บทที่ 7

โครงสร้างต้นไม้แบบเอวีแอล

ในบทนี้เราจะมาดูโครงสร้างต้นไม้แบบพิเศษ ที่เรียกว่า เอวีแอล (AVL)

ลักษณะทั่วไปของโครงสร้างต้นไม้เอวีแอล

ต้นไม้ AVL ก็คือต้นไม้ค้นหาแบบทวิภาคแบบหนึ่งนั้นเอง แต่ว่ามีลักษณะพิเศษแทรกเข้ามาคือ สำหรับแต่ละโนดนั้น

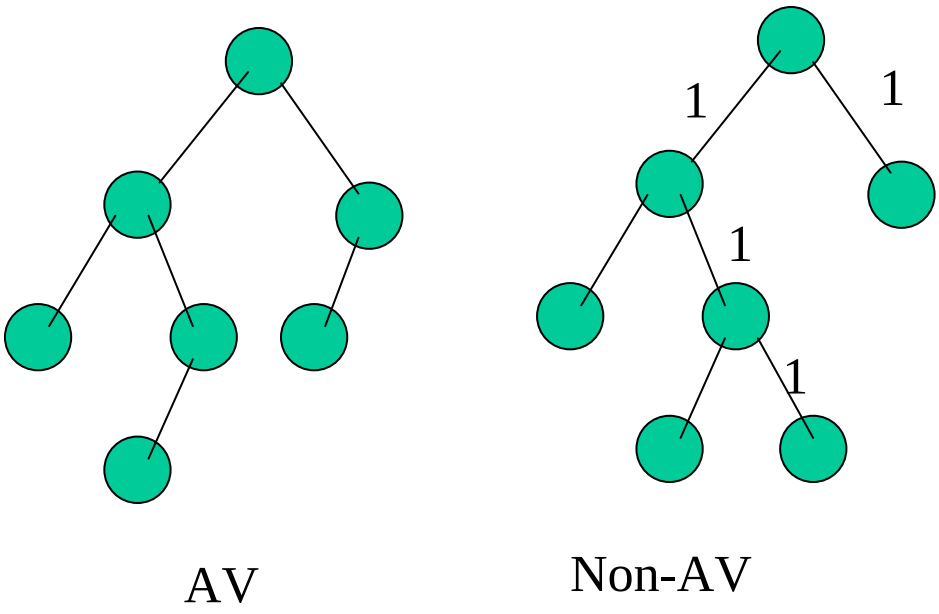
- ความสูงของ left กับ right subtree ต่างกันได้อย่างมากแค่ 1
- ถ้า tree เป็น empty tree เราให้ height เป็น -1

ความสูงของต้นไม้ชนิดนี้จะเป็น log N ซึ่งมีค่าจริงคือ 1.44log(N+2)-.328 โดย N คือจำนวนสมาชิกในต้นไม้นี้ ที่มีการจัดการบังคับเรื่องความต่างของขนาดต้นไม้ย่อย ก็เพื่อที่จะให้แน่ใจว่าต้นไม้อยู่ในสภาพที่หาของได้ง่ายที่สุดโดยรวม ไม่กลายเป็น skewed tree (การทำฟังก์ชั่นต่างๆ บนต้นไม้ AVL นี้ จะต้องปรับตัวต้นไม้ให้อยู่ในสภาพตามกฎที่ให้ด้านบนนี้เสมอ)

เนื่องจากแต่ละข้างนั้นสูงกันได้ต่างกันแค่หนึ่งหน่วย ดังนั้นจำนวนโนดที่น้อยที่สุดของต้นไม้ที่มีความสูง h, s(h) จะมีค่าเท่ากับ s(h-1)+s(h-2)+1 ซึ่ง

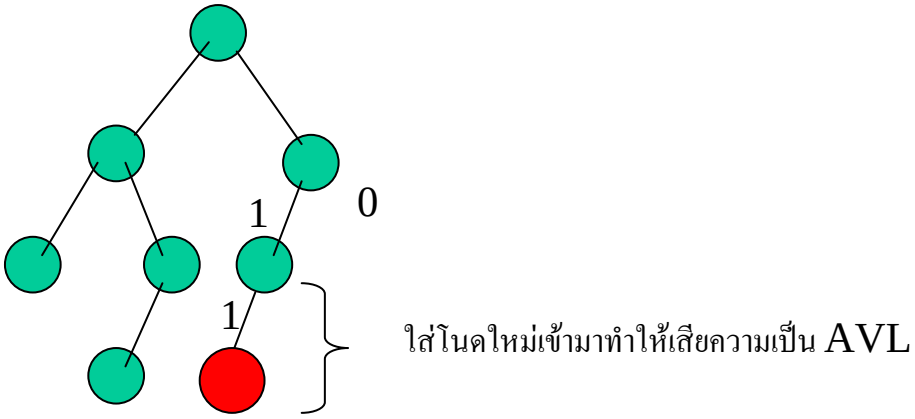
- เมื่อ h=0, s(h) =1 และ
- เมื่อ h=1, s(h) =2

ซึ่งจะเห็นว่าคล้ายตัวเลขของ Fibonacci ตัวอย่างต้นไม้ที่เป็น AVL และไม่เป็น AVL นั้นอยู่ในรูปที่ 7.1 ซึ่งจากรูปขวาที่ไม่ใช่ต้นไม้ AVL จะเห็นได้ว่า เป็นเพราะว่าเมื่อดูจากราก ข้างซ้ายสามารถนับกิ่งลงไปได้มากที่สุดสามกิ่ง ในขณะที่ข้างขวาสามารถนับกิ่งลงไปได้เพียงแค่หนึ่งกิ่งเท่านั้น จึงมีความสูงต่างกันอยู่สองหน่วย ไม่ใช่ต้นไม้แบบ AVL



รูป 7.1 ต้นไม้ที่เป็น AVL และไม่เป็น

ฟังก์ชั่นต่างๆที่กระทำกับต้นไม้ AVL มีความเร็ว O(log(N)) หมด ยกเว้นการ insert กับ remove ซึ่งอาจทำให้เสียความเป็น AVL ไป ตัวอย่างการเสียความเป็น AVL ดูได้จากรูปที่ 7.2 ซึ่งจะเห็นว่า เมื่อมีการเติมของเข้าไปในต้นไม้ อาจทำให้ความสูงของต้นไม้ย่อยเปลี่ยนไปจนขัดกับกฎของ AVL ได้



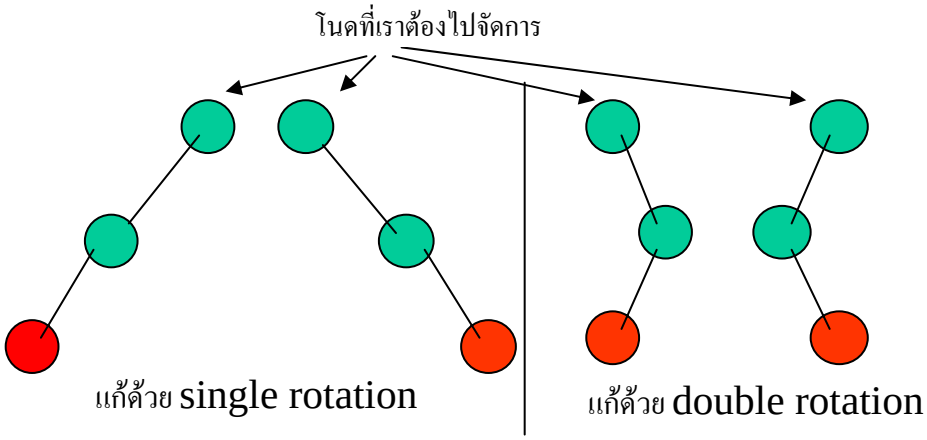
รูป 7.2 ต้นไม้ที่เสียความเป็น AVL

ซึ่งประเด็นสำคัญของการโปรแกรมต้นไม้ชนิดนี้ คือการจัดให้ความเป็น AVL คงอยู่ตลอดนั่นเอง

การแก้ความไม่เป็น AVL

เราใช้การหมุน (rotate) ซึ่งเราจะได้พูดถึงการ rotate จากการเติมของเข้ามาในต้นไม้ โดยเรามีหลักว่า

- โนดที่อยู่บนทางจากจุดที่ใส่ถึง root เท่านั้นที่มีความสูงเปลี่ยน
- ดังนั้นถ้าเราไล่ดูจากจุดที่ insert ขึ้นไป ก็จะพบโนดที่เสียความเป็น AVL ที่อยู่ลึกสุด เราจะแก้ที่โนดนี้ โนดนี้มีได้สี่แบบเท่านั้น ดังแสดงในรูป 7.3

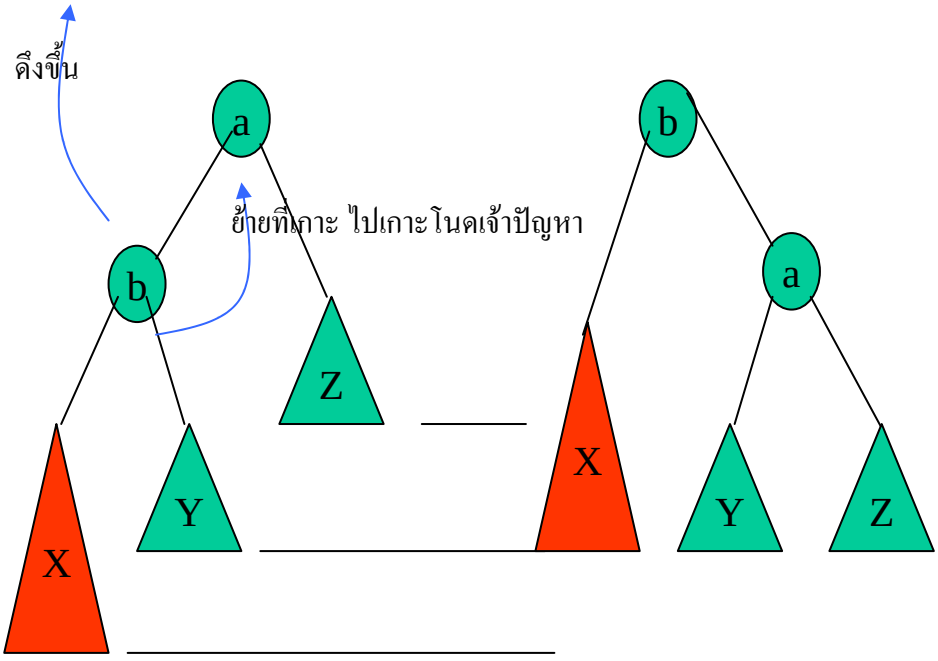


รูป 7.3 แสดงชนิดของโนด ที่เราต้องไปจัดการ เมื่อการใส่ของทำให้เสียความเป็น AVL

ซึ่งเราสามารถแยกวิธีแก้ได้สองแบบคือ single rotation และ double rotation โดยขึ้นอยู่กับรูปร่างของต้นไม้ย่อย ถ้าในการไปถึงโนดที่เติมเข้ามาใหม่เราต้องลงจากโนดที่เสียความเป็น AVL มาทางซ้ายสองครั้ง หรือขวาสองครั้ง จะต้องใช้ single rotation แก้ แต่ถ้าลงมาซ้ายทีขวาที หรือขวาทีซ้ายที ก็จะต้องใช้ double rotation แก้

การแก้ด้วย single rotation

มีวิธีการดังรูปที่ 7.4 โดยในรูปนั้นเป็นต้นไม้ที่เสียความเป็น AVL จากการเติมของลงในต้นไม้ย่อย X ทำให้โนดที่มี a เสียความเป็น AVL

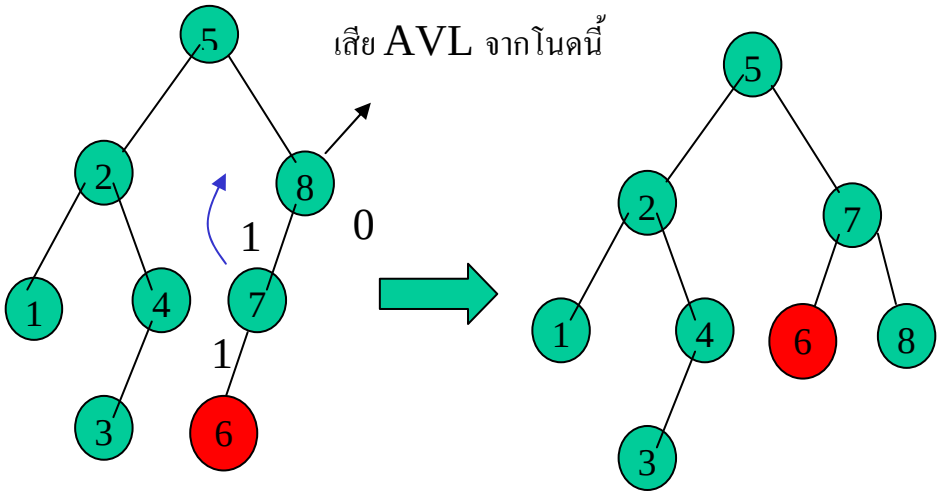


รูป 7.4 การทำ single rotation

จากโนดที่เสียความเป็น AVL ถ้าเรานับลงมาในทิศทางของค่าที่เติมเข้าไปในต้นไม้ จะเห็นว่ามาทางซ้ายสองที ดังนั้นจะต้องแก้ด้วย single rotation ตามขั้นตอนในรูป

ตัวอย่างที่ 7-1

สมมุติว่ามีการเติมเลข 6 ลงไปในต้นไม้ดังรูปที่ 7.5 จะทำให้โนดที่มีเลข 8 เสียความเป็น AVL ดังนั้นทางแก้ก็คือทำ single rotation เอา 8 ลงมาแล้วเอา 7 ขึ้นไป จะสังเกตว่า ในตัวอย่างนี้ เมื่อนับจากโนดที่เสียความเป็น AVL ลงมายังโนดที่ใส่เข้าไป ก็จะลงมาทางซ้ายสองที

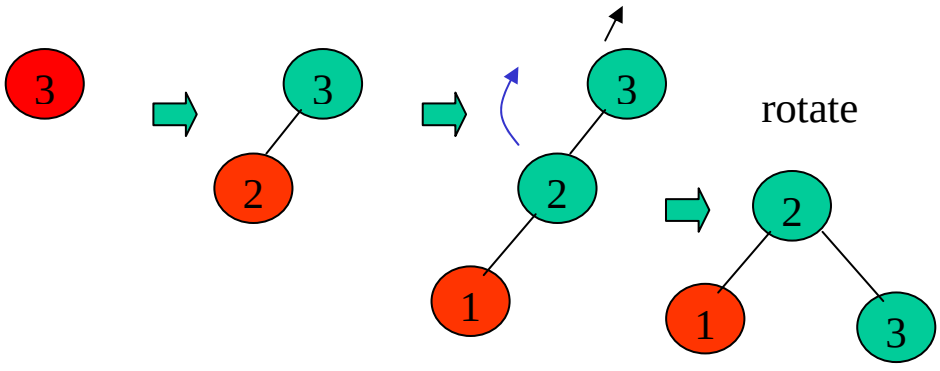


รูป 7.5 ปรับต้นไม้ที่เสียความเป็น AVL ด้วย single rotation

มาดูตัวอย่างต่อไปเพื่อความเข้าใจที่ชัดเจนนะ

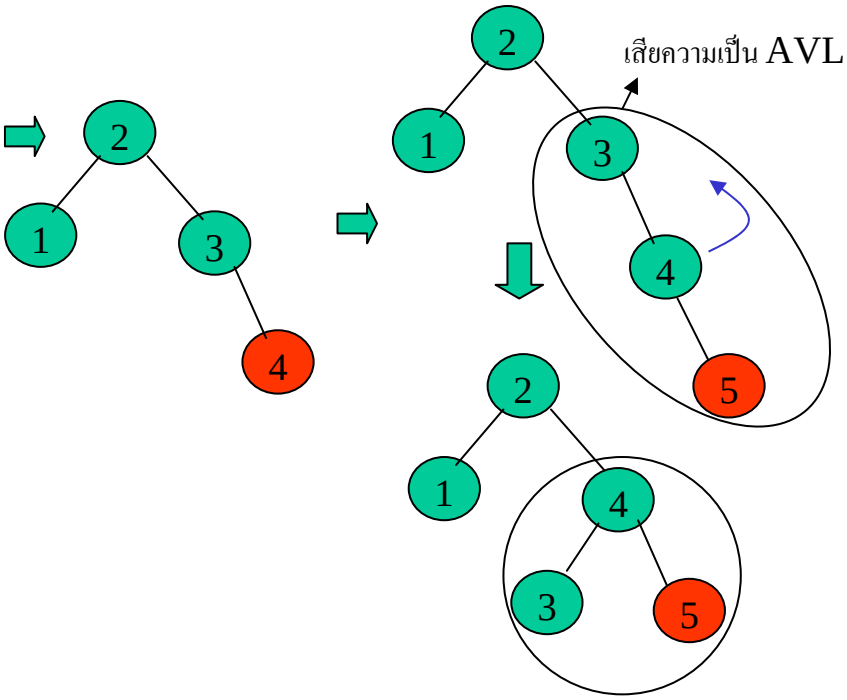
ตัวอย่างที่ 7-2

สมมุติว่ามีการเติมเลข 3,2,1 ตามลำดับ ลงในต้นไม้ AVL ที่ตอนแรกไม่มีอะไรเลย รูป 7.6 แสดงสถานะต่างๆของต้นไม้โดยมีขั้นตอนการ rotate แต่ละขั้นด้วย



รูป 7.6 การใส่ 3, 2, 1 ลงในต้นไม้ AVL

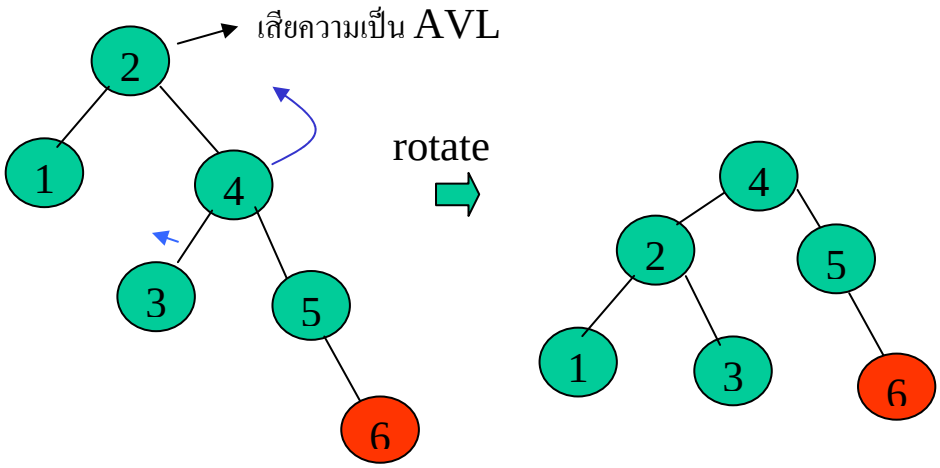
ต่อจากนั้น ใส่ 4 กับ 5 ลงไป ตอนใส่ 4 นั้นไม่มีอะไร แต่ตอนใส่ 5 เข้าไป ต้อง rotate ดังแสดงในรูป 7.7



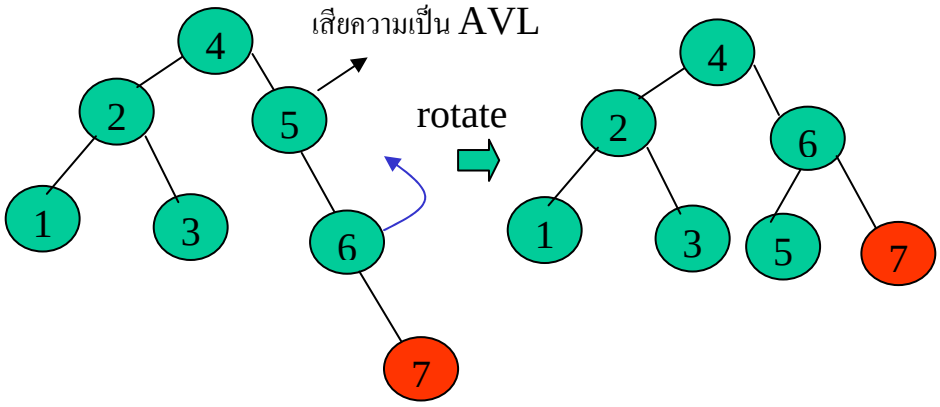
รูป 7.7 สภาพการณ์หลังจากใส่ 4 กับ 5 ลงไป

ต่อจากนั้น เราก็ทำต่อ โดยใส่ 6 ลงไป (รูป 7.8) คราวนี้จะสังเกตได้ว่าไม่เหมือนกรณีก่อนๆ เพราะโนดที่เสียความเป็นAVL นั้นอยู่สูงขึ้นไป แต่อย่างไรหลักการก็ยังเหมือนเดิม โปรดสังเกตว่าเลข 3 จะย้ายไปเกาะกับเลข 2

ต่อมา เราใส่เลข 7 ลงไปต่อ ครั้งนี้สถานการณ์กลับมาเป็นการทำ rotate อย่างง่าย (รูป 7.9) ซึ่งจะเห็นได้ว่าในการใส่ของลงต้นไม้แต่ละครั้งนั้น โนดที่เสียความเป็น AVL อาจจะเป็นโนดที่ใกล้หรือไกลจากจุดใส่ออกไปก็ได้ ดังนั้นต้องตรวจสอบจากจุดที่ใส่ของลงไปในต้นไม้เป็นสำคัญ



รูป 7.8 การ rotate ต้นไม้ หลังจาก 6 ทำให้เสียความเป็น AVL

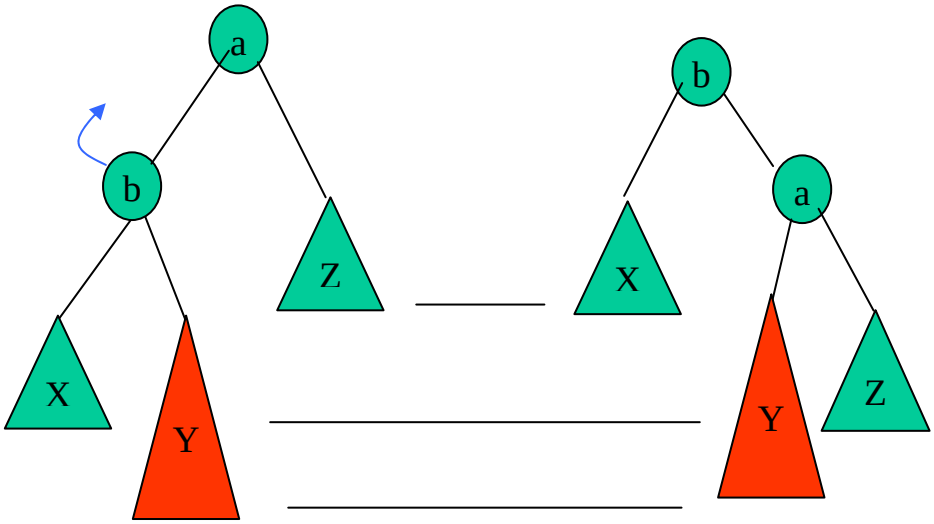


รูป 7.9 การ rotate หลังจากใส่เลข 7 ลงไป

การแก้ด้วย **double rotation**

เราจะใช้เมื่อ การ single rotation นั้นไม่ได้ผล รูป 7.10 แสดงกรณีที่ single rotation ไม่สามารถใช้แก้ปัญหาได้ เมื่อมีการใส่ของลงในต้นไม้ย่อย Y กรณีนี้จะเกิดขึ้นเมื่อทิศทางจากโนดที่เสียความเป็น AVL ไปยังโนดที่เราเติมเข้ามาใหม่นั้น ต้องลงมาซ้ายทีขวาที

จะสังเกตได้ว่า การ rotate นั้นก็ยังทำให้ต้นไม้ไม่เป็น AVL อยู่ดี a b Y แค่เปลี่ยนเป็น b a Y นั่นคือ ความสูงยังเท่าเดิม



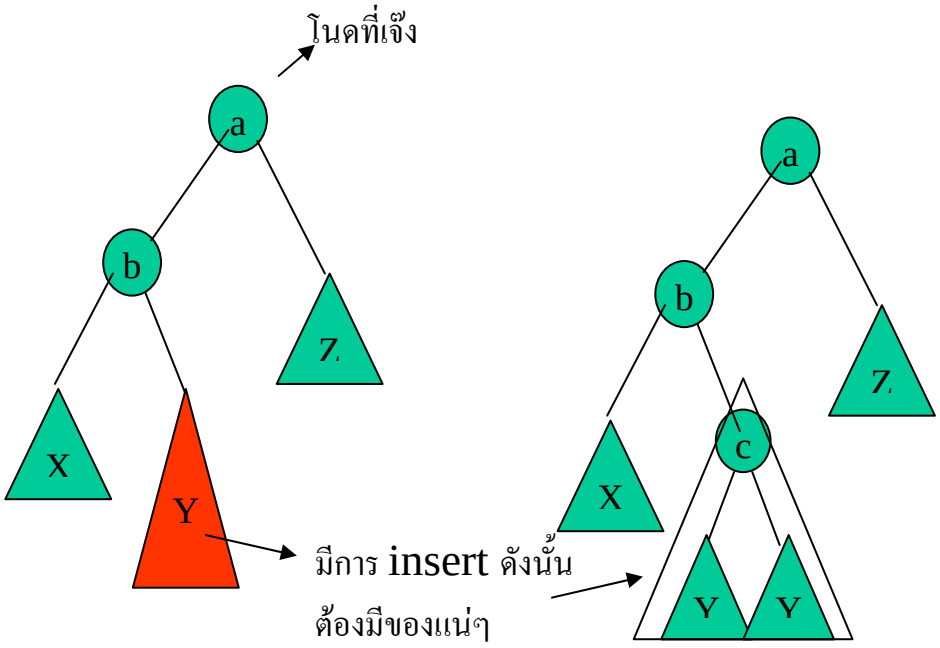
รูป 7.10 กรณีที่ single rotation ใช้แก้ปัญหาไม่ได้

ดังนั้นเราจึงต้องมาแก้ด้วย double rotation ซึ่งมีหลักการดังนี้ เราใช้ต้นไม้ในรูป 7.10 เป็นต้นแบบ ต้นไม้นี้มีการเติมของเข้าไปในต้นไม้ย่อย Y ดังนั้นตอนนี้ Y ต้องมีของแน่ๆ ขอเปลี่ยนรูปของต้นไม้ โดยเปลี่ยน Y เป็น โนด c ซึ่งมีต้นไม้ย่อย Y1 และ Y2 ขนาบข้าง (รูป 7.11)

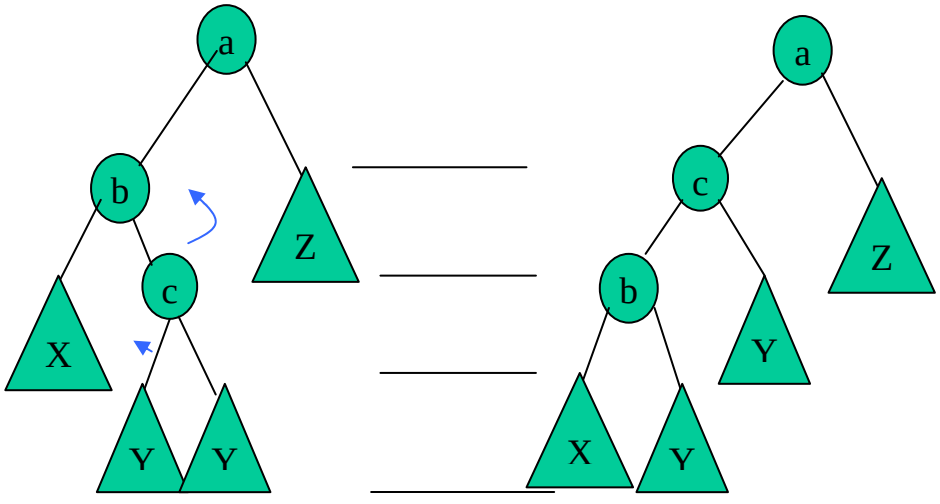
จากนั้นการ rotate ทำได้โดย rotate c ขึ้นไปแทน b และ rotate c ขึ้นไปแทน a นับเป็นการหมุนสองทีติดต่อกัน โปรดดูรูปที่ 7.12 และ 7.13 ตามลำดับ การ rotate ครั้งที่หนึ่ง และครั้งที่สองนั้นเป็น single rotation ทั้งคู่ ดังนั้น double rotation ก็คือการทำ single rotation สองครั้งนั่นเอง

แม้ว่าเราจะไม่รู้ว่าข้าง Y1 หรือ Y2 กันแน่ที่ทำให้ต้นไม้เสียความเป็น AVL แต่ก็ไม่เป็นไร เพราะเมื่อทำเสร็จระดับความสูงก็จะลงตัวเอง ภาพรวมของการหมุนต้นไม้แบบ double rotation นี้ อยู่ในรูป 7.14 โดยเราอาจคิดได้ว่าตัว c นั้น เป็นสไปเดอร์แมน ปล่อย Y1 กับ Y2 แล้วกระโดดขึ้นหิ้ว a และ b แทน ส่วน Y1 กับ Y2 นั้นก็ต้องหาที่ปลอดภัยเกาะข้างๆนั่นเอง

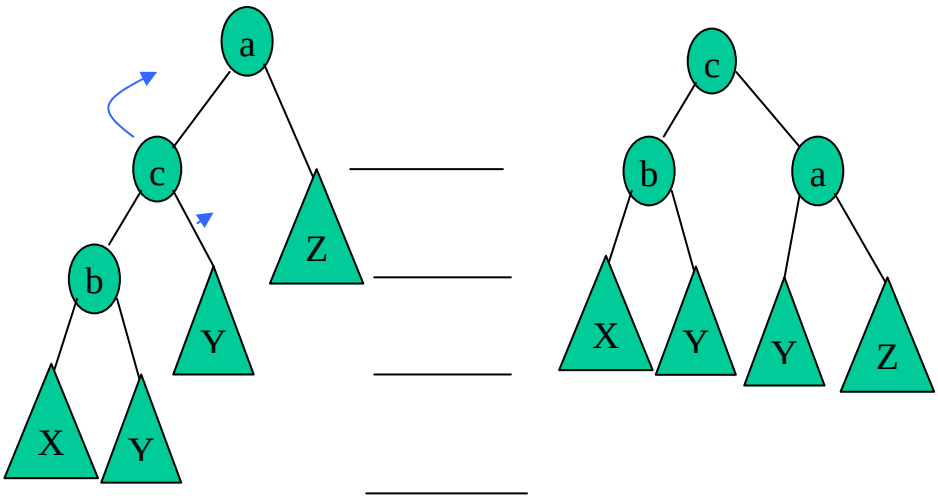
โดยส่วนตัว ผมคิดว่าแนวคิดแบบการหมุนสองที เข้าใจได้ง่ายกว่าแบบสไปเดอร์แมน



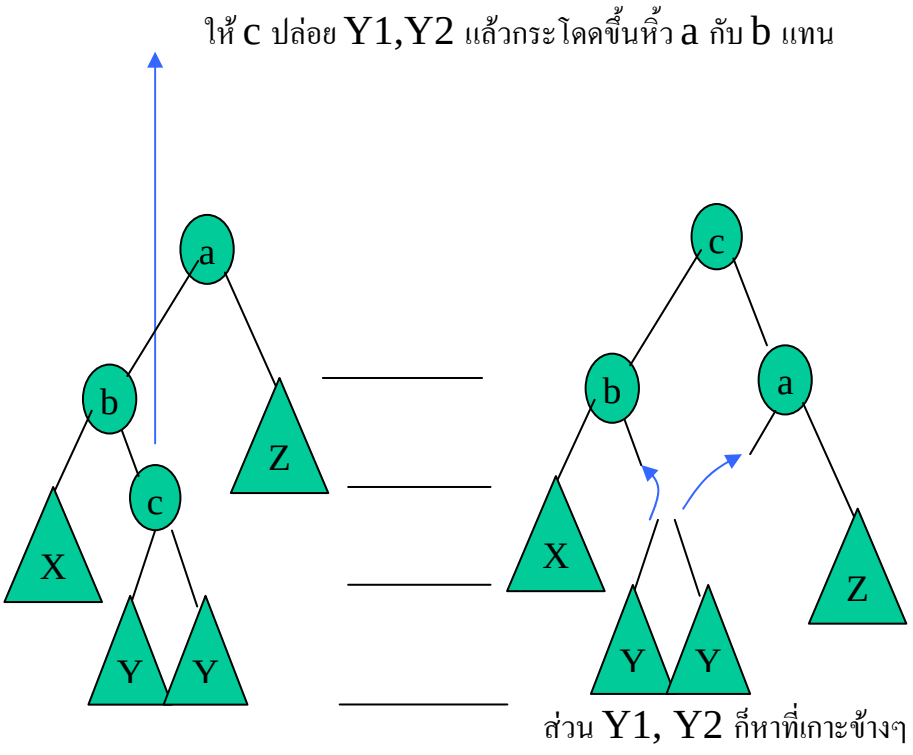
รูป 7.11 ต้นไม้จากรูป 7.10 เปลี่ยน Y ให้เป็นอีกรูปแบบหนึ่ง เพื่อเตรียมการ rotate



รูป 7.12 การหมุนครั้งที่หนึ่ง



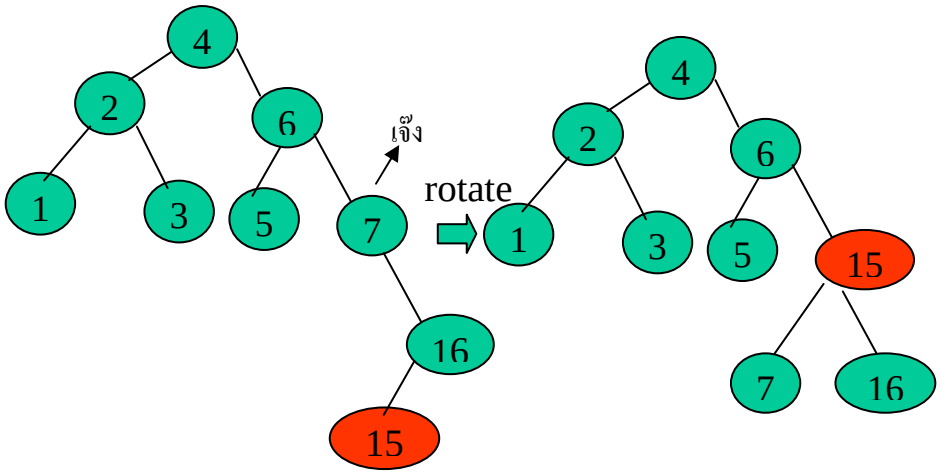
รูป 7.13 การหมุนครั้งที่สอง ซึ่งทำให้ความสูงลดลง



รูป 7.14 double rotation ทำในครั้งเดียว ด้วยแนวคิดที่ว่า c เป็นสไปเดอร์แมน

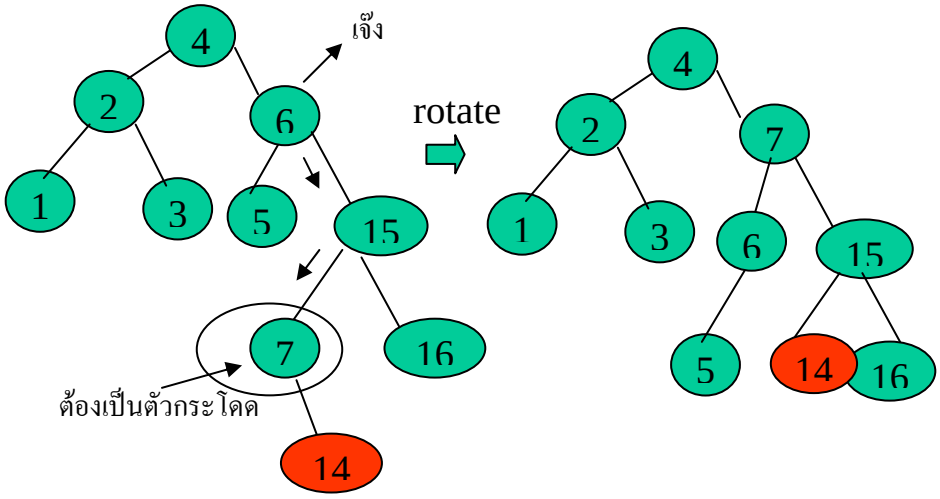
ตัวอย่างที่ 7-3

เติมเลข 16 กับ 15 เข้าไปในต้นไม้ผลลัพธ์จากรูป 7.9 ตอนเติมเลข 16 จะไม่มีอะไร แต่ตอนเติมเลข 15 จะเกิดการเสียความเป็น AVL ขึ้น ทำให้ต้องแก้ดังรูป 7.15



รูป 7.15 การใช้ double rotation แก้ความไม่เป็น AVL หลังจากเติม 16

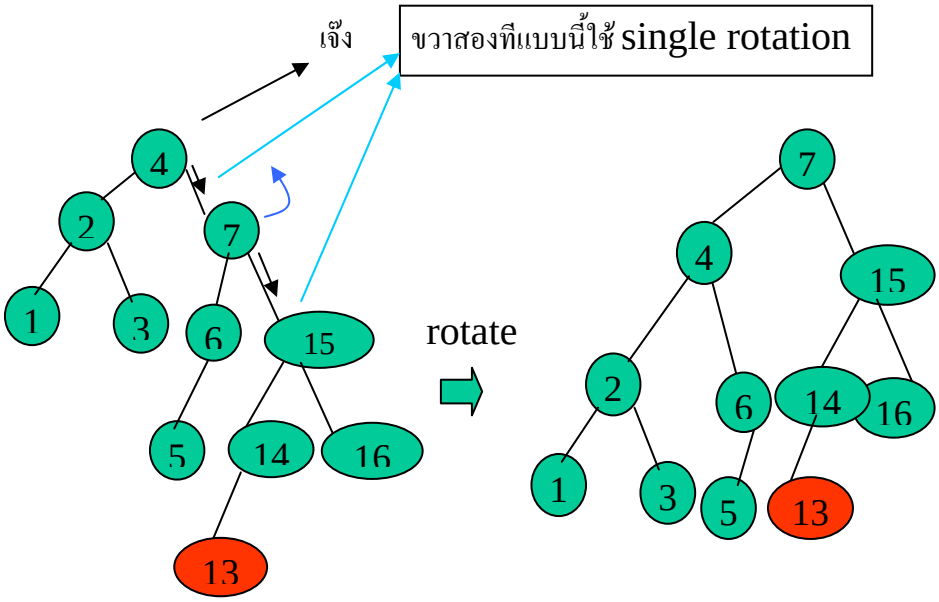
ต่อจากนั้น เราใส่ 14 ลงไป คราวนี้ทำให้โนดที่มีเลข 6 เสียความเป็น AVL (รูป 7.16)



รูป 7.16 double rotation หลังจากใส่ 14 ลงไปในต้นไม้

นับว่ากรณีนี้ไกลจากจุดที่ใส่ของ มากกว่ากรณีอื่นๆที่เคยเห็นมา แต่หลักการแก้ก็ยังเหมือนเดิม โดยต้อง rotate โนดที่มีเลข 7 ขึ้นไปสองที อย่าลืมว่าสถานการณ์ที่ต้องใช้ double rotation นั้นคือ เมื่อเราตามต้นไม้จากโนดที่เสียความเป็น AVL ไปยังโนดที่เติมเข้ามาใหม่ จะต้องไปซ้ายทีขวาทีต่อกันเลย หรือขวาทีซ้ายทีต่อกันเลย ถ้าลงมาทางซ้ายสองทีหรือขวาสองที ต้องใช้ single rotation แก้

ตัวอย่างที่ 7-4
เติมเลข 13 ลงไปในต้นไม้ที่เป็นผลลัพธ์ในรูป 7.16 จะเห็นว่าจากโนดที่เสียความเป็น AVL ลงไปยังโนดที่เราพึ่งใส่เข้าไปนั้น จะต้องลงมาทางขวาสองที แล้วจึงซ้ายสองที แต่เราสนใจแค่การลงต้นไม้มาสองครั้งแรกเท่านั้น ดังนั้นการใส่ 13 เข้ามาในต้นไม้นี้ ต้องแก้ด้วย single rotation ดังรูป 7.17



รูป 7.17 single rotation แก้ความไม่เป็น AVL ในต้นไม้ที่ใส่ของในโนดที่อยู่ลึก

ต่อไปเรามาดูตัวโค้ดกัน เริ่มจากโค้ดของโนดก่อน ซึ่งแสดงในรูป 7.18

```
1:     class AvlNode{
2:       // Constructors
3:       AvlNode( Comparable theElement){
4:          this(theElement, null, null);
5:       }
6:
7:       AvlNode(Comparable theElement, AvlNode lt, AvlNode rt){
8:          element  = theElement;
9:          left     = lt;
10:         right    = rt;
11:         height   = 0;
12:       }
13:
14:     //Friendly data; accessible by other package routines
15:      Comparable element;      // The data in the node
16:      AvlNode    left;         // Left child
17:      AvlNode    right;        // Right child
18:      int        height;       // Height
19:    }
```

รูป 7.18 โค้ดของโนดของ AVL

ตัวคลาส AVL Tree เองนั้น ก็ประกอบขึ้นมาจากโนดเท่านั้น ดังแสดงในรูป 7.19 โดยคอนสตรัคเตอร์จะสร้าง root ที่เป็น null ขึ้นมา

```
1:    public class AvlTree{
2:       private AvlNode root;
3:       public AvlTree( ){
4:          root = null;
5:       }
6:       //ให้ถือว่าส่วนของเมธอดอื่นๆจะอยู่ด้านล่างไป
7:    }
```

รูป 7.19 ตัวคลาส AvlTree

เมธอดอื่นๆนั้นเหมือนกับการใช้ต้นไม้ค้นหาแบบทวิภาคทั่วไปทุกประการ แต่เมธอดที่เราต้องปรับคือ insert และ remove ในชั้นเรียนนี้จะสอนการปรับเมธอด insert เท่านั้น ส่วนเมธอด remove นั้นผมขอให้พวกเราลองทำกันดูเองนะ

ก่อนที่จะทำการ insert ได้ถูกต้อง (และไม่ต้องเขียนโค้ดทีเดียวเยอะ) เราจำเป็นจะต้องเตรียมเมธอดต่างๆไว้ใช้งาน รูป 7.20 แสดงเมธอดที่หาความสูงของโนด t (ถ้าโนดเป็น null ให้รีเทิร์น -1) และเมธอดที่หาค่ามากที่สุดระหว่างจำนวนเต็มสองตัว

```
1:    /**
2:     * รีเทิร์นความสูงของโนด t หรือไม่ก็รีเทิร์น -1 ถ้า t เป็น null
3:    */
4:    private static int height( AvlNode t ){
5:        return t == null ? -1 : t.height;
6:    }
7:
8:    /**
9:    * รีเทิร์นค่าที่มากที่สุดระหว่าง lhs และ rhs
10:   */
11:   private static int max( int lhs, int rhs ){
12:       return lhs > rhs ? lhs : rhs;
13:   }
```

รูป 7.20 เมธอด height และ max

นอกจากนี้ยังมีเมธอดช่วยอีก 4 เมธอด ที่ทำการ rotate ต้นไม้ ทั้ง single rotation และ double rotation โดยแต่ละเมธอดนั้น ใช้สำหรับ rotate ในสถานการณ์ต่างๆกันดังต่อไปนี้

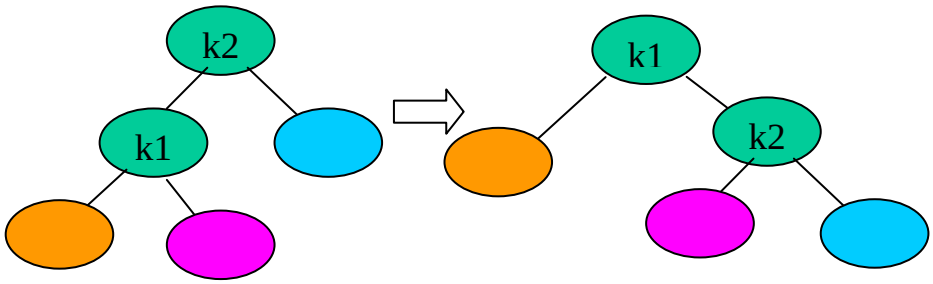
- rotateWithLeftChild ใช้เมื่อจุดที่เสียความเป็น AVL ต้องลงกิ่งข้างซ้ายไปสองทีจึงจะไปทางของที่เราใส่เข้ามา นี่เป็น single rotation
- rotateWithRightChild ใช้เมื่อจุดที่เสียความเป็น AVL ต้องลงกิ่งข้างขวาไปสองทีจึงจะไปทางของที่เราใส่เข้ามา นี่เป็น single rotation
- doubleWithLeftChild ใช้เมื่อจุดที่เสียความเป็น AVL ต้องลงซ้ายแล้วต่อด้วยขวา เป็น double rotation
- doubleWithRightChild ใช้เมื่อจุดที่เสียความเป็น AVL ต้องลงขวาแล้วต่อด้วยซ้าย เป็น double rotation

เรามาดูโค้ดของ rotateWithLeftChild กันก่อน ดูรูป 7.21 นะ จะสังเกตว่าโค้ดมีการเปลี่ยน height และรีเทิร์นโนดที่หมุนขึ้นไปแล้ว

```
1:     private static AvlNode rotateWithLeftChild(AvlNode k2){
2:       AvlNode k1 = k2.left;
3:       k2.left = k1.right;
4:       k1.right = k2;
5:       k2.height = max(height(k2.left), height(k2.right))+1;
6:       k1.height = max(height(k1.left),k2.height)+1;
7:       return k1;
8:     }
```

รูป 7.21 โค้ดของ rotateWithLeftChild

โดยการย้ายที่ของโนดต่างๆนั้นเป็นดังรูป 7.22 ซึ่งแสดงให้เห็นถึงการหมุนเอาโนด k2 ที่เสียความเป็น AVL ลงมา

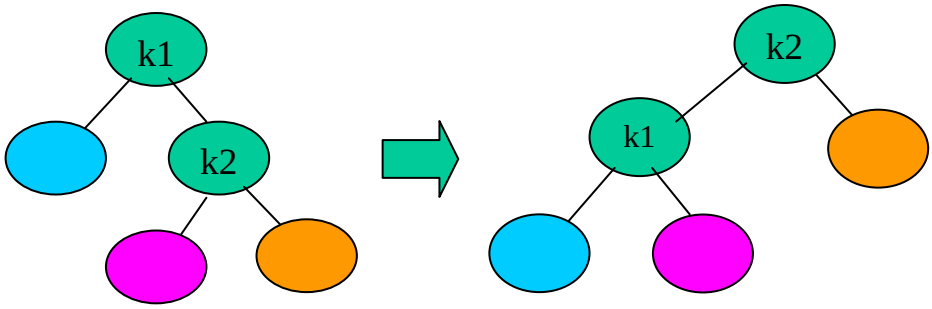


รูป 7.22 การย้ายโนดของ rotateWithLeftChild

โค้ดของ rotateWithRightChild ก็เพียงแต่กลับทางกับ rotateWithLeftChild เท่านั้น โค้ดอยู่ในรูป 7.23 และการย้ายโนดอยู่ในอยู่รูป 7.24

```
1:     private static AvlNode rotateWithRightChild(AvlNode k1){
2:        AvlNode k2 = k1.right;
3:        k1.right = k2.left;
4:        k2.left = k1;
5:        k1.height = max(height(k1.left), height(k1.right))+1;
6:        k2.height = max( height( k2.right ), k1.height ) + 1;
7:        return k2;
8:     }
```

รูป 7.23 โค้ดของ rotateWithRightChild

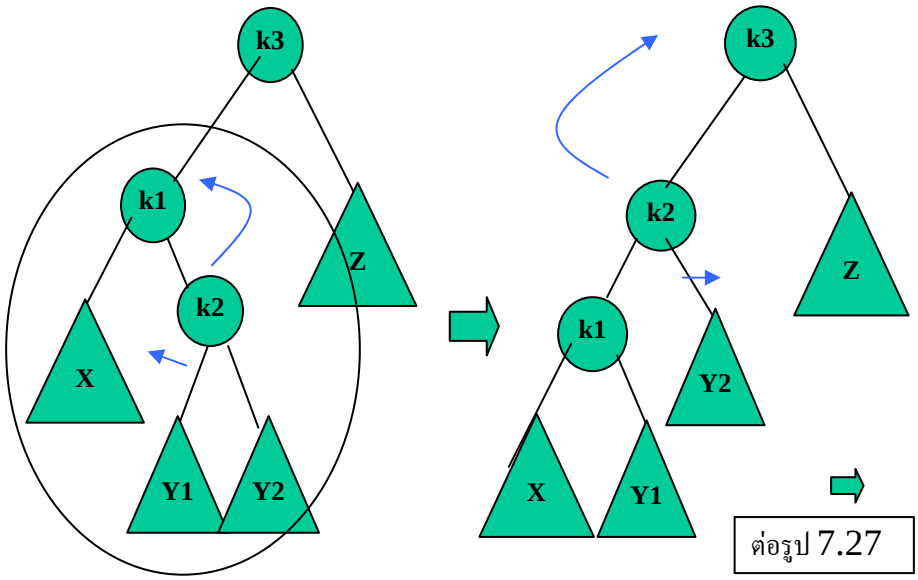


รูป 7.24 การย้ายโนดของ rotateWithRightChild

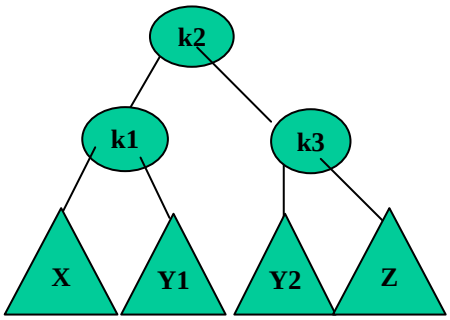
ต่อไปเป็นเมธอด doubleWithLeftChild ซึ่งเป็นการทำ double rotation โดยจริงๆแล้วทำถึงตรงนี้จะง่ายแล้วเพราะก็คือ single rotation สองที โค้ดของเมธอดนี้อยู่ในรูป 7.25 และส่วนแสดงการย้ายโนดนั้นอยู่ในรูป 7.26 และ 7.27 (rotate และผลลัพธ์สุดท้าย) ตามลำดับ

```
1:     private static AvlNode doubleWithLeftChild(AvlNode k3){
2:        k3.left = rotateWithRightChild( k3.left );
3:        return rotateWithLeftChild( k3 );
4:     }
```

รูป 7.25 โค้ดของ doubleWithLeftChild



รูป 7.26 การ rotate ย้ายโนดของ doubleWithLeftChild

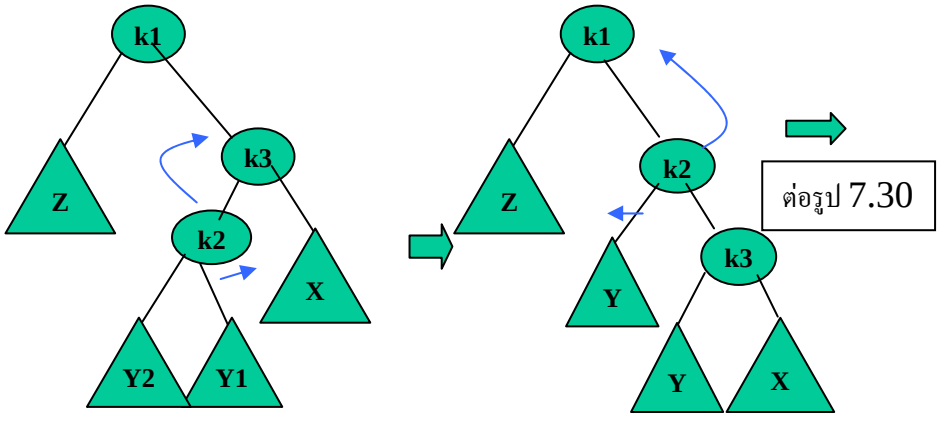


รูป 7.27 ผลลัพธ์การย้ายโนดด้วย doubleWithLeftChild

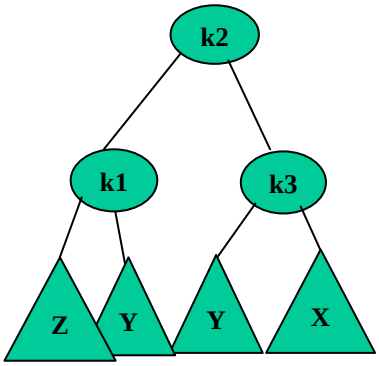
สำหรับเมธอด doubleWithRightChild นั้นก็ทำเหมือนกัน เพียงแต่หมุนคนละข้างเท่านั้น รูปที่ 7.28 แสดงโค้ดของ doubleWithRightChild ส่วนรูปที่ 7.29 และ 7.30 แสดงขั้นตอนการ rotate และผลลัพธ์สุดท้าย

```
1:     private static AvlNode doubleWithRightChild( AvlNode k1 ){
2:        k1.right = rotateWithLeftChild( k1.right );
3:        return rotateWithRightChild( k1 );
4:     }
```

รูป 7.28 โค้ดของ doubleWithRightChild



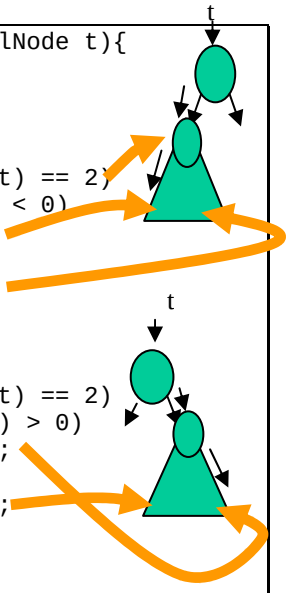
รูป 7.29 ขั้นตอนการหมุนต้นไม้ด้วย doubleWithRightChild



รูป 7.30 ผลลัพธ์จากการ rotate ด้วย doubleWithRightChild

สุดท้ายนี้ หลังจากเตรียมเมธอดต่างๆเรียบร้อย เราก็มาดูการ insert กัน โค้ดนั้นอยู่ในรูป 7.31

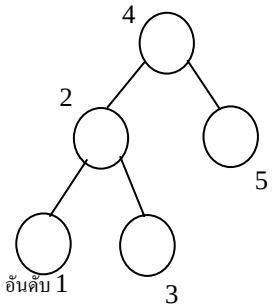
```
1:     private AvlNode insert(Comparable x, AvlNode t){
2:        if( t == null )
3:           t = new AvlNode( x, null, null );
4:        else if(x.compareTo(t.element) < 0){
5:           t.left = insert( x, t.left );
6:           if(height(t.left) - height(t.right) == 2)
7:              if(x.compareTo(t.left.element) < 0)
8:                 t = rotateWithLeftChild(t);
9:              else
10:                t = doubleWithLeftChild(t);
11:       }
12:       else if(x.compareTo(t.element) > 0){
13:          t.right = insert(x, t.right);
14:          if(height(t.right) - height(t.left) == 2)
15:             if(x.compareTo(t.right.element) > 0)
16:                t = rotateWithRightChild(t);
17:             else
18:                t = doubleWithRightChild(t);
19:       }
20:       else
21:          ;  // Duplicate; do nothing
22:       t.height = max(height(t.left), height(t.right)) + 1;
23:       return t;
24:    }
25:
26:    public void insert(Comparable x){
27:       root = insert( x, root );
28:    }
```



รูป 7.31 เมธอด insert ซึ่งเลือกใช้ การ rotate ต่างๆตามสถานการณ์

แบบฝึกหัด

1. ถ้าใน AvlNode มี instance variable: size (ซึ่งเป็น integer) เพิ่มเข้ามา ซึ่ง size นี่คือ จำนวนโนดของ left กับ right รวมกัน จงเปลี่ยน rotateWithLeftChild ให้มีค่า size ถูกต้อง
2. จงเขียนเมธอด findkth ใน ต้นไม้ AVL (ที่มี size ด้วย) ซึ่งเมธอดนี้หาค่าที่อยู่เป็นอันดับที่ k ในต้นไม้ โดยตัวอย่างของอันดับนั้นเป็นดังรูป



โดยเมธอดนี้มีหัวเมธอดคือ private AvLNode findKth(int k, AvlNode t){

3. จากผลลัพธ์ของรูป 7.17 จงวาดต้นไม้หลังจากมีการเติม 12, 11, 10, 8, 9 ลงไป ตามลำดับ
4. จงเขียนเมธอด remove เพื่อทำการเอาของออกจากต้นไม้ ในขณะเดียวกันก็ยังต้องให้ต้นไม้อยู่ในรูปของ AVL
5. จงเขียนเมธอด mergeAVL ซึ่งรับต้นไม้แบบ AVL อีกต้นหนึ่งแล้วมารวมเป็นต้นเดียวกัน โดยต้องยังคงความเป็น AVL อยู่
6. จงเขียนเมธอด removeSubTree ซึ่งรับโนดที่เป็นรากของต้นไม้ย่อย แล้วเอาทั้งต้นไม้ย่อยนั้นออกจากต้นไม้ AVL ของเรา โดยส่วนที่เหลือต้องยังคงความเป็น AVL อยู่