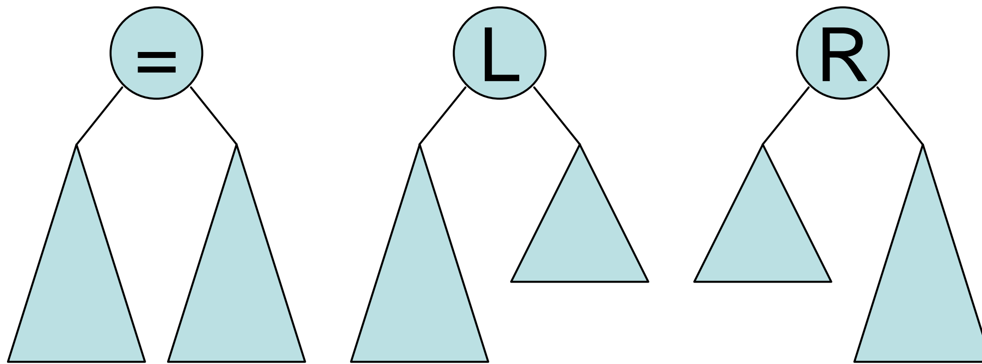
เมท็อดของ AVL

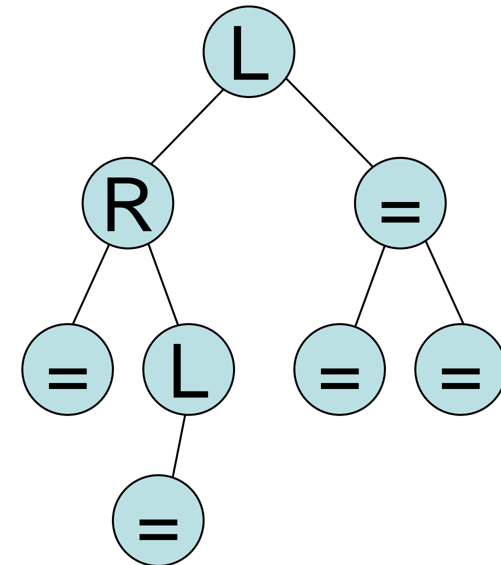
- contains, getMin, getMax, successor, ...ที่ไม่เปลี่ยนแปลงข้อมูล ยังเหมือนของ BST
- add, remove ต้องเปลี่ยน เพื่อรักษากฎ
- แนวคิด :
  - add ตามปกติ (เหมือนของ BST)
  - ทำเสร็จ ถ้าผิดกฎ ก็แก้ไขให้ถูกต้อง



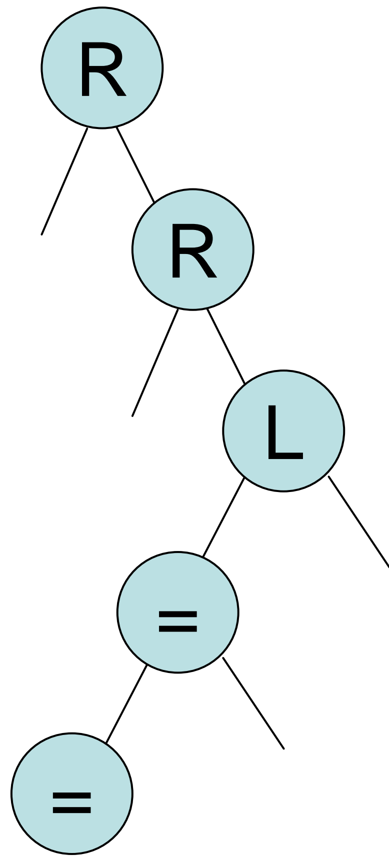
รู้ได้อย่างไรว่าผิดกฎ

- แต่ละ Entry เติม field `balanceFactor`
  - มีค่า =, L , หรือ R

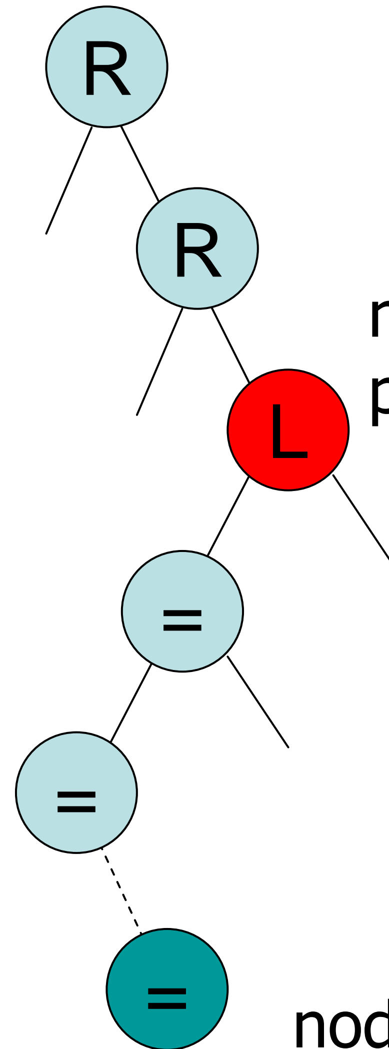




node ที่อาจผิดกฎ



ก่อนเพิ่ม



node หลังสุดใน path ที่ไม่ใช่ =

node ใหม่

```
public boolean add(Object o) {
    if (root == null) {
      root = new Entry(o, null);        ต้นไม้ว่าง
      size++; return true;
    } else {
      Entry temp = root;
      int c;
      while (true) {
        c = ((Comparable) o).compareTo(temp.element);
        if (c == 0) return false;       ข้อมูลซ้ำ
        if (c < 0) {
          ข้อมูลใหม่มีค่าน้อยกว่า เพิ่มทางซ้าย
        } else {
          ข้อมูลใหม่มีค่ามากกว่า เพิ่มทางขวา
        }
      }
    }
}
```

```java
while (true) {
  c = ((Comparable) o).compareTo(temp.element);
  if (c == 0) return false;
  if (c < 0) {
    if (temp.balanceFactor != '=') ancestor = temp;
    if (temp.left != null) {
      temp = temp.left;
    } else {
      temp.left = new Entry(o, temp);

      fixAfterInsertion(ancestor, temp.left);

      size++;
      return true;
    }
  } else {

  }
}
```

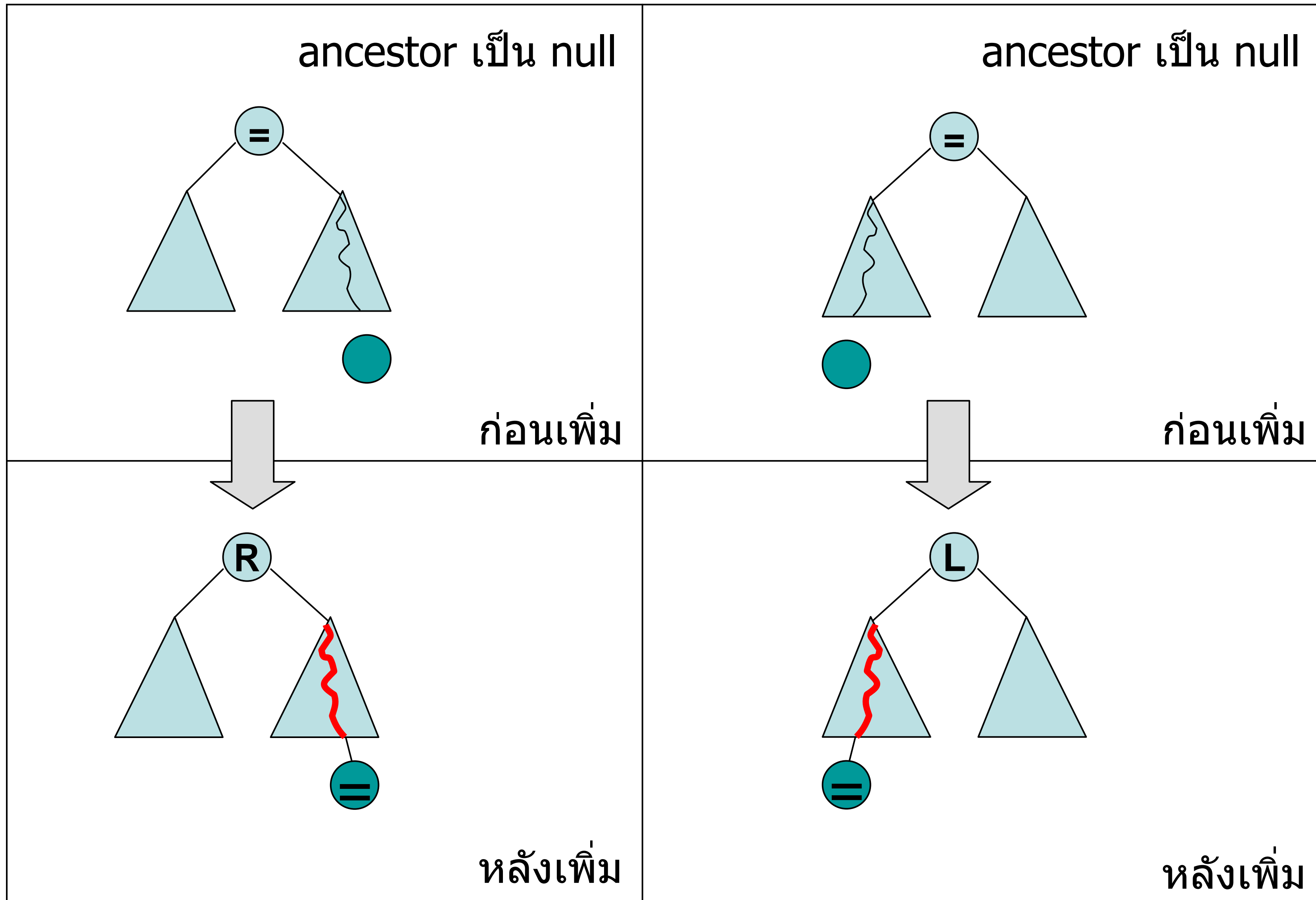
จำ node หลังสุดที่ไม่ใช่ =

เพิ่มในลักษณะเดิม

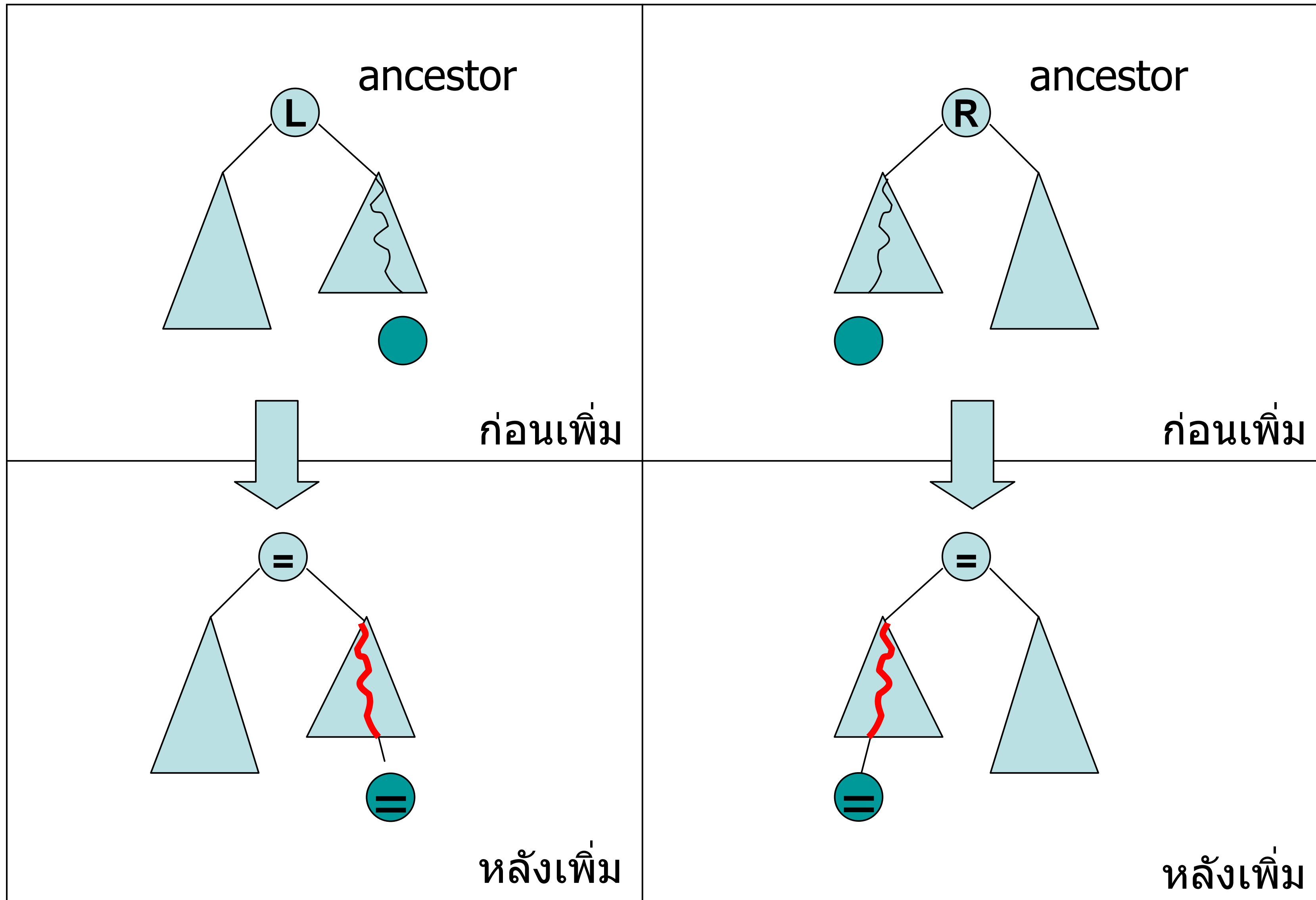
แก้ไขหลังเพิ่ม

ข้อมูลใหม่มีค่ามากกว่า เพิ่มทางขวา

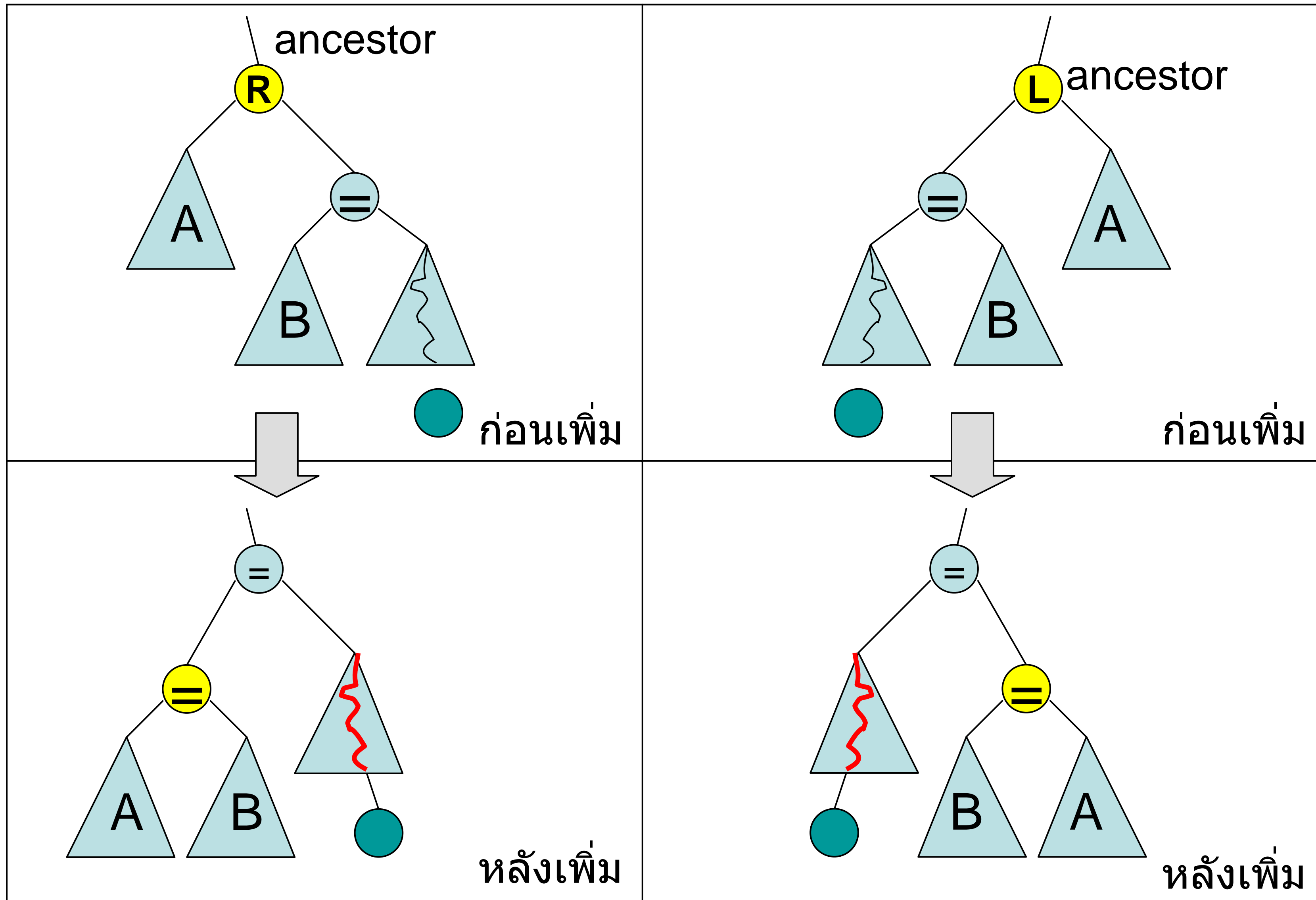
fixAfterInsertion : กรณีที่ 1



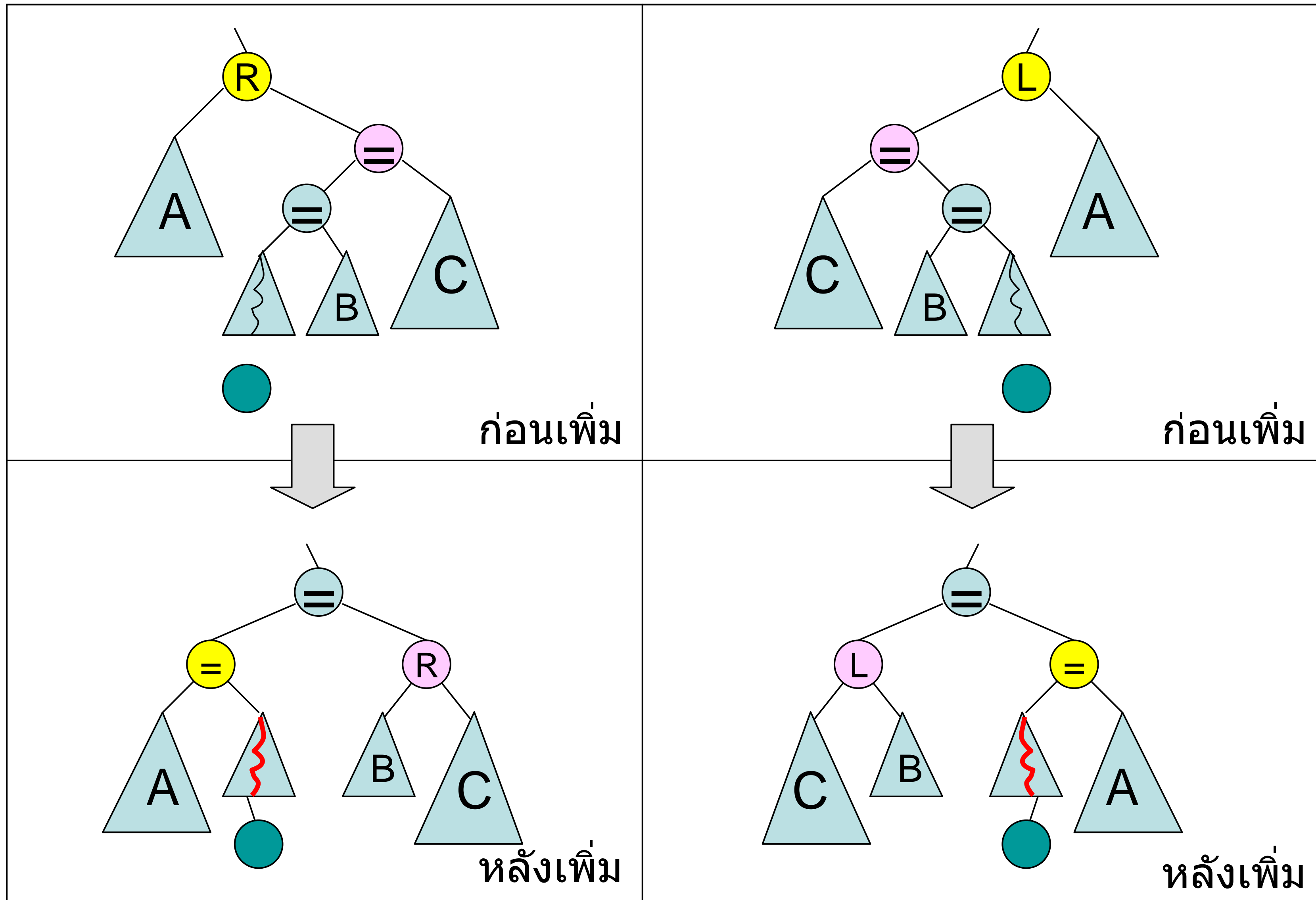
fixAfterInsertion : กรณีที่ 2



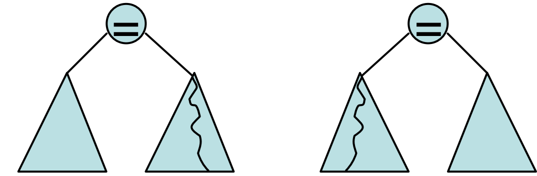
fixAfterInsertion : กรณีที่ 3, 4

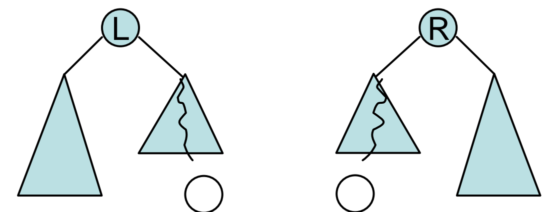


fixAfterInsertion : กรณีที่ 5, 6

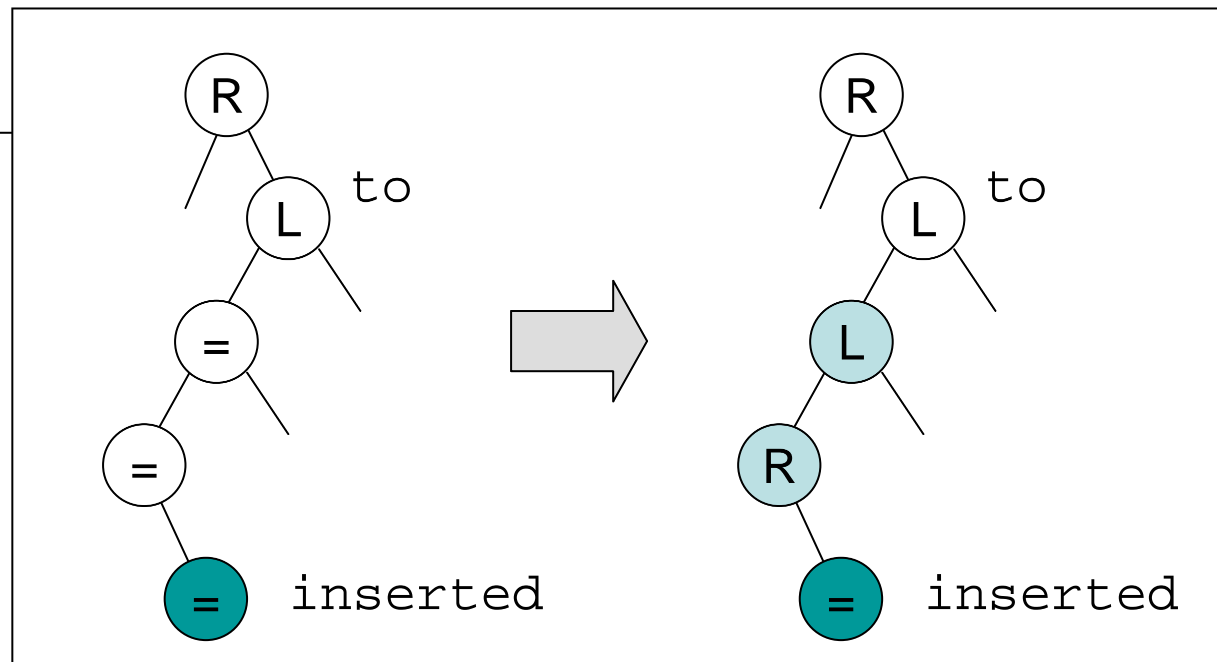


```
void fixAfterInsertion(Entry ancestor, Entry inserted) {
  Comparable o = (Comparable) (inserted.element);
  if (ancestor == null) { /* case 1 */
    root.balanceFactor = o.compareTo(root.element) < 0
                         ? 'L' : 'R';
    adjustPath(root, inserted);
  }
  else if((ancestor.balanceFactor == 'L' &&
           o.compareTo(ancestor.element) > 0) ||
          (ancestor.balanceFactor == 'R' &&
           o.compareTo(ancestor.element) < 0)) {
    ancestor.balanceFactor = '=';
    adjustPath(ancestor, inserted);
  }
  else if( /* case 3 */ ) { ... }
  else if( /* case 4 */ ) { ... }
  else if( /* case 5 */ ) { ... }
  else {   /* case 6 */     ... }
}
```

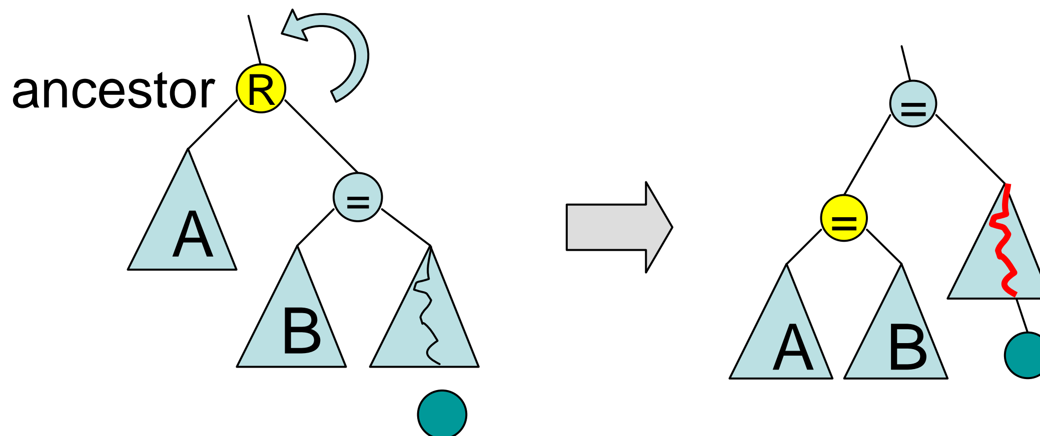




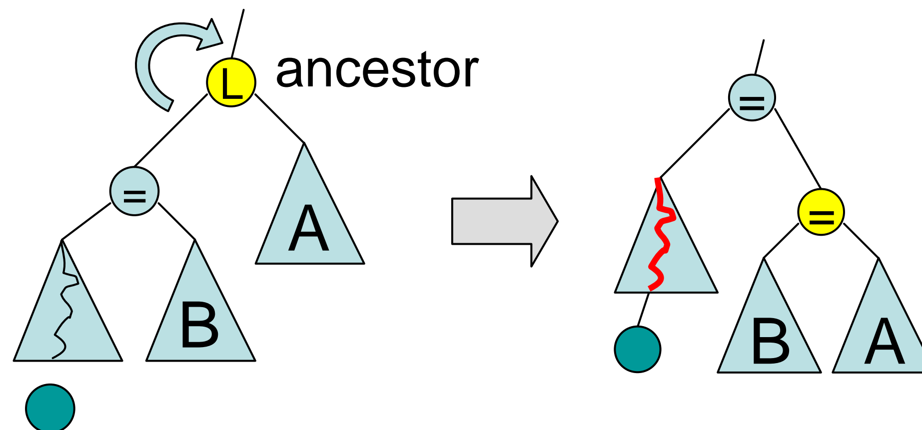
```java
private void adjustPath(Entry to, Entry inserted) {
    Comparable o = (Comparable) (inserted.element);
    Entry temp = inserted.parent;
    while (temp != to) {
      if (o.compareTo(temp.element) < 0) {
        temp.balanceFactor = 'L';
      } else {
        temp.balanceFactor = 'R';
      }
      temp = temp.parent;
    }
  }
```



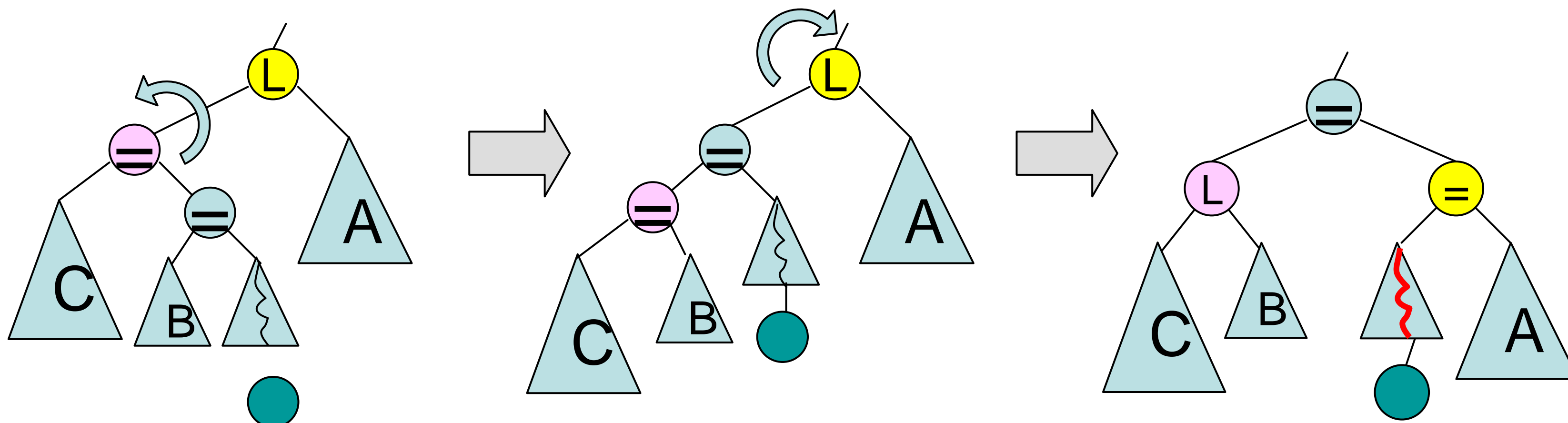
```
void fixAfterInsertion(Entry ancestor, Entry inserted) {
  Comparable o = (Comparable) (inserted.element);
  if (      /* case 1 */ ) { ... }
  else if( /* case 2 */ ) { ... }
  else if( ancestor.balanceFactor == 'R' &&
           o.compareTo(ancestor.right.element) > 0) {
    ancestor.balanceFactor = '=';
    rotateLeft(ancestor);
    adjustPath(ancestor.parent, inserted);
  }
  else if( /* case 4 */ ) { ... }
  else if( /* case 5 */ ) { ... }
  else {   /* case 6 */       ... }
}
```



```
void fixAfterInsertion(Entry ancestor, Entry inserted) {
  Comparable o = (Comparable) (inserted.element);
  if (      /* case 1 */ ) { ... }
  else if( /* case 2 */ ) { ... }
  else if( /* case 3 */ ) { ... }
  else if( ancestor.balanceFactor == 'L' &&
           o.compareTo(ancestor.left.element) < 0) {
    ancestor.balanceFactor = '=';
    rotateRight(ancestor);
    adjustPath(ancestor.parent, inserted);
  }
  else if( /* case 5 */ ) { ... }
  else {   /* case 6 */      ... }
}
```

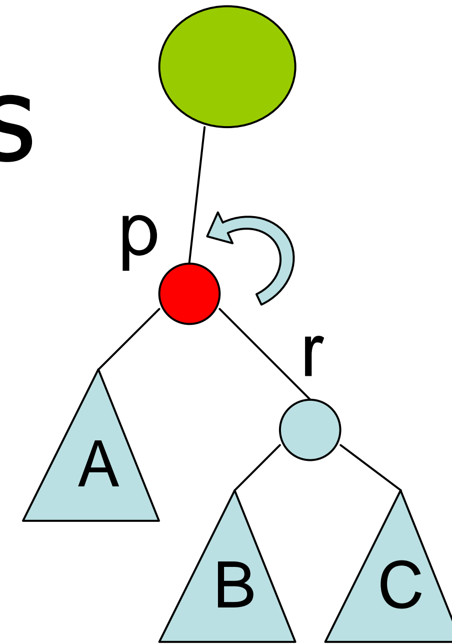


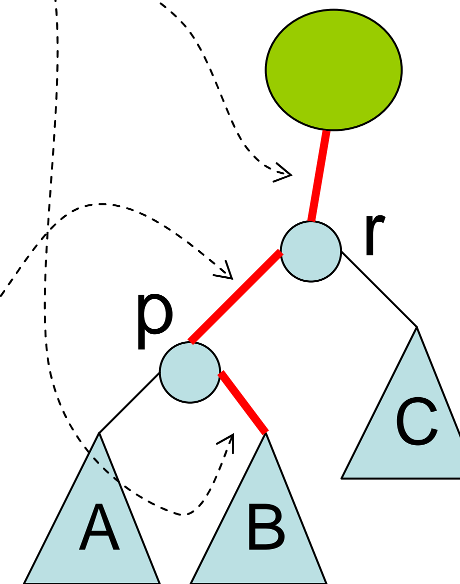
```
void fixAfterInsertion(Entry ancestor, Entry inserted) {
  Comparable o = (Comparable) (inserted.element);
  if (      /* case 1 */ ) { ... }
  else if( /* case 2 */ ) { ... }
  else if( /* case 3 */ ) { ... }
  else if( /* case 4 */ ) { ... }
  else if( ancestor.balanceFactor == 'L' &&
           o.compareTo(ancestor.left.element) > 0) {
    rotateLeft(ancestor.left);
    rotateRight(ancestor);
    adjustLeftRight(ancestor, inserted);
  }
  else {   /* case 6 */      ... }
}
```



Left Rotations

```
private void rotateLeft(Entry p) {
  Entry r = p.right;
  p.right = r.left;
  if (r.left != null) {
    r.left.parent = p;
  }
  r.parent = p.parent;
  if (p.parent == null) {
    root = r;
  } else if (p.parent.left == p) {
    p.parent.left = r;
  } else {
    p.parent.right = r;
  }
  r.left = p;
  p.parent = r;
}
```

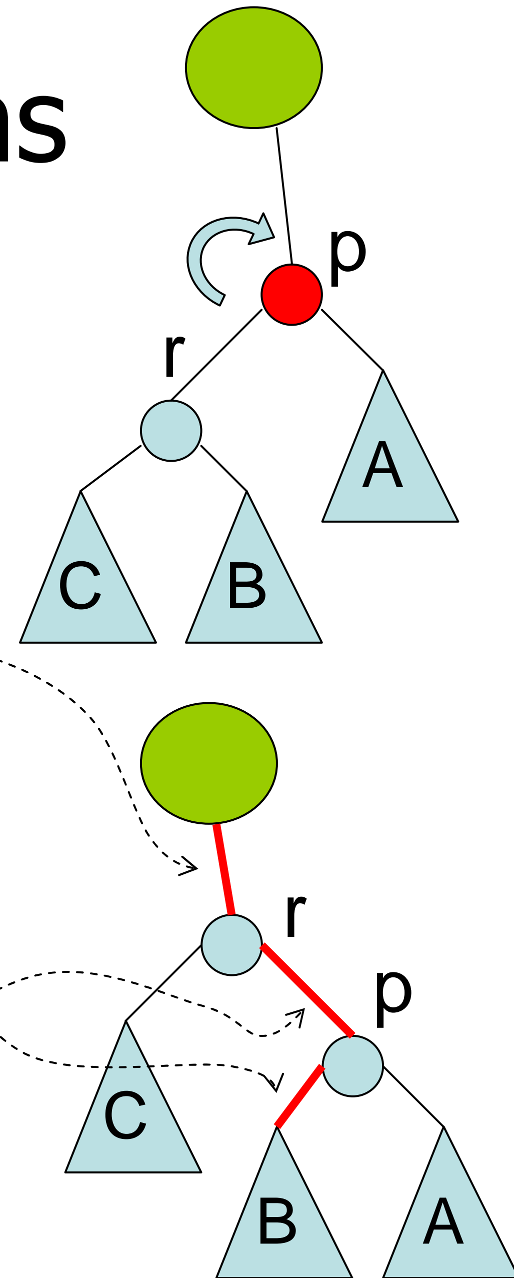




หลังหมุนก็ยังเป็น search tree

Right Rotations

```java
private void rotateRight(Entry p) {
  Entry r = p.left;
  p.left = r.right;
  if (r.right != null) {
    r.right.parent = p;
  }
  r.parent = p.parent;
  if (p.parent == null) {
    root = r;
  } else if (p.parent.left == p) {
    p.parent.left = r;
  } else {
    p.parent.right = r;
  }
  r.right = p;
  p.parent = r;
}
```

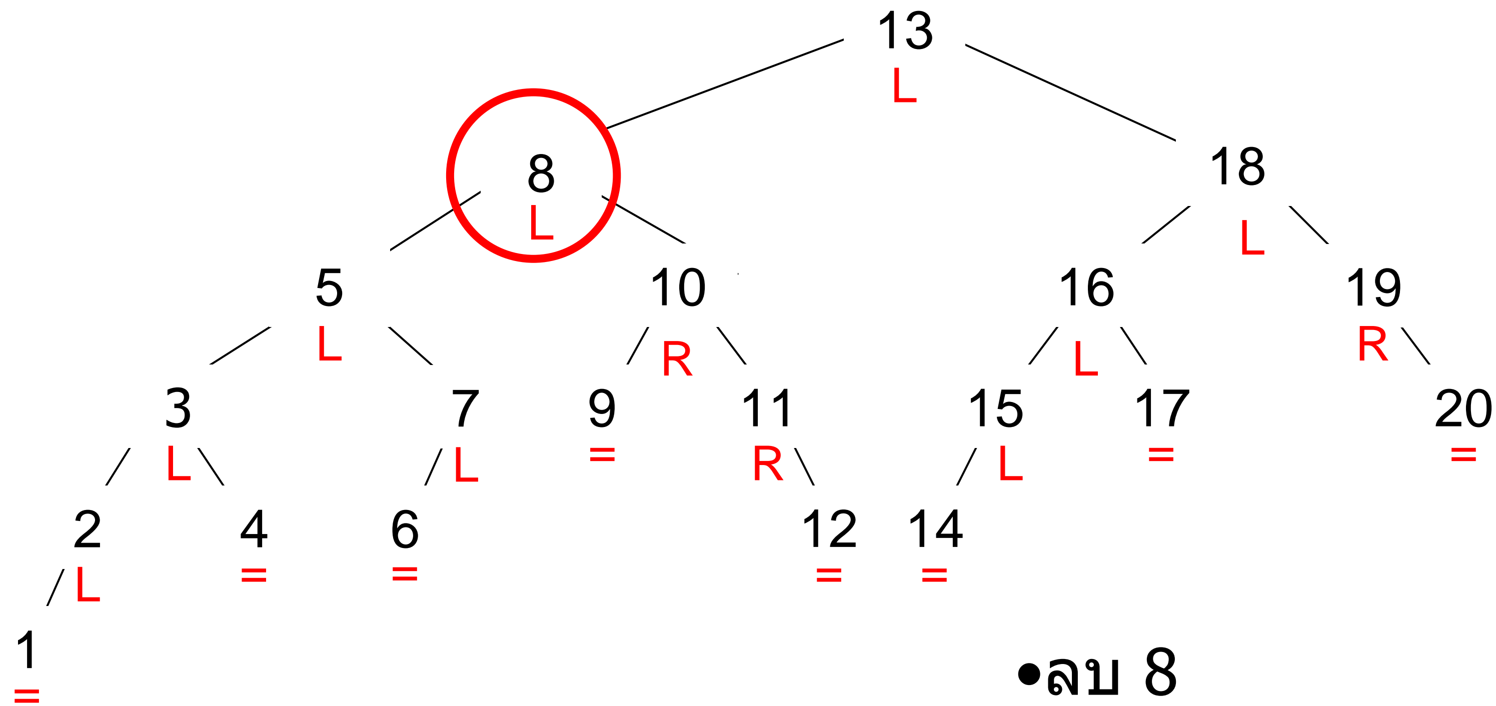


หลังหมุนก็ยังเป็น search tree

add ใช้เวลา O(log n)

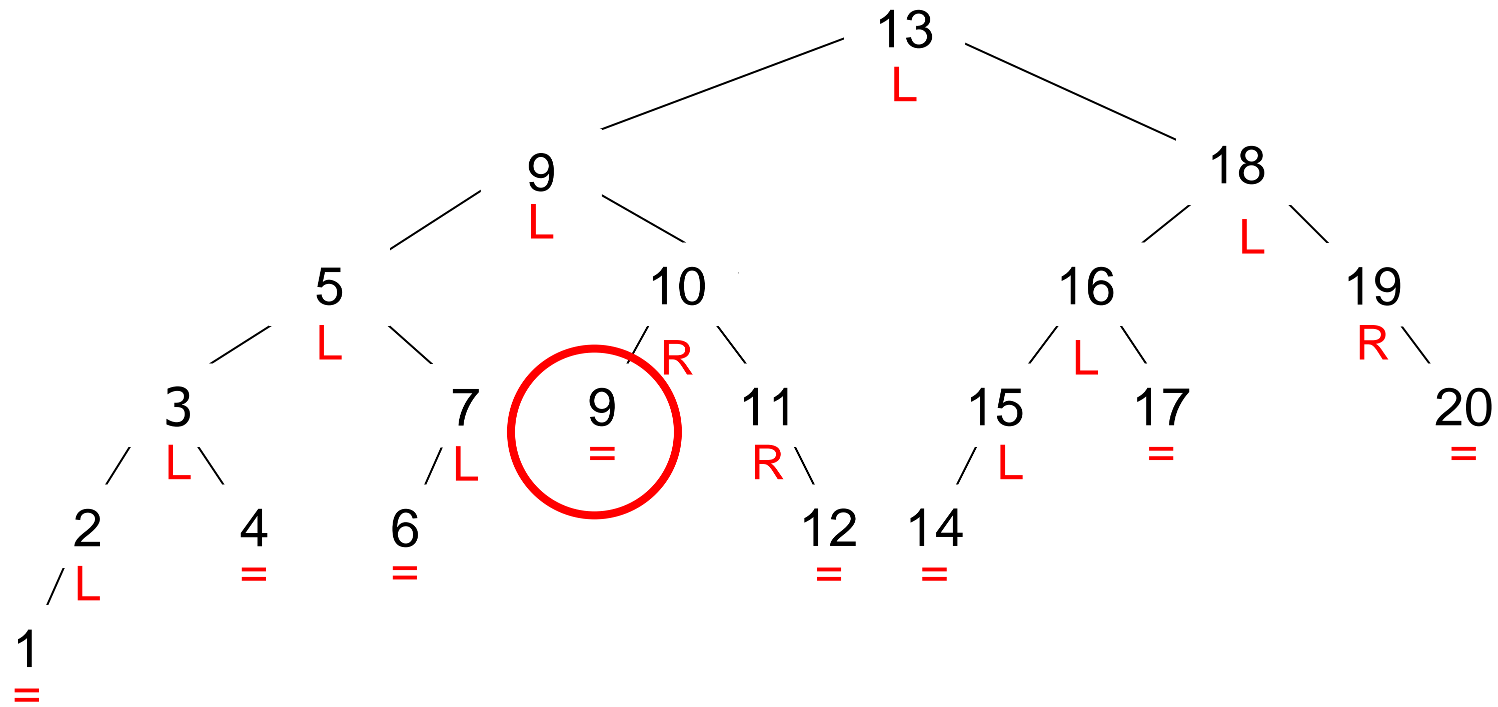
- เพิ่มตามปกติ วิ่งลงจากรากถึงใบ O(h)
- ต้อง adjustPath จาก node ใหม่ถึงราก O(h)
- อาจมีการหมุนอย่างมากสองครั้ง O(1)
- เวลารวม O(h)
- $h_{AVL} = O(\log n)$
- เวลาการ add เป็น O(log n)

remove

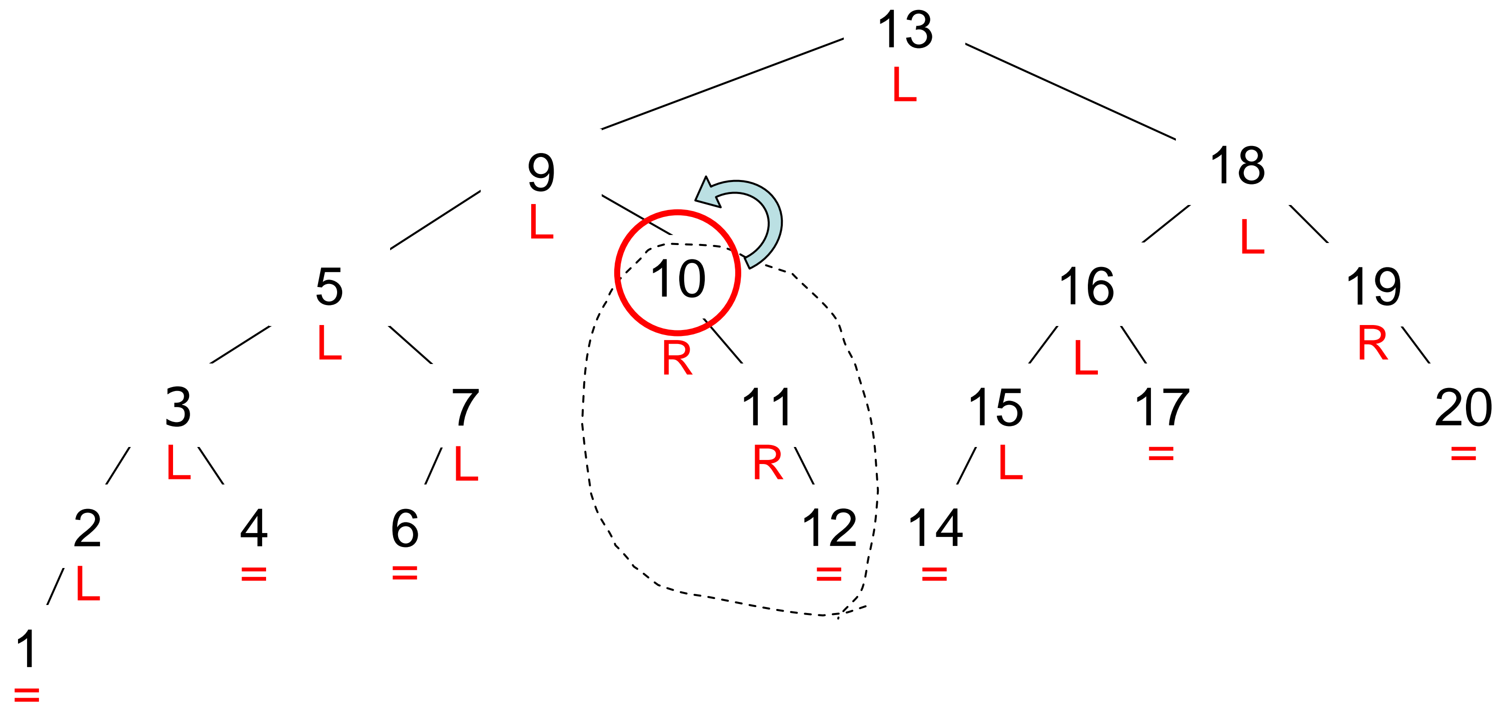


- ลบ 8
- หา 8
- หา successor(8) = 12
- copy 12 ไปที่ 8
- ลบ entry 12

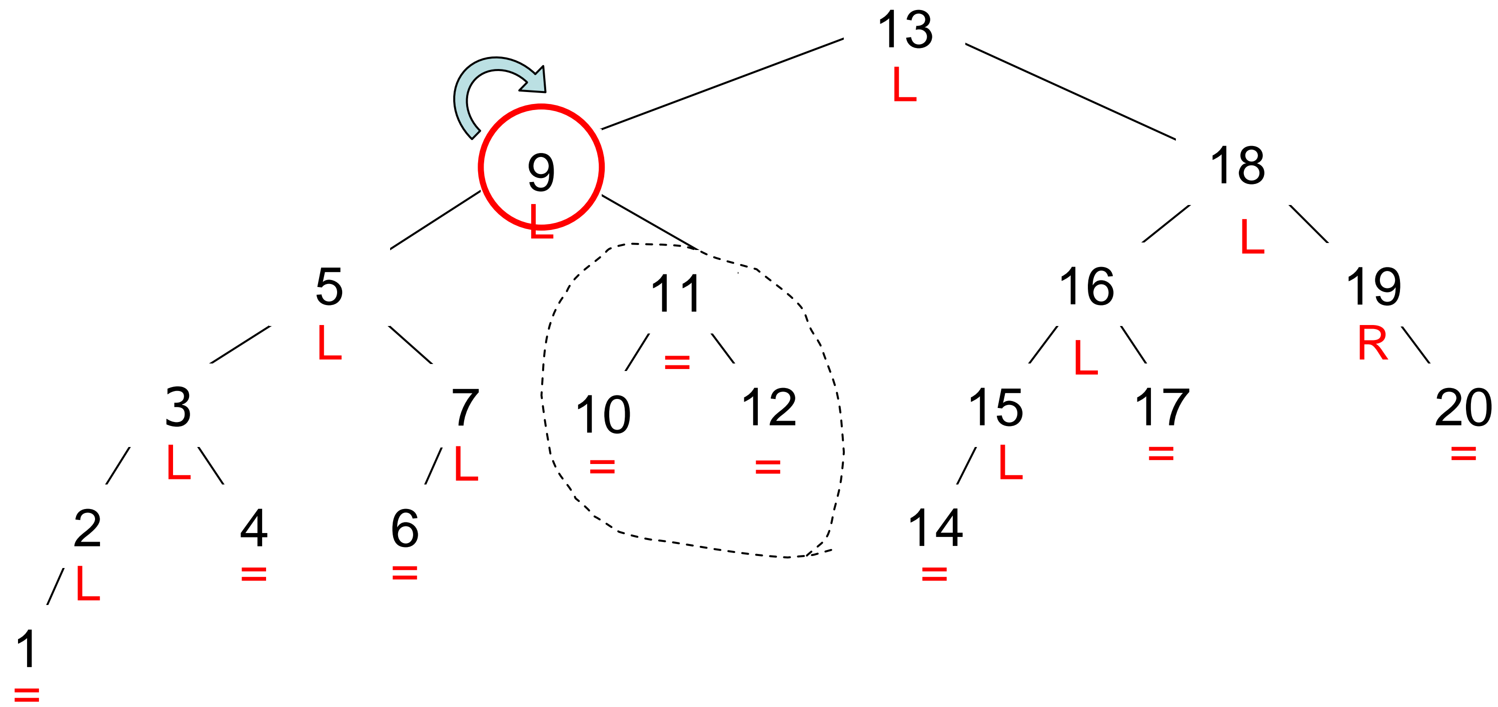
remove



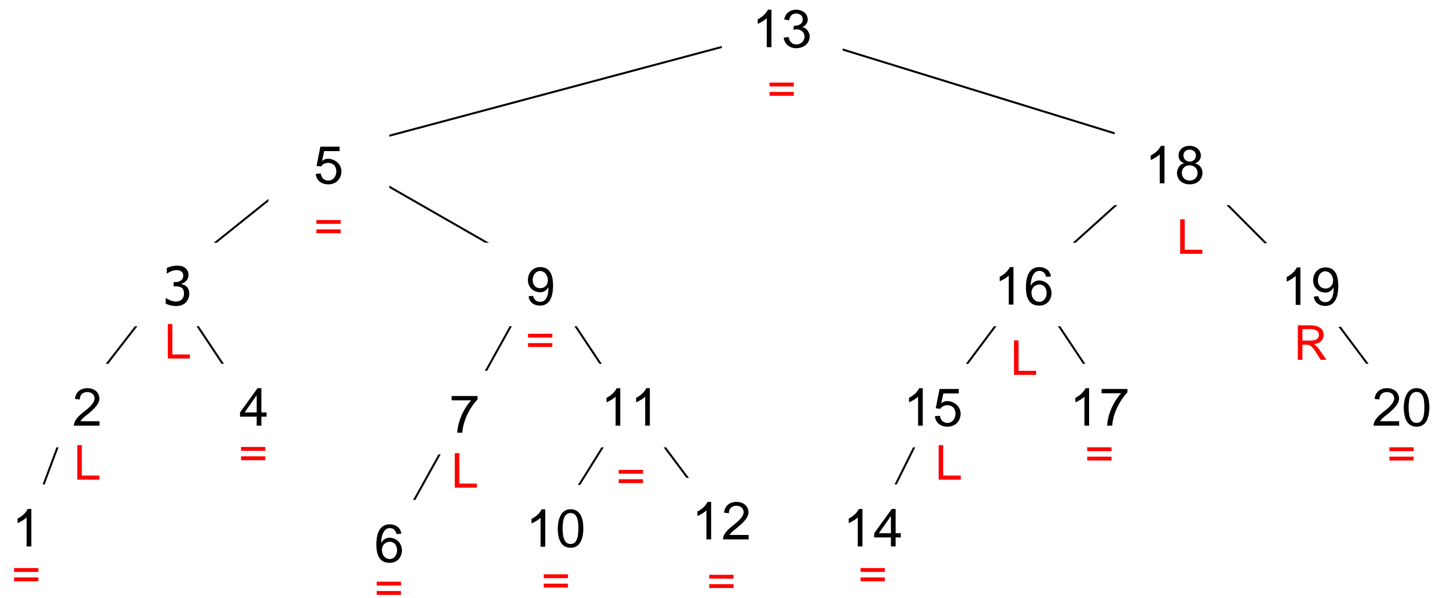
remove



remove



remove

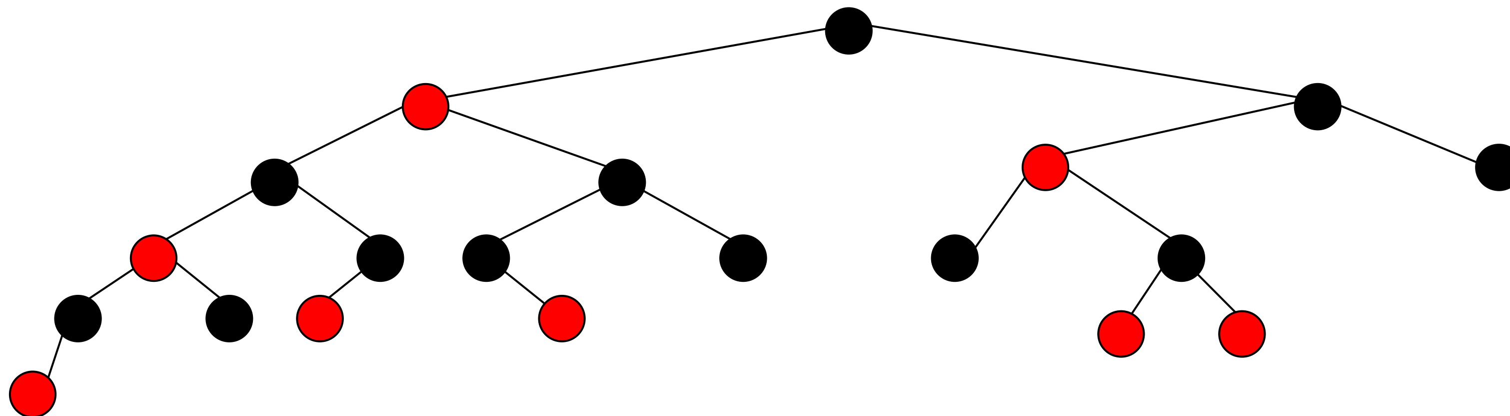


TreeSet + TreeMap

- Collections framework มี TreeSet และ TreeMap
- TreeMap implements Map เก็บ (key, value)
- TreeSet เก็บข้อมูลใน key ของ TreeMap (ไม่สนใจ value ของ TreeMap)
- TreeMap จัดเก็บ (key, value) ตาม nodes ต่างๆ ของต้นไม้ Red-Black
- Red-Black tree เป็น balanced binary search tree ($h_{RB} \leq 2\log_2 (n+1)$)

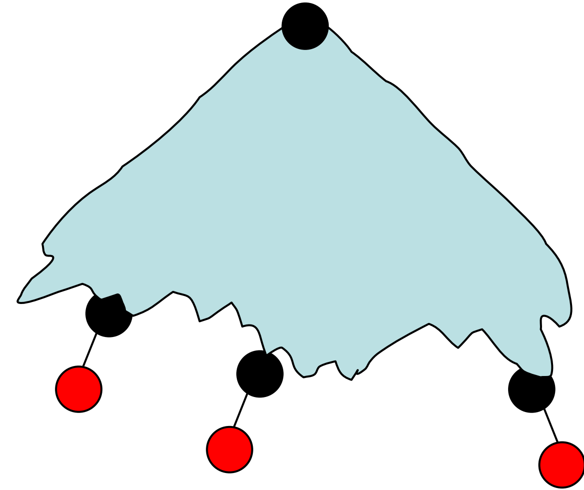
ลักษณะของ Red-Black Trees

- แต่ละปมมีสีแดงหรือสีดำกำกับ
- รากมีสีดำ
- ลูกๆ ของปมแดงต้องเป็นสีดำ
- จำนวนปมดำของทุกๆ วิถีที่เริ่มจากปมหนึ่งถึงปมล่างๆ ที่ไม่มีลูก หรือที่มีลูกเดียว ต้องเท่ากัน



ข้อสังเกต

- ปมภายในส่วนมากมีสองลูก
- ปมไหนมีลูกเดียว
  - ปมนั้นต้องมีปมดำ
  - ลูกของปมนั้นต้องเป็นใบแดง
- ถ้าปมใหม่ที่เพิ่งเพิ่มเป็นลูกของปมดำ ก็ให้เป็นปมใหม่เป็นสีแดง รับรองไม่ผิดกฎ



จำนวนปม

- ให้ bh(t) แทนจำนวนปมดำของวิถีที่เริ่มจาก t ไปยังปมที่ไม่มีลูกหรือที่มีลูกเดียว จะได้ว่า

$$n(t) \geq 2^{bh(t)} - 1$$

- พิสูจน์ : ใช้วิธีอุปนัยบน h(t)
  - basis : $h(t) = 0$, ได้ $n(t)=1$, $bh(t)=1$, $1 \geq 2^1 - 1$
    $h(t) = 1$ และ t มีลูกเดียว, ได้ $n(t) = 2$,
    $bh(t) = 1$, $2 \geq 2^1 - 1$

จำนวนปม

- พิสูจน์ :
  - inductive : ให้ t มีสองลูก u และ v
    - เมื่อ t เป็มปมแดง : $bh(u) = bh(v) = bh(t)$
    - เมื่อ t เป็นปมดำ : $bh(u) = bh(v) = bh(t) - 1$
    - $h(u) < h(t)$ และ $h(v) < h(t)$ เพราะ u และ v เป็นลูก
    - $n(t) = n(u) + n(v) + 1$
    - จาก inductive hypothesis
      - $n(u) \geq 2^{bh(u)} - 1 \geq (2^{bh(t) - 1}) - 1$
      - $n(v) \geq 2^{bh(v)} - 1 \geq (2^{bh(t) - 1}) - 1$
    - $n(t) \geq (2^{bh(t) - 1}) - 1 + (2^{bh(t) - 1}) - 1 + 1 = 2^{bh(t)} - 1$

ความสูงของ Red-Black Trees

- จากกฎ : ลูกของแดงต้องดำหมดทั้งสองลูก
  - วิถีจาก t ถึงใบใดๆ
    - อาจดำหมด
    - ถ้ามีแดงคั่น หนึ่งแดงต้องมีหนึ่งดำ
  - ดังนั้น $bh(t) \ge h(t)/2$
- สรุป $n(t) \ge 2^{h(t)/2} - 1$
- ใส่ $\log_2$ ได้ $h(t) \le 2 \log_2 (n(t)+1)$

Red-black tree ที่มี n ปม สูงไม่เกิน $2 \log_2 (n+1)$

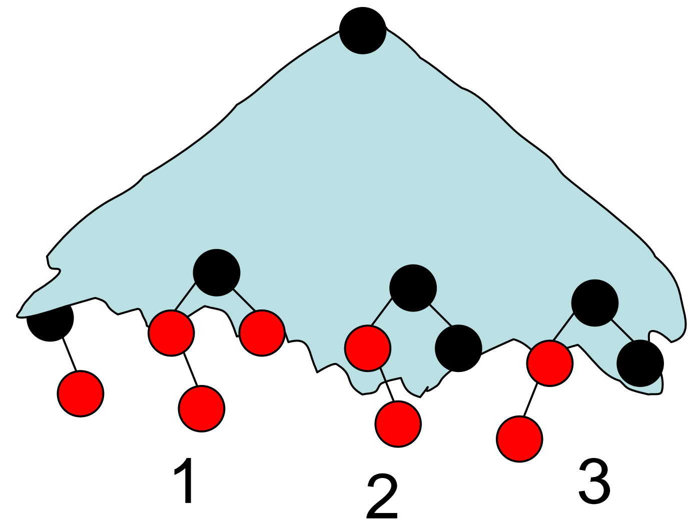
TreeMap

```
public TreeMap ... {
  private Entry root = null;
  private int size = 0, modCount = 0;

  static class Entry...{
    Object key, value;
    Entry left = null, right = null;
    Entry parent;
    boolean color = BLACK;
    Entry(Object key, Object value, Entry parent) {
      this.key = key;
      this.value = value;
      this.parent = parent;
    }
    ...
  }
}
```

```
public Object put(Object key, Object value) {
  Entry t = root;
  if (t == null) {
    incrementSize(); // size++; modCount++;
    root = new Entry(key, value, null);
    return null;
  }
  while (true) {
    int c = compare(key, t.key); //key.compareTo(t.key)
    if (c == 0) { return t.setValue(value); }
    else if (c < 0) {
      if (t.left != null) { t = t.left; }
      else {
        incrementSize();
        t.left = new Entry(key, value, t);
        fixAfterInsertion(t.left);
        return null;
      }
    } else { /* add in the right subtree */ }
  }
}
```

fixAfterInsertion(x)

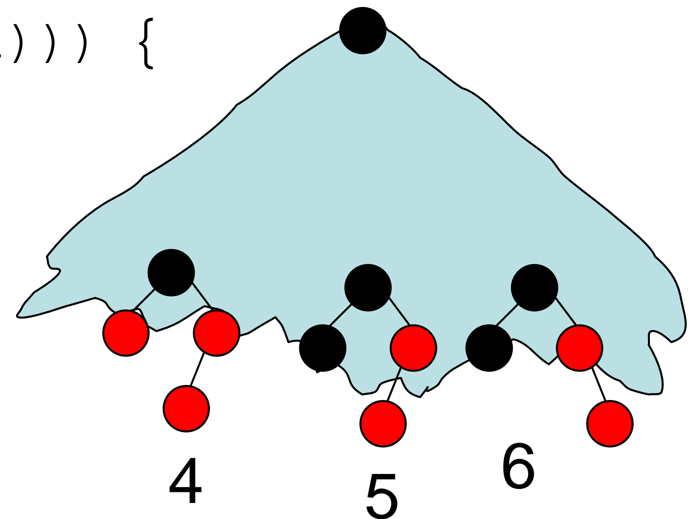
```
private void fixAfterInsertion(Entry x) {
  x.color = RED;
  while(x != null && x!=root && x.parent.color== RED) {
    if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
      Entry y = rightOf(parentOf(parentOf(x)));
      if (colorOf(y) == RED) {
        /* case 1 */
      } else {
        if (x == rightOf(parentOf(x))) {
          /* case 2 */
        }
        /* case 3 */
      }
    } else { /* case 4, 5, 6 */ }
  }
  root.color = BLACK;
}
```



```java
private static final boolean RED   = false;
private static final boolean BLACK = true;
private static boolean colorOf(Entry p) {
  return (p == null ? BLACK : p.color);
}

private static Entry  parentOf(Entry p) {
  return (p == null ? null: p.parent);
}

private static void setColor(Entry p, boolean c) {
  if (p != null)  p.color = c;
}

private static Entry  leftOf(Entry p) {
  return (p == null)? null: p.left;
}

private static Entry  rightOf(Entry p) {
  return (p == null)? null: p.right;
}
```

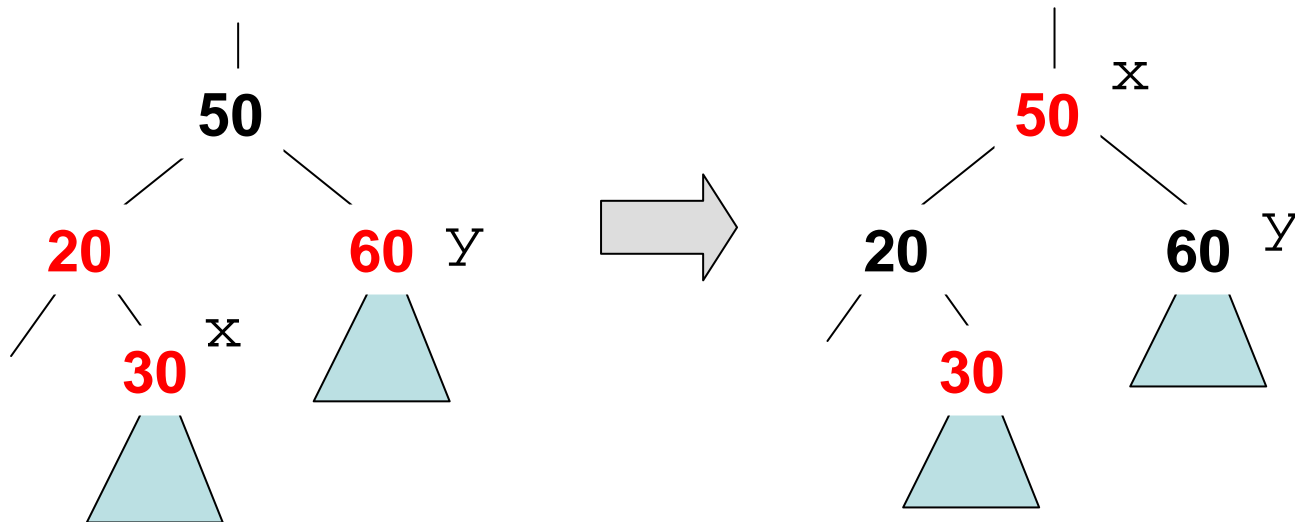
fixAfterInsertion(x)

```
private void fixAfterInsertion(Entry x) {
  x.color = RED;
  while(x != null && x!=root && x.parent.color== RED) {
    if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
      /* case 1, 2, 3 */
    } else {
      Entry y = leftOf(parentOf(parentOf(x)));
      if (colorOf(y) == RED) {
        /* case 4 */
      } else {
        if (x == leftOf(parentOf(x))) {
          /* case 5 */
        }
        /* case 6 */
      }
    }
  }
  root.color = BLACK;
}
```



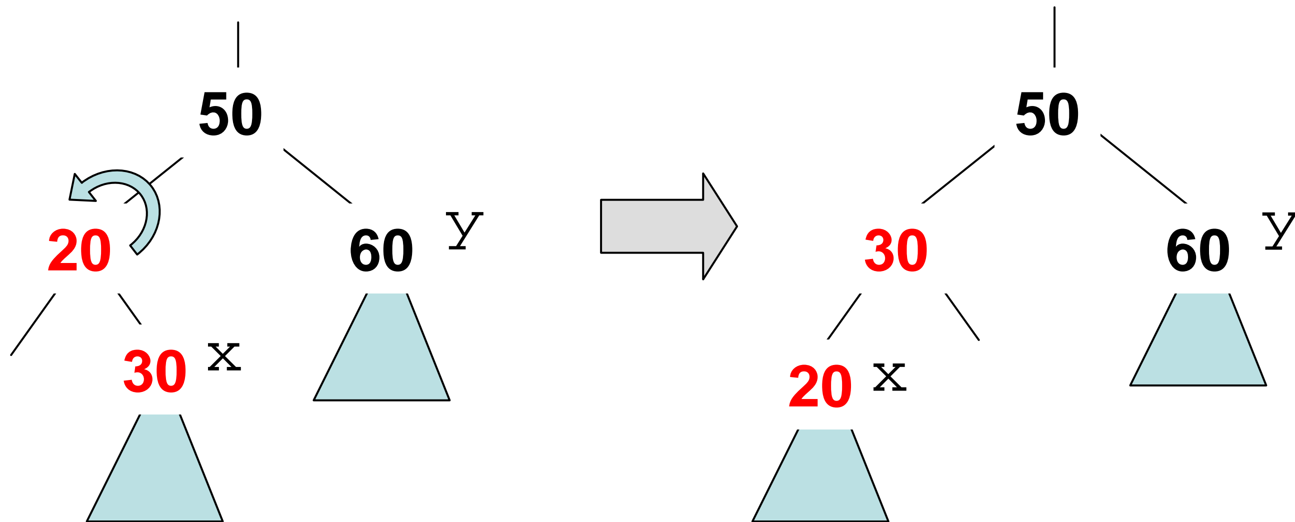
fixAfterInsertion(x) : กรณีที่ 1

```
if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
  Entry y = rightOf(parentOf(parentOf(x)));
  if (colorOf(y) == RED) {
    setColor(parentOf(x), BLACK);
    setColor(y, BLACK);
    setColor(parentOf(parentOf(x)), RED);
    x = parentOf(parentOf(x));
  }
```



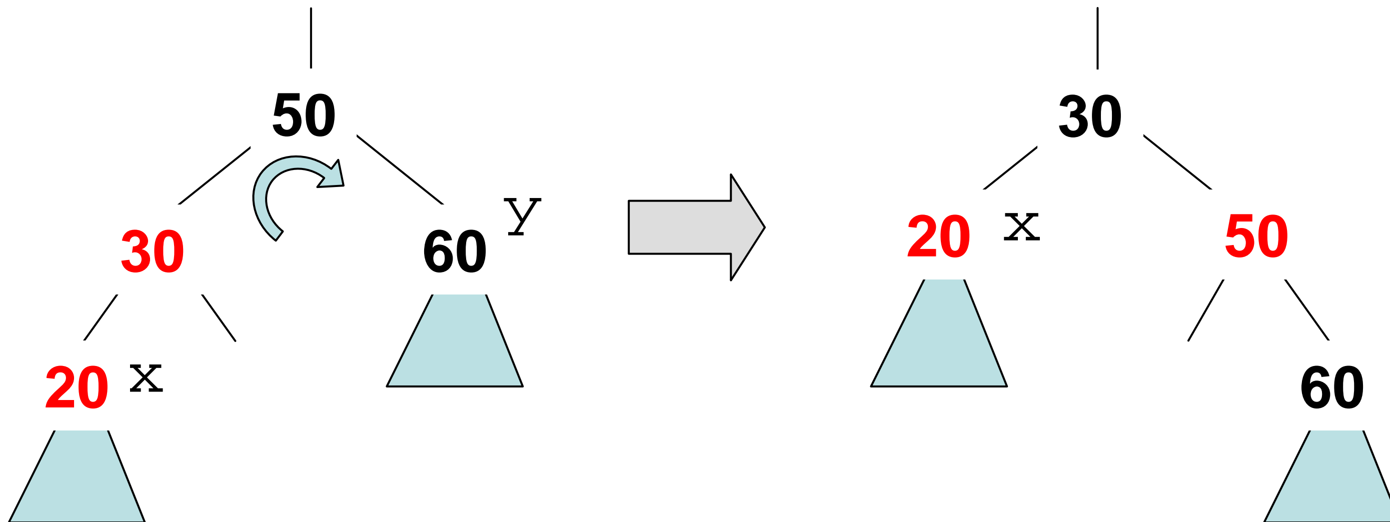
fixAfterInsertion(x) : กรณีที่ 2

```
if (colorOf(y) == RED) {
  /* case 1 */
} else {
  if (x == rightOf(parentOf(x))) {
    x = parentOf(x);
    rotateLeft(x);
  }
```

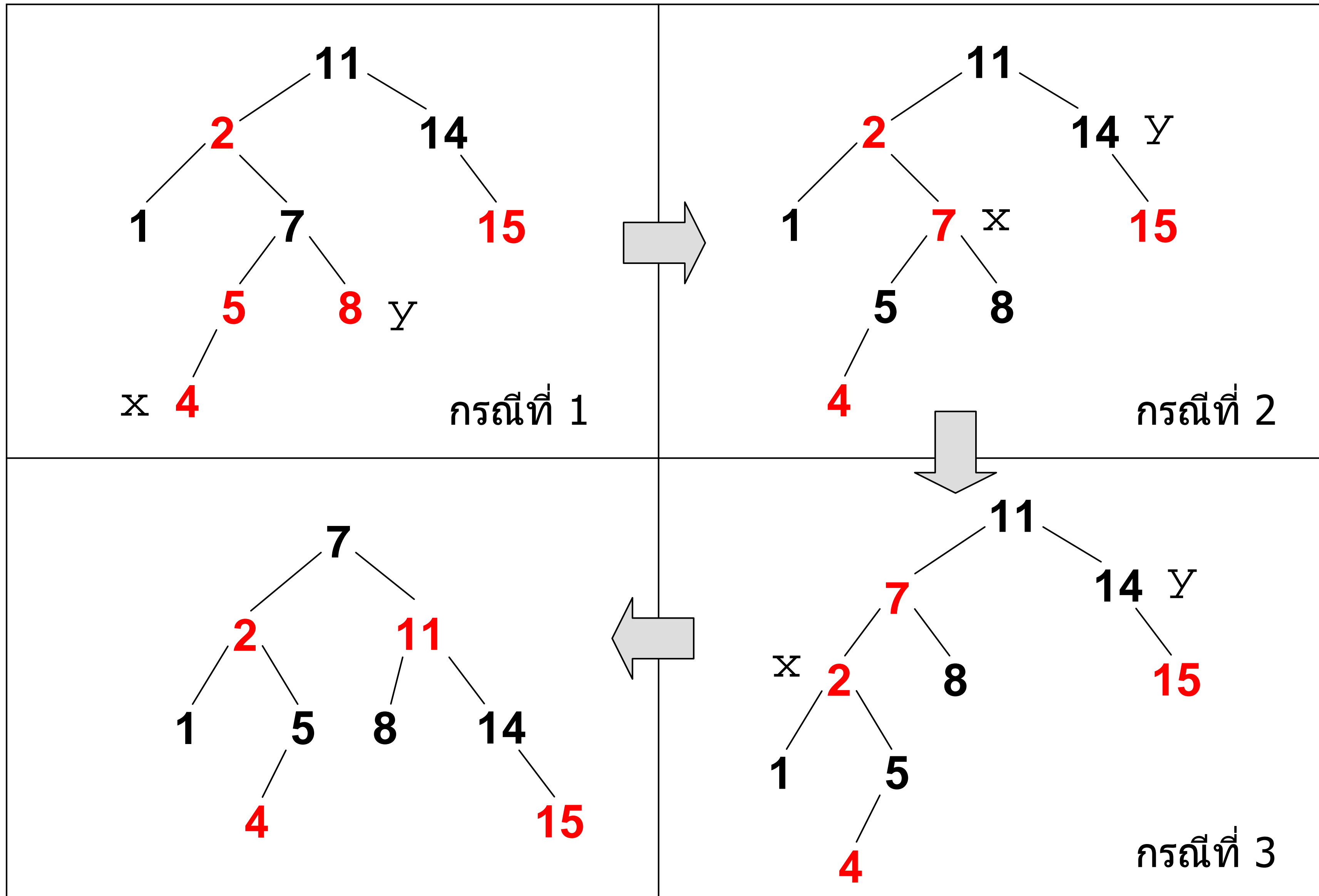


fixAfterInsertion(x) : กรณีที่ 3

```
if (colorOf(y) == RED) {
  /* case 1 */
} else {
  /* case 2 */
  setColor(parentOf(x), BLACK);
  setColor(parentOf(parentOf(x)), RED);
  if (parentOf(parentOf(x)) != null) {
    rotateRight(parentOf(parentOf(x)));
  }
```



ตัวอย่าง



หัวข้อที่ครอบคลุม

- AVL :
  - บทที่ 8 หน้าที่ 323 – 348
- Red-black :
  - บทที่ 8 หน้าที่ 348 – 355
  - บทที่ 9 หน้าที่ 361 – 375
- อ่านเพิ่ม : หัวข้อ 10.2 10.3 และ 10.4