ค่ายอบรมโอลิมปิกวิชาการ 2 (วันที่ 1)

โครงสร้างข้อมูล: ต้นไม้

Data Structure: Tree

รัชดาพร คณาวงษ์

17-18 มีนาคม 2561

ศูนย์มหาวิทยาลัยศิลปากร

ต้นไม้ (Tree) สำหรับนักคอมพิวเตอร์

- เป็นโครงสร้างข้อมูลที่แสดงถึงความสัมพันธ์ของข้อมูลแบบมีลำดับชั้น โดยเปรียบเทียบจากส่วนประกอบต่างๆ ของต้นไม้ในโลกความจริง

ส่วนประกอบที่สำคัญคือ

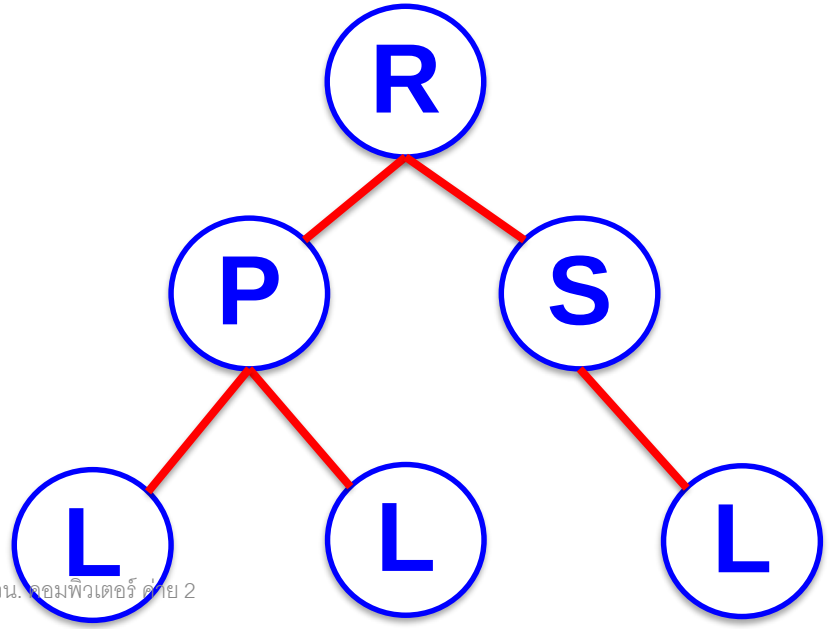
- ✧ ราก
- ✧ กิ่งก้าน
- ✧ ใบ

องค์ประกอบของต้นไม้

- ต้นไม้ในคอมพิวเตอร์มีองค์ประกอบอยู่สองแบบ
  - โหนด (node)
  - เส้น (edge) เส้นแสดงความสัมพันธ์ระหว่างโหนด 2 โหนด

หมายเหตุ

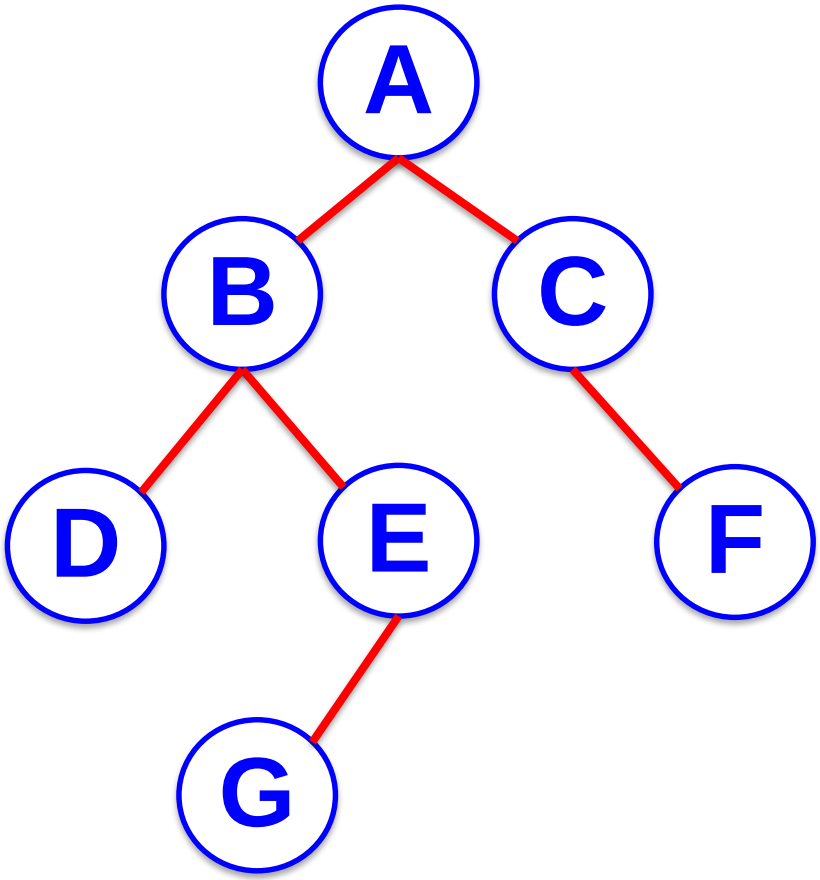
เพื่อความง่ายต่อการวาดต้นไม้ จึงนิยมวาดต้นไม้คว่ำ เริ่มจากให้รากอยู่บนสุด และวาดเส้นแทนกิ่งก้านแตกแขนงเป็นลำดับลงมาเรื่อยๆ



โหนดของต้นไม้

- โหนดบางตำแหน่งจะมีบทบาทแตกต่างกัน เช่น
  - ราก (root) คือโหนดที่อยู่บนสุด
  - ลูก (child) คือโหนดที่มีโหนดด้านบน
  - พ่อหรือแม่ (parent) คือโหนดที่มีโหนดเชื่อมต่อด้านล่าง (root ไม่ถือเป็น parent)
  - ใบ (leaf) คือโหนดที่ไม่มีโหนดเชื่อต่ออยู่ด้านล่าง
  - พี่น้อง (sibling, has same parent) โหนดที่มีพ่อร่วมกัน
  - โหนดภายใน (inner node) คือโหนดที่ไม่ใช่ leaf และ root

ตัวอย่าง โหนดของต้นไม้



- root คือ A
- parent ของ E คือ B
- parent ของ F คือ C
- child node ของ B คือ D,E
- child node ของ C คือ F
- inner node คือ B,C,E
- leaf node คือ D,G,F
- Sibling ของ E คือ D

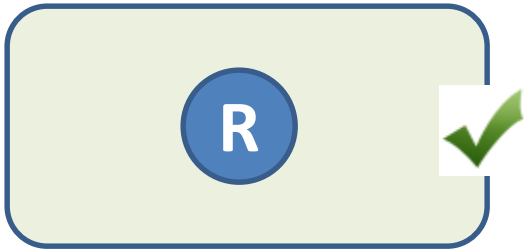
ลักษณะของต้นไม้ (Tree)

- ใช้สัญลักษณ์วงกลมแทนโหนด (node)
- เชื่อมวงกลมแต่ละวงด้วยเส้นตรง (edge)
- ต้นไม้จะมีรูทโหนดอยู่ด้านบนเพียงโหนดเดียวเท่านั้น
- โหนดแต่ละโหนดสามารถมีลูกได้ตั้งแต่ 0 - n โหนด ขึ้นกับประเภทต้นไม้
- โหนดลูกสามารถมีโหนดพ่อได้เพียงโหนดเดียวเท่านั้น
- ต้นไม้ว่าง (empty tree) ถือเป็นต้นไม้ แต่เป็นต้นไม้ที่ไม่มีโหนด

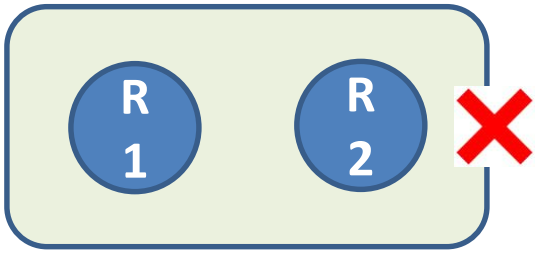
อะไรที่ใช่และไม่ใช่ทรี (สำคัญมากห้ามสับสน)

- ทรีทุกอันเป็นกราฟ แต่กราฟอาจไม่ใช่ทรี
- ไม่มีอะไรเลยก็เรียกว่าทรี (Empty tree)

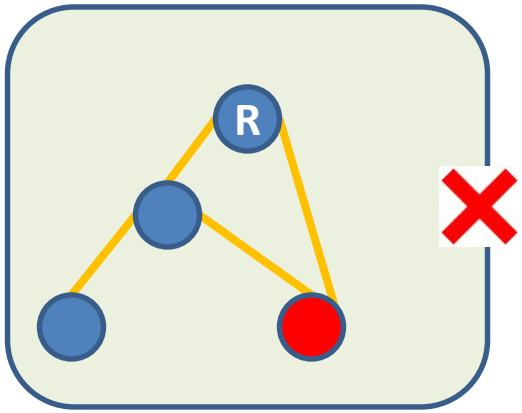
- มีแต่รูทก็เรียกว่าทรี



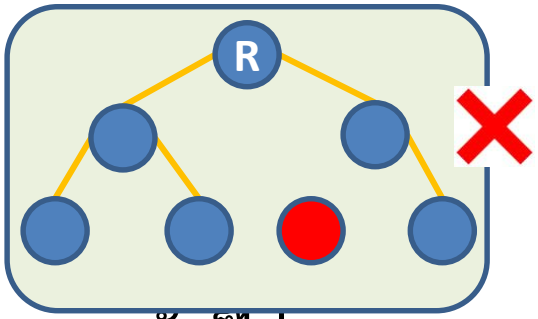
- แต่มีสองรูทถือว่าไม่ใช่ทรี



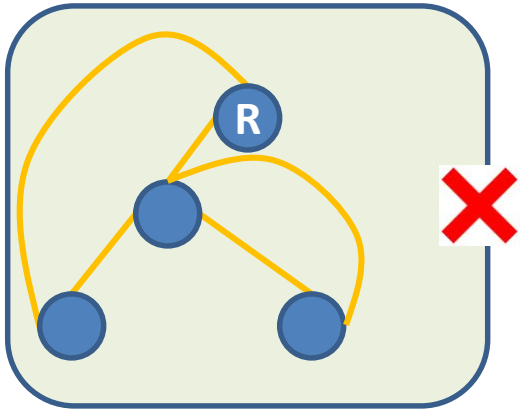
- มีโหนดที่มีหลายพ่อก็ไม่ใช่



- มีโหนดกำพร้าก็ไม่ได้ (โหนดกำพร้าที่จริงคือรูทอีกตัว)



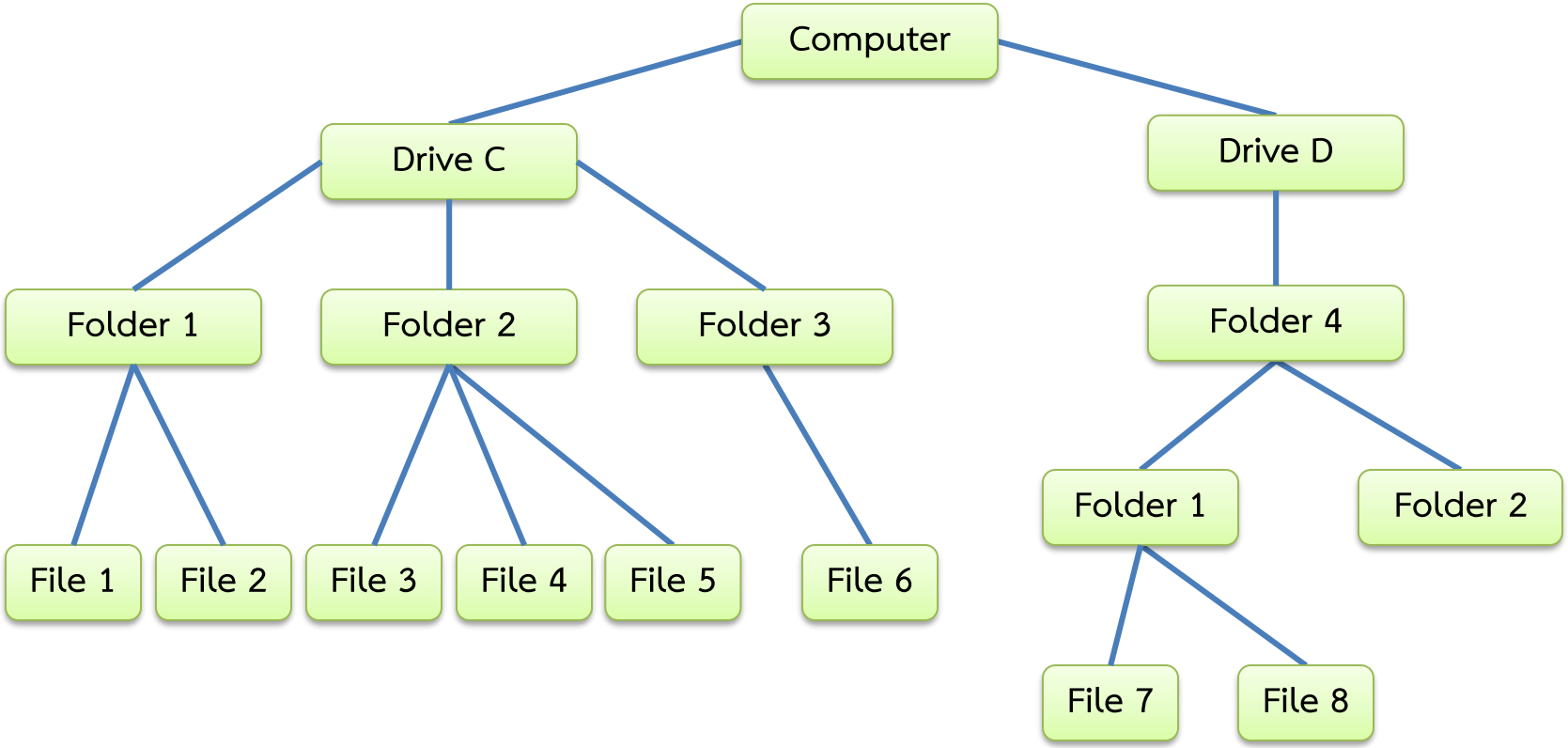
- วนกลับก็ไม่ยอม (การวนกลับทำได้ในกราฟบางประเภท แต่ไม่ใช่ต้นไม้)



**การดำเนินการ (operation) ของโครงสร้างข้อมูลแต่ละรูปแบบมีลักษณะเฉพาะของมัน จึงจำเป็นที่ต้องสร้างโครงสร้างข้อมูลที่ถูกต้อง

ตัวอย่างโครงสร้างต้นไม้หรือทรี (tree)

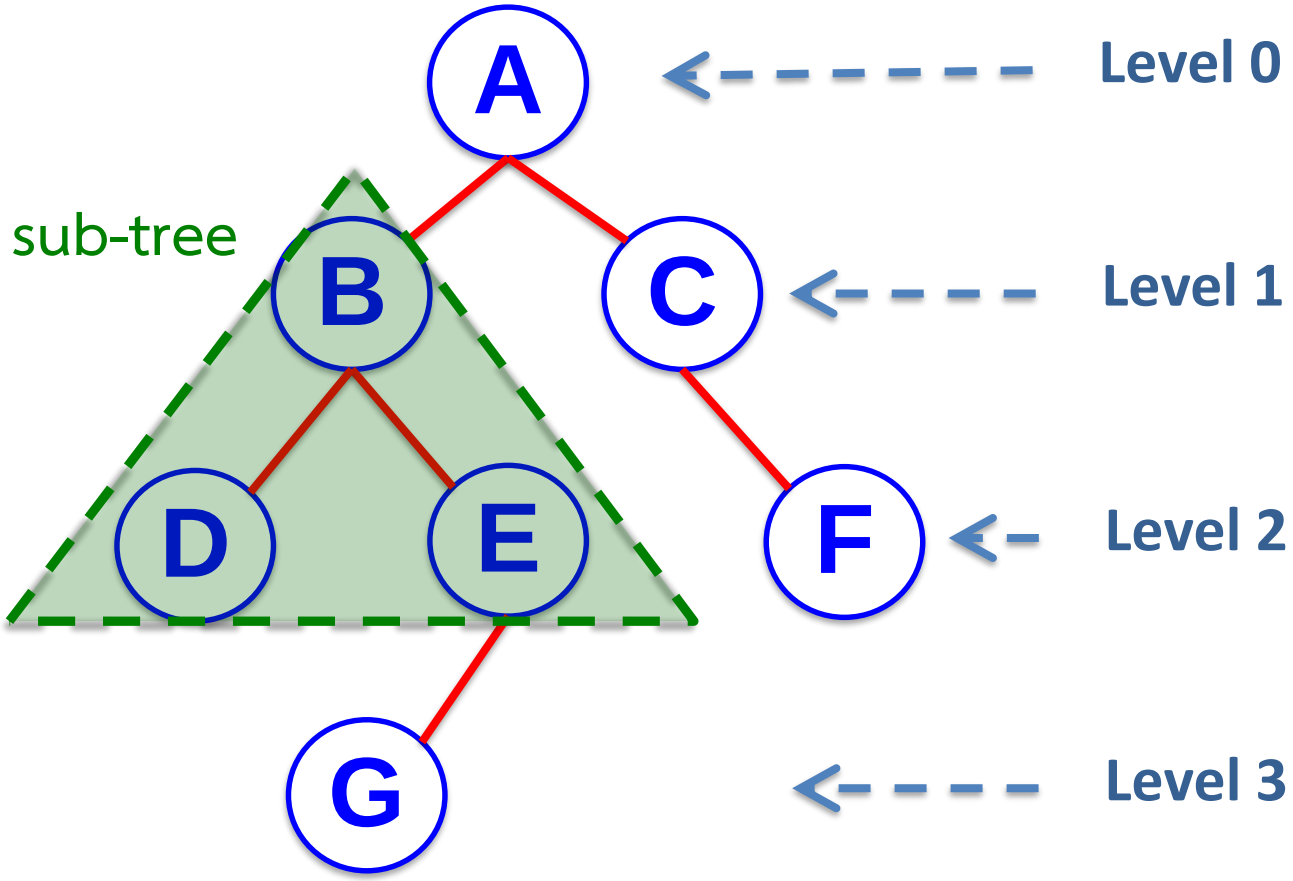
- โครงสร้างไฟล์และโฟลเดอร์



คำนิยามต่างๆ เกี่ยวกับทรี

- Path คือ เส้นทางจากโหนดใดโหนดหนึ่งไปยังโหนดสุดท้ายที่อยู่ในเส้นทางนั้น
- ต้นไม้ย่อย (sub-tree) กลุ่มของโหนดที่เชื่อมต่อกัน โดยมีโหนดที่อยู่บนสุดทำหน้าที่เสมือนเป็นรูท
- ระดับชั้น (level หรือ height) คือจำนวนเส้นที่ยาวที่สุดจากโหนดรูท (root) ถึงโหนดใบ (leaf)

นิยามด้วยภาพ



ตัวอย่างการประยุกต์ใช้ทรี

- Binary Search Tree

8	3	10	1	6	14	4	7	13

  - Parent มีลูกอย่างมากสองโหนด
  - ค่าของลูกด้านซ้ายน้อยกว่าค่าของ parent
  - ค่าของลูกด้านขวามากกว่าค่าของ parent

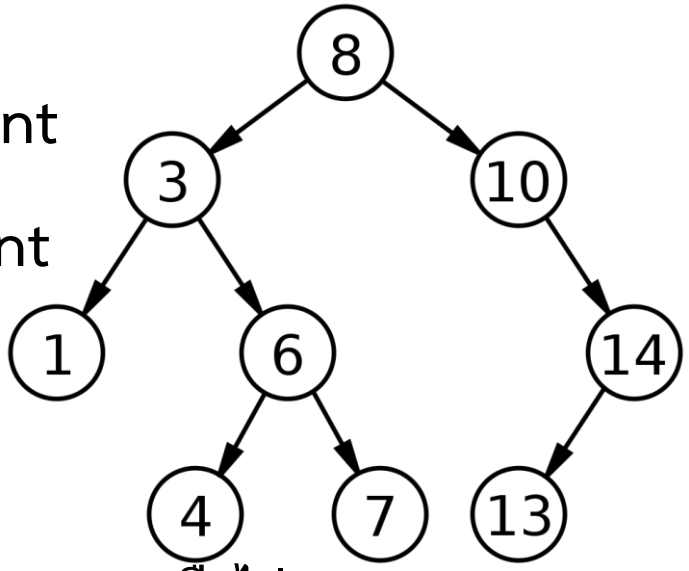


Image source: wikipedia.org

หมายเหตุ เนื่องจากทรีถูกเขียนด้วย linked-list จึงมีข้อดีกว่าการค้นหาด้วยการเรียงข้อมูลเก็บไว้ในอาร์เรย์ ในการแทรกค่าใหม่และลบค่าเดิมได้อย่างรวดเร็ว

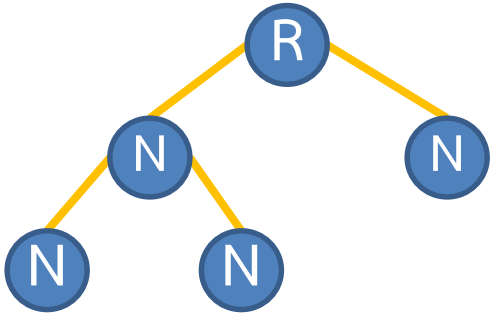
เราสามารถค้นได้อย่างรวดเร็วว่าค่า (value) ที่สนใจมีอยู่ในระบบหรือไม่

การประยุกต์ขั้นสูงขึ้นจะนำไปสู่โครงสร้างข้อมูลที่เรียกว่า Trie (ทรัย)

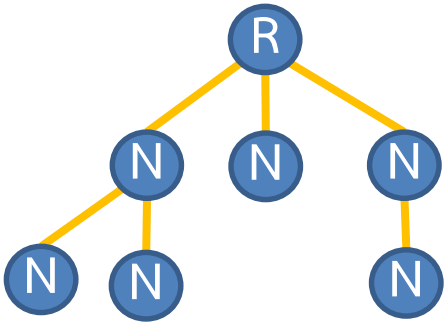
ทรัยมักถูกใช้กับการสร้างพจนานุกรมและการวิเคราะห์เอกสารข้อความ

ทรีแบบต่าง ๆ

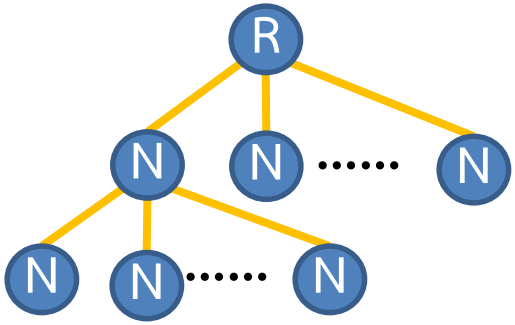
- แบ่งตามดีกรี (จำนวนโหนดลูกสูงสุดที่ยอมให้มีได้): Binary, Ternary, N-ary



Binary



Ternary



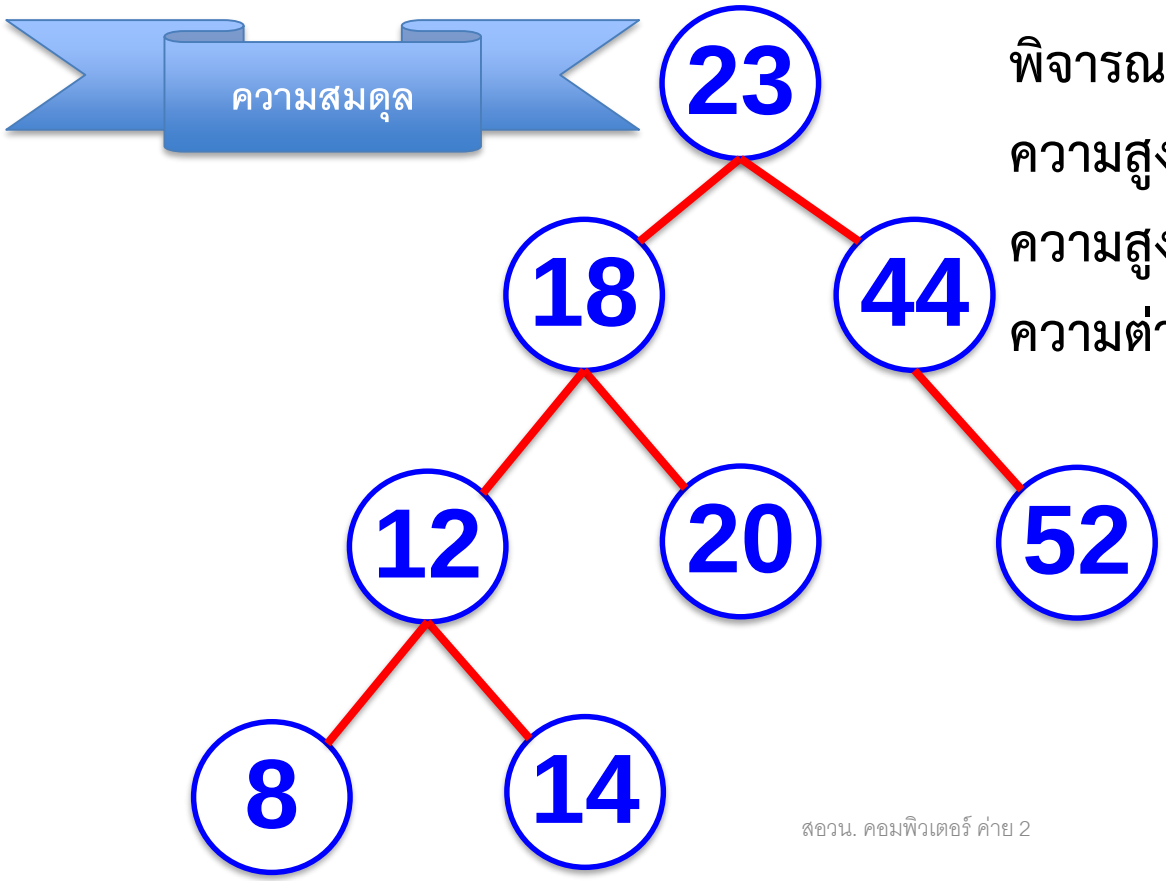
N-ary

** ดีกรี (degree) คือจำนวนโหนดลูกที่มีได้มากที่สุด

- Binary tree มักใช้งานแทนแบบอื่น ๆ ได้หมด แต่ประสิทธิภาพอาจจะไม่ดีนักในบางกรณี

ความสมดุล (balance) ของโหนดในทรี

- ความสมดุลเกิดจากการกำหนดความสูงด้านซ้ายและความสูงด้านขวาของ tree หรือ sub-tree ให้มีความแตกต่างกันไม่เกิน 1 ความสูง



พิจารณา รูท(node 23)

ความสูงด้านซ้าย (height of left:HL) 3

ความสูงด้านขวา (height of right:HR) 2

ความต่าง = |HL-HR| = |3-2| = 1

พิจารณา node 18

ความต่าง = |HL-HR| = |2-1| = 1

พิจารณา node 44

ความต่าง = |HL-HR| = |0-1| = 1

Balance and Complete Tree

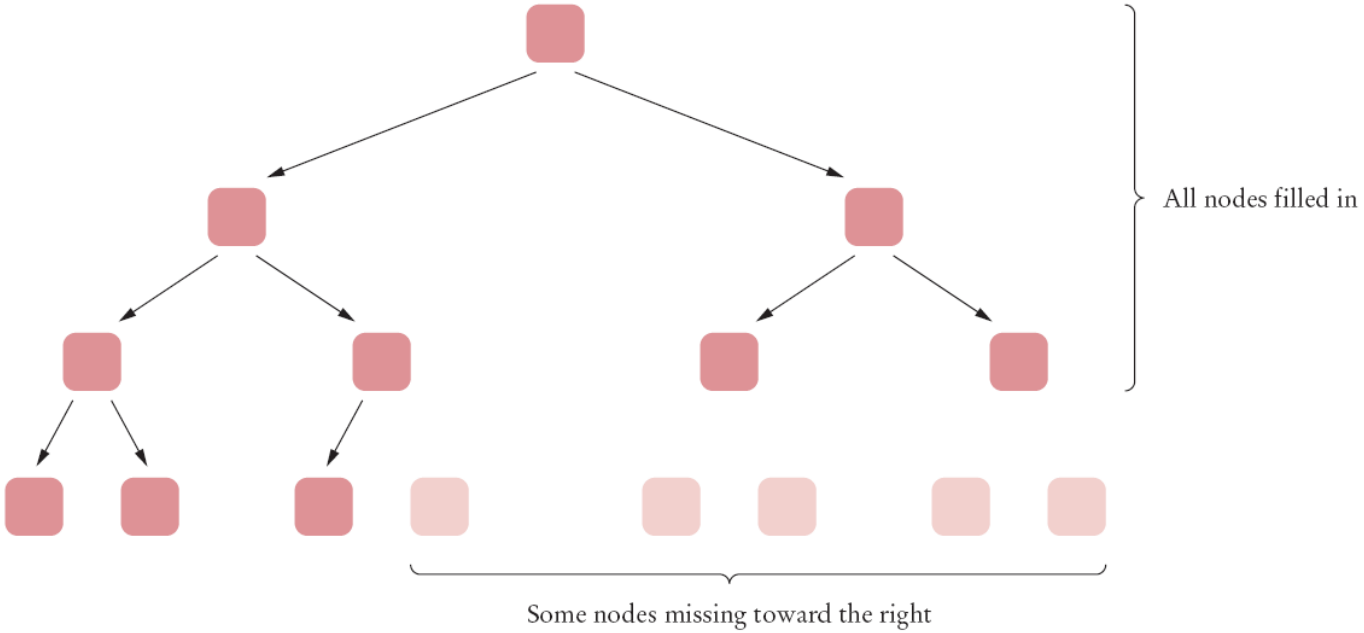
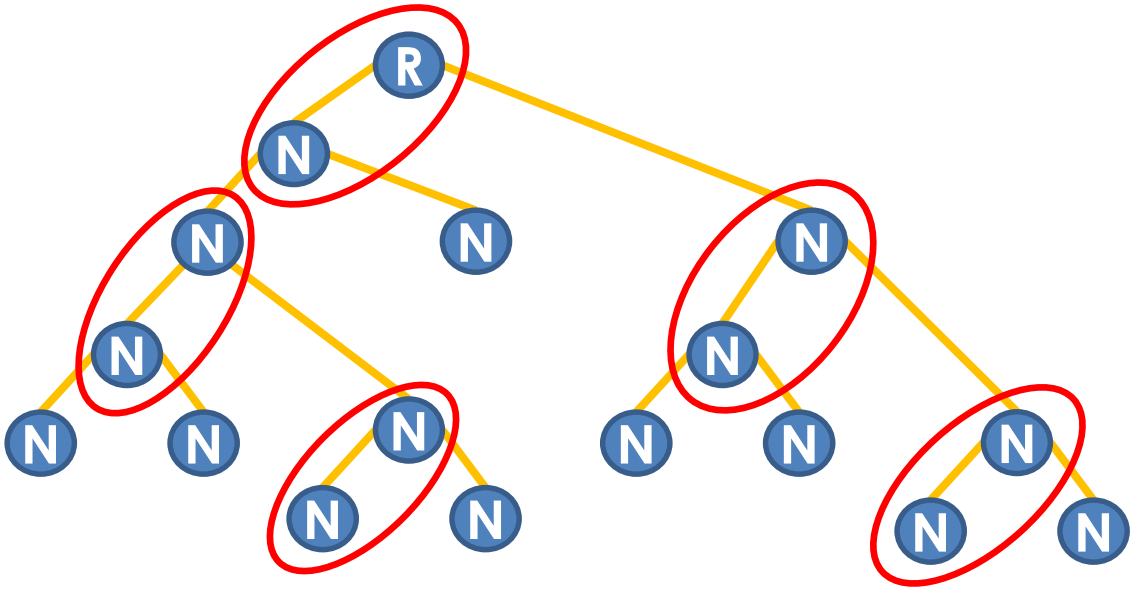


Figure 16 An Almost Complete Tree

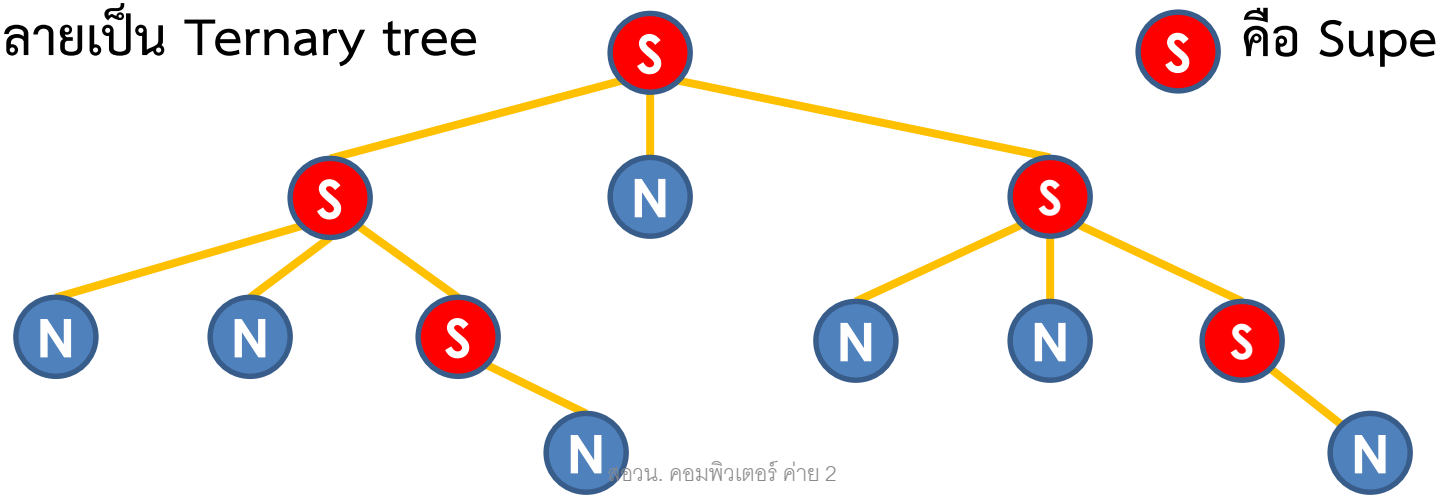
Image source: http://www.iro.umontreal.ca/~pift1025/bigjava/Ch21/ch21.html

ไบนารีทรีแทนทรีแบบอื่นได้



ยุบรวมสองโหนดพ่อและลูกด้านซ้ายเข้าด้วยกัน

กลายเป็น Ternary tree



S คือ Super node

มาปลูกต้นไม้ในคอมพิวเตอร์กัน

- เริ่มต้นด้วย binary tree ที่มีโหนด
- โดยโหนดจะมีส่วนข้อมูลและส่วนเชื่อมโยงไปยังโหนดอื่นไม่เกิน 2 โหนด


```
typedef struct treenode {
    EntryType key_value;
    struct treenode *llink;
    struct treenode *rlink;
} TreeNode;

struct treenode *root = 0;
```

ปลูกต้นไม้ด้วย C++ กัน (1)

- องค์ประกอบพื้นฐานที่สุด: โหนด

```
class TreeNode {
  public:
    Object key;            // Object is often int, string . . .
    TreeNode* left;
    TreeNode* right;
    TreeNode* parent;
    // TreeNode* nextSibling; // unnecessary for binary tree

    TreeNode(Object key) {
      this->key = key;
      left = right = parent = NULL;
    };
};
```

- เพื่อให้เห็นภาพเราจะปลูก Binary Search Tree ขึ้นมาสักต้น และแทน Object ด้วย int

ปลูกต้นไม้ด้วย C++ กัน (2)

- เตรียมต้นไม้เปล่าพร้อมตัวดำเนินการ (operator) ยอดนิยม

```
class Tree {
  TreeNode* root;
  TreeNode* insert(int key, TreeNode* root);
  TreeNode* remove(int key, TreeNode* root);

  TreeNode* find(int key, TreeNode* start);  // recursive version,
  TreeNode* find(int key, TreeNode* root);  // non-recursive
version,
  TreeNode* findMin(TreeNode* start, TreeNode* root);
  TreeNode* findMax(TreeNode* start, TreeNode* root);
}
```

สวยกว่า เขียนเสร็จ
เร็วกว่า ง่ายกว่า

ทำงาน
เร็ว โค้ด
เข้าใจง่าย

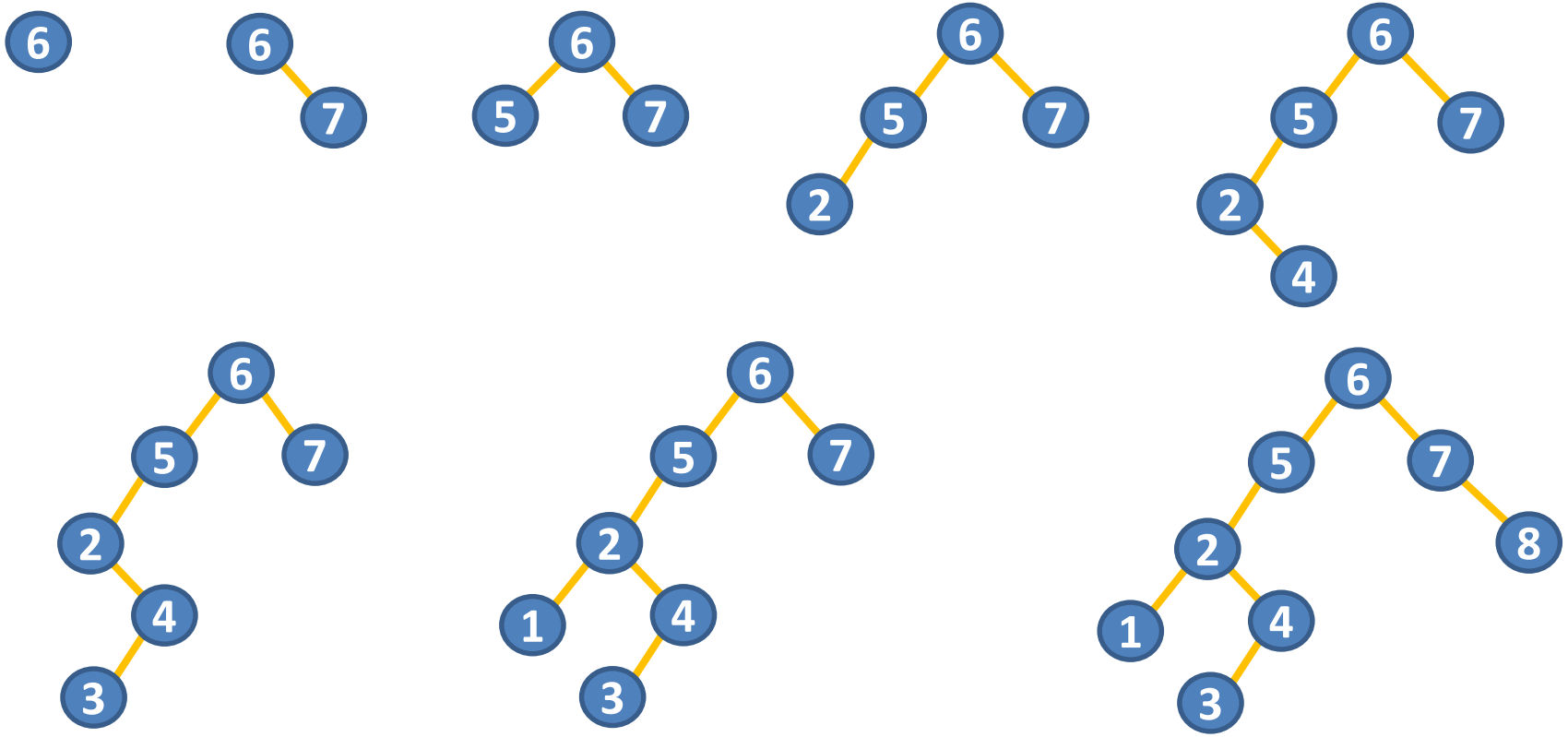
- เพาะรากขึ้นมาด้วยการ insert ค่าตัวแรกเข้าไป
  - ว่าแต่ต้องทำอย่างไง ถึงจะใส่ค่าต่าง ๆ เข้าไปใน binary search tree ได้อย่างถูกต้อง ?
  - อย่าลืมว่า binary search tree จัดลำดับตามค่าที่ใส่เข้าไป ค่าน้อยไปด้านซ้าย ค่ามากไปด้านขวา

การดำเนินการบนทรี (Operation on Tree)

- เปรียบเหมือนกับการตัดแต่ง และการเติบโตของต้นไม้
- จากต้นไม้ว่าง ๆ จะมีราก มีโหนดที่ถูกเติมและลบออก
- Operation ตัวแรก insert
  - ต้องเข้าใจพฤติกรรมของการใส่ค่าเข้าไปใน binary search tree
  - สมมติให้ลำดับของค่าที่จะใส่เข้าไปคือ 6 7 5 2 4 3 1 8

หลักการสร้างต้นไม้ไบนารี

- สมมติให้ลำดับของค่าที่จะใส่เข้าไปคือ 6 7 5 2 4 3 1 8



Insert โหนดแบบ Non-Recursive

- แนวคิดตรงไปตรงมา
- โค้ดค่อนข้างจะยาว เพราะการจำแนกแต่ละกรณีในการใส่โหนดเป็นเรื่องที่ค่อนข้างซับซ้อน
- มักทำงานเร็วกว่าแบบ recursive เล็กน้อยเพราะมีโอเวอร์เฮด (overhead) ในการทำงาน้อยกว่า

```
TreeNode* Tree::insertN(int key) {
```

ใส่โหนดแรก

```
  if (root == NULL) {
    root = new TreeNode(key);
    return root;
  }
```

ถ้า curr == NULL แสดงว่าเจอที่ใส่โหนด

```
  TreeNode* curr = root;
  TreeNode* prev = NULL;
  while(curr != NULL) {
    if (key == curr->key) {
// duplicate, do nothing and return NULL.
      return NULL;
    }
    else
```

ค่าน้อยไปทางซ้าย

```
      if (key < curr->key) {
         prev = curr;
         curr = curr->left;
      }
      else
```

ค่ามากไปทางขวา

```
        if (key > curr->key) {
          prev = curr;
          curr = curr->right;
        }
  }
```

```
  TreeNode* newNode = new TreeNode(key);
  newNode->parent = prev;
```

Update links

```
  if (key < prev->key) {
    prev->left = newNode;
  } else if (key > prev->key) {
    prev->right = newNode;
    }
  return newNode;
}
```

Insert โหนดแบบ Recursive

- โค้ดจะสั้นลง ดูสวยงามกว่าเดิม และตรวจสอบความถูกต้องได้ง่าย
- สามารถอ่านโค้ดให้เข้าใจได้โดยง่าย
- แนวคิด: เราสามารถมองโหนดลูกของรากว่าเป็นรากของต้นไม้ย่อยได้ และสามารถมองอย่างนี้ซ้อนไปเรื่อย ๆ ได้

```
TreeNode* Tree::insertR(int key, TreeNode*& current, TreeNode*
parent)
{
    if (current == NULL) {
        current = new TreeNode(key);
        current->parent = parent;
        return current;
    }
    if (key == current->key)
        return NULL;    // duplicate, do nothing and return NULL.
    else if (key < current->key)
        return insertR(key, current->left, current);
    else // key > current->key
        return insertR(key, current->right, current);
}
```

สุดยอดทริค น่าประทับใจมาก

ค้นหาโหนดที่มีค่า key สูงสุด/ต่ำสุด

- โหนดที่อยู่ทางขวาสุดคือโหนดที่มีค่ามากที่สุด ➔ มุ่งหน้าไปตาม node->right ไปเรื่อย ๆ
- โหนดที่อยู่ทางซ้ายสุดคือโหนดที่มีค่าน้อยที่สุด ➔ มุ่งหน้าไปตาม node->left ไปเรื่อย ๆ
- ไม่ค่อยมีความแตกต่างด้านการเขียนโค้ดสำหรับวิธีแบบ recursive

```
TreeNode* Tree::findMaxR(
                    TreeNode*
start) {
  if (start == NULL)
    return NULL;
  else if (start->right == NULL)
    return start;
  else
    return findMaxR(start-
>right);
}
```

```
TreeNode* Tree::findMaxN(
                      TreeNode*
root) {
  if (root == NULL)
    return NULL;
  else {
    TreeNode* curr = root;
    while (curr->right != NULL)
      curr = curr->right;
    return curr;
  }
}
```

ค้นหาโหนดที่มี key ที่เราสนใจ

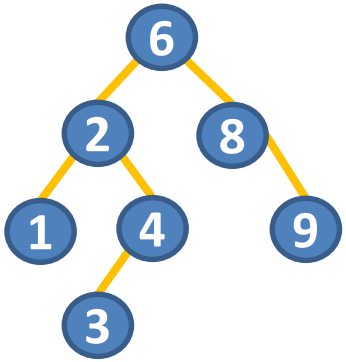
- ใช้ key ในการค้นหา ถ้าหากมีโหนดที่มี key ที่หาอยู่ ก็ให้คืน pointer ของโหนดนั้นไป

```
TreeNode* Tree::findR(int key,
                      TreeNode* start) {
  if (start == NULL)
    return NULL;
  else if (key == start->key)
    return start;
  else if (key < start->key)
    return findR(key, start->left);
  else
    return findR(key, start->right);
}
```

```
TreeNode* Tree::findN(int key) {
  if (root == NULL)
    return NULL;

  TreeNode* curr = root;
  while(curr != NULL) {
    if (curr->key == key)
      return curr;
    else if (key < curr->key)
      curr = curr->left;
    else if (key > curr->key)
      curr = curr->right;
  }

  return NULL;     // No match
}
```

การลบโหนด (Remove Node)

- ใช้ key ในการค้นหาและลบโหนดออกไป
- เป็นการดำเนินการที่นับว่าซับซ้อนพอสมควรเพราะต้องรักษาความเป็น binary search tree ไว้
- สามารถนำมาประยุกต์ใช้กับการเปลี่ยนค่าโหนดได้ เช่น
  ลบโหนดที่จะเปลี่ยนออก แล้วใส่โหนดใหม่ที่มีค่าที่ต้องการเข้าไป
- โค้ดแบบ non-recursive ยืดยาวและอาจเขียนผิดได้ง่าย
- ก่อนเขียนโค้ดต้องเข้าใจวิธีรักษาคุณสมบัติของ Binary search tree ไว้ให้ได้ก่อน



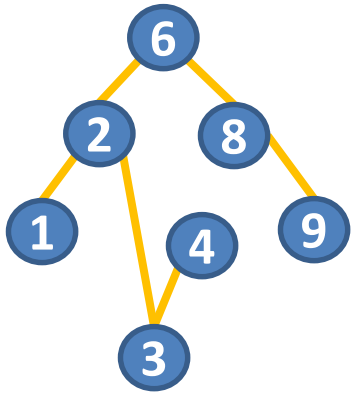
ต้องการลบ 4 ออกจากต้นไม้

มันเป็นเรื่องง่าย ถ้าโหนดที่ถูกลบมีลูกแค่อันเดียว

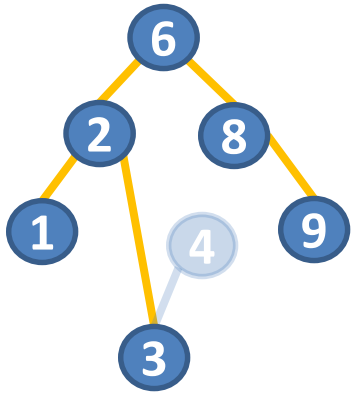
➔ โยงลิงค์ใหม่ได้เลย

โหนดทางต้นไม้ย่อยทางขวา ยังไงก็มีค่ามากกว่าโหนดทางซ้าย

➔ ลบโหนดที่ไม่ต้องการทิ้งไปได้เลย



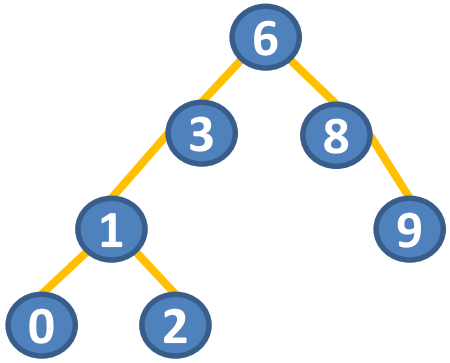
โยงลิงค์ใหม่



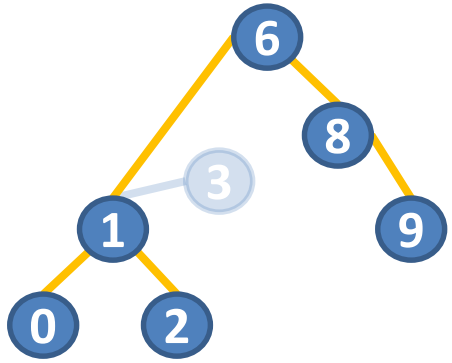
ลบโหนดทิ้ง

การลบโหนด (Remove Node) (2)

- ถ้าโหนดที่โดนลบมีลูกแค่โหนดเดียวถึงแม้จะมีทั้งลูกและหลาน ยังไงก็ง่าย

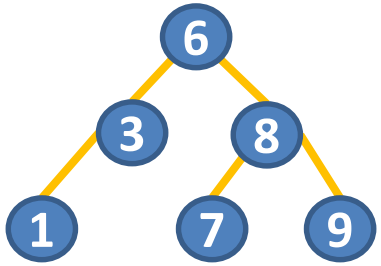


ต้องการจะลบ 3 ออกจากต้นไม้

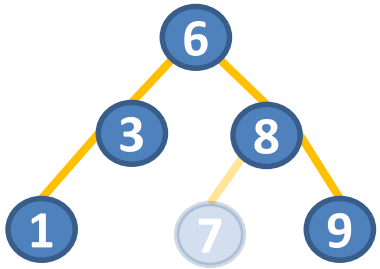


เปลี่ยนลิงค์ และลบออกได้เหมือนเดิม แทบไม่มีอะไรต่างกันเลย

- ยิ่งง่ายเข้าไปอีก ถ้าโหนดที่ถูกลบเป็นใบ (leaf) คือไม่มีโหนดลูก



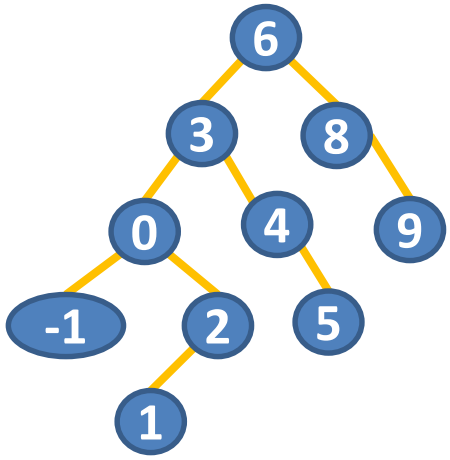
ต้องการจะลบ 7 ออกจากต้นไม้



อย่าลืมอัพเดตลิงค์ของโหนด 8 ให้กลายเป็น NULL

การลบโหนด (Remove Node) (3)

- ถ้าโหนดที่จะลบมีลูกอยู่สองโหนด การดำเนินงานจะกลายเป็นเรื่องซับซ้อนขึ้นมาทันที

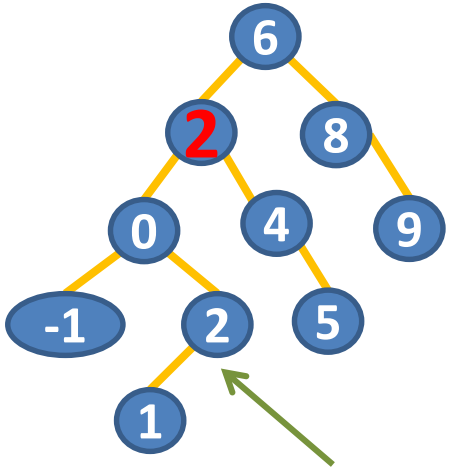


ต้องการจะลบ 3 ออกจากต้นไม้

ถ้าเราแทนค่าโหนด 3 ด้วยค่าในโหนด 2

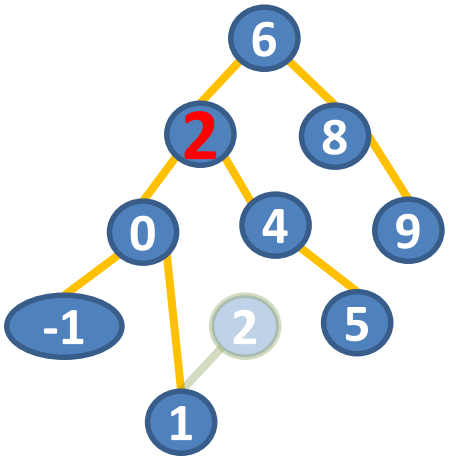
จะเปรียบได้ว่าโหนด 3 ถูกลบออก

(copy key ของโหนด 2 ไปทับโหนด 3)



มีโหนดลูกอันเดียว ลบด้วยวิธีเดิม ๆ ได้

ผลลัพธ์จากการลบโหนด 2



ผลลัพธ์ที่ได้รักษาคุณสมบัติของ binary search tree ไว้ได้ทุกประการ

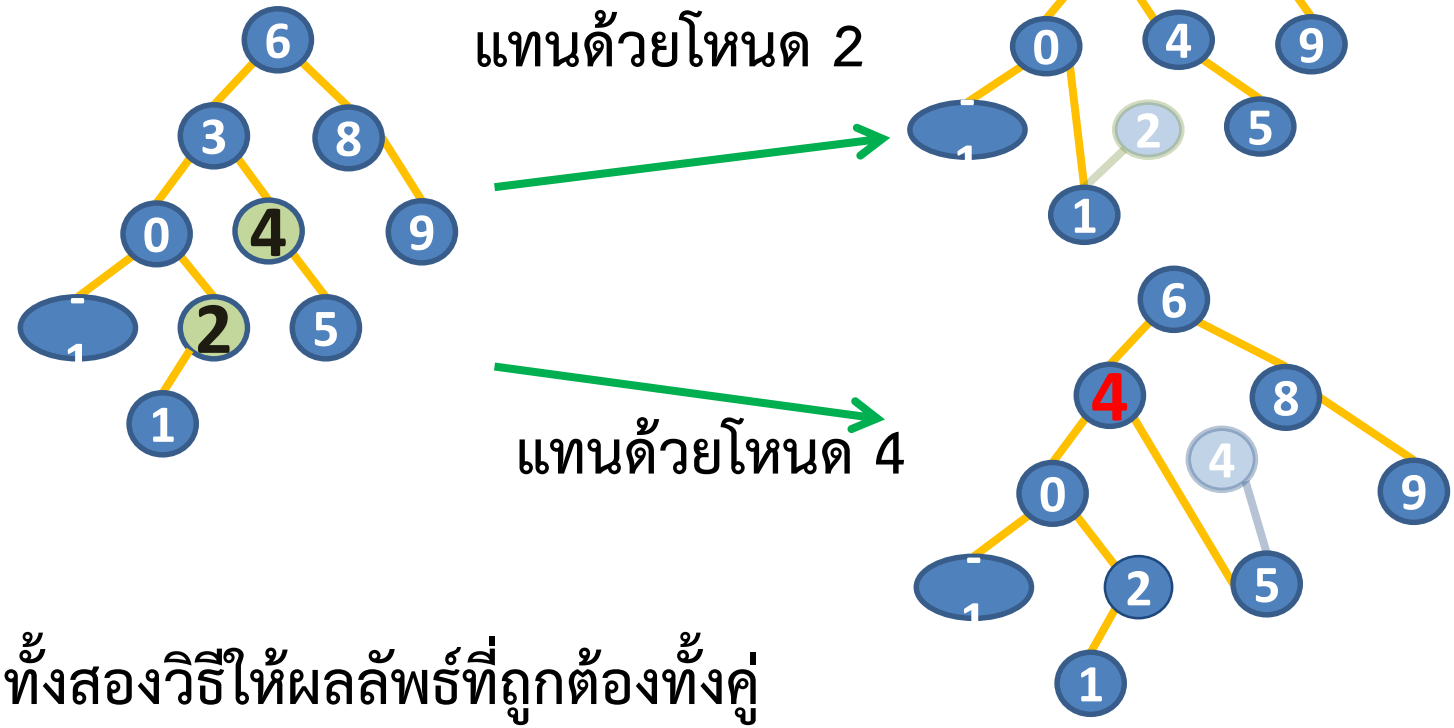
“แล้วรู้ได้ไงว่าต้องเลือกโหนด 2 มีหลักการเลือกหรือเปล่า ?”

การลบโหนด (Remove Node) (4)

- ถ้าเลือกโหนดที่มีค่ามากสุดในต้นไม้ย่อยด้านซ้าย หรือ เลือกโหนดที่มีค่าน้อยสุดในต้นไม้ย่อยทางขวา
  จะรับประกันได้ว่า
  1. การแทนค่าเข้าไปในโหนดที่ถูกสั่งลบจะไม่ผิดกฎ
  2. โหนดที่ถูกเลือกมาแทนที่จะมีลูกแค่โหนดเดียวเป็นอย่างมากเสมอ

ลองเลือกทั้งสองวิธี

ต้องการจะลบ 3 ออกจากต้นไม้



ทั้งสองวิธีให้ผลลัพธ์ที่ถูกต้องทั้งคู่

เพื่อความง่ายในการสอน จะเลือกใช้เฉพาะแบบแรก

การลบโหนดเกิดขึ้นได้สี่กรณี

1. โหนดที่ถูกลบเป็นใบ (ไม่มีลูก) ➔ เปลี่ยนลิงค์ของพ่อให้เป็น NULL และลบใบทิ้ง
2. โหนดที่ถูกลบมีลูกสองโหนด ➔ เขียนทับค่าโหนดแล้วลบโหนดที่ถูกเลือกมาแทนที่
3. โหนดที่ถูกลบมีเฉพาะโหนดลูกทางด้านซ้าย ➔ เปลี่ยนลิงค์แล้วลบโหนด
4. โหนดที่ถูกลบมีเฉพาะโหนดลูกทางด้านขวา ➔ เปลี่ยนลิงค์แล้วลบโหนด

<u>สองกรณีหลังสามารถยุบรวมกันเวลาเขียนโค้ดเพราะทำงานคล้ายกันมาก</u>

C++ Code สำหรับการลบโหนด

มีการใช้ pointer กับ pass-by reference ที่สวยงามมาก

```
void Tree::removeR(int key, TreeNode*& start) {
    if (start == NULL)              // Nothing to remove
        return;
    else if (key < start->key)  // Search for target node
        removeR(key, start->left);
    else if (key > start->key)
        removeR(key, start->right);
    else if (start->left != NULL && start->right != NULL) {
        // key == start->key and has two children
        TreeNode* leftMax = findMax(start->left);
        start->key = leftMax->key;
        removeR(leftMax->key, start->left);
    }
    else {    // no child or exactly one child
        TreeNode* temp = start;
        if (start->left != NULL)
            start = start->left;
        else
            start = start->right;
        delete temp;
    }
}
```

ทำให้ต้นไม้มีประโยชน์กว่าเดิม

ปรับโครงสร้างข้อมูลด้วยการใส่ field/operator/rule เพิ่มเติม (augment data structure)

- ใส่ตัวนับจำนวนข้อมูลซ้ำ
  - นับความถี่ของข้อมูล
  - Frequency dictionary (พจนานุกรมที่นับความถี่คำในเอกสาร--มีประโยชน์มาก)
- เพิ่มกฎในการบังคับให้ต้นไม้สมดุล (เช่น Red-Black Tree) เพื่อรับประกันความเร็วในการทำงาน
- เชื่อม key กับข้อมูลที่สนใจที่อยู่บนดิสก์
  - เทคนิคนี้ทำให้เราดำเนินการกับ key บนเมมโมรี โดยไม่ต้องเคลื่อนข้อมูลที่อยู่บนดิสก์จนกว่าจะถึงเวลาที่จำเป็นจริง ๆ
  - ใช้ได้กับโครงสร้างข้อมูลอื่น ๆ เช่น อาเรย์

!!! อย่ากลัวที่จะดัดแปลงโครงสร้างข้อมูลเพื่อให้มันทำงานที่เราต้องการได้_มันเป็นเรื่องปรกติ

ตัวอย่าง: การนับความถี่ข้อมูล

- เพิ่ม field (variable) ใหม่เข้าไปเพื่อทำการนับ

```
class TreeNode {
  public:
    Object key; // Object is often int, string, …
    int count;
    TreeNode* left;
    TreeNode* right;
    TreeNode* parent;
    TreeNode(Object key);
};
TreeNode::TreeNode(Object key) {
  this->key = key;
  count = 1;
  left = right = parent = NULL;
}
```

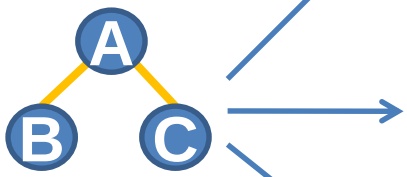
การ insert กับ remove โหนดก็ต้องเปลี่ยนไปจากเดิมด้วย

- ในตอน Insert ถ้ามีโหนดอยู่แล้วก็ให้เพิ่มตัวนับ (counter)
  ถ้าไม่มีก็ให้ใส่โหนดใหม่เข้าไปและตั้งตัวนับให้เป็น 1 (คล้ายแบบเดิมแต่มี counter มาเกี่ยวข้อง)
- การ Remove ถ้ามีซ้ำเกิน 1 ตัวก็ไม่ต้องลบโหนดออก แต่ให้ปรับ counter ให้ลดลงแทน
  ถ้ามีแค่ตัวเดียวก็ให้ลบโหนดออกไปเลย (คล้ายแบบเดิม)

การแวะผ่านต้นไม้ (Tree Traversal)

- เป็นการเดินเยี่ยมโหนดทุกโหนดในต้นไม้ (visit all nodes in a tree)
- มีอยู่สามลักษณะคือแบบ In-order (ตามลำดับ), pre-order (ก่อนลำดับ), และ post-order (หลังลำดับ)
- มุมมองของการนับลำดับดูที่ parent node เป็นตัวอ้างอิง และมองซ้อนแบบเดิมไปเรื่อย ๆ

ตัวอย่างแบบง่าย (ยังไม่จำเป็นต้องมองแบบ recursive)



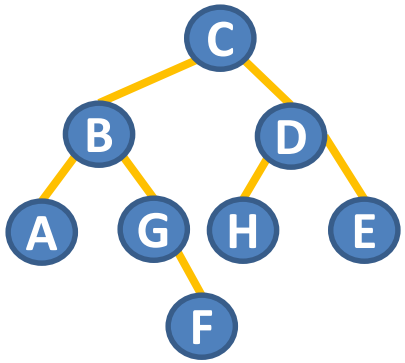
In-order → left child, parent, right child: B

pre-order → parent, left child, right child:

post-order → left child, right child, parent:

การแวะผ่านต้นไม้ (Tree Traversal) (2)

ถ้าต้นไม้ซับซ้อนขึ้นให้พิจารณาแบบ recursive



pre-order

C B A G F D H E

post-order

A F G B H E D C

สมมติว่าจะแวะผ่านแบบ in-order

1. จาก root (โหนด C) เราจะต้องแวะไปที่ลูกด้านซ้ายก่อน ซึ่งก็คือโหนด B
2. แต่โหนด B ก็ต้องแวะผ่านแบบ in-order เหมือนกัน เราจึงต้องแวะไปที่โหนด A ซึ่งเป็นลูกด้านซ้ายก่อน
3. โหนด A ไม่มีลูก → จัดการแวะได้เลย แล้ววกกลับหาโหนดพ่อ (โหนดB)
4. โหนด B ตอนนี้เยี่ยมลูกทางซ้ายแล้ว ก็แวะเยี่ยมตัวเองได้ แล้วไปลูกทางขวา
5. โหนด G ไม่มีลูกทางซ้าย แวะตัวเองได้เลย แล้วไปลูกทางขวา
6. โหนด F ไม่มีลูก แวะโหนด F ได้เลย แล้วย้อนกลับไป (ขณะนี้ลำดับการแวะคือ A B G F)
7. โหนด G กับ B ได้รับการแวะแบบ in-order ไปแล้ว จึงย้อนขึ้นไปถึงโหนด C
8. แวะโหนด C (สังเกตด้วยว่าลูกทางซ้ายทั้งหมดของ C ถูกแวะหมดแล้ว)
9. ทำต่อไปในลักษณะเดียวกันที่ต้นไม้ทางขวา จะได้ลำดับการแวะผ่านเป็น A B G F C H D E

การแวะผ่านต้นไม้แบบ In-order

ตอนแรกดูเหมือนจะยาก แต่พอลองเขียนโค้ดแบบ recursive ดู จะรู้ว่าง่ายมาก

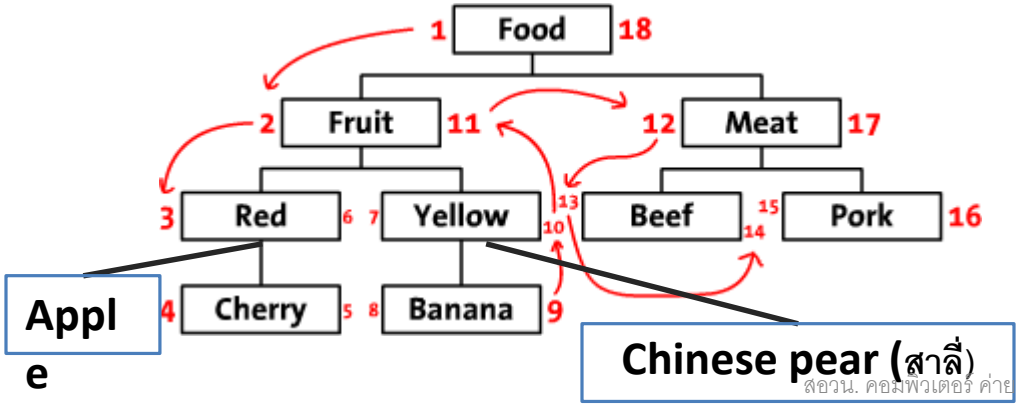
```
void inorder(TreeNode* current) {
    if (current == NULL)
        return;
    else {
        inorder(current->left);
        print(current);
        inorder(current->right);
    }
}
```

Tip: การแวะผ่านต้นไม้จะมีการเช็ค pointer ของลิงค์ในโหนดทุกโหนดทุกอัน เราสามารถใช้การแวะผ่านตรวจดูได้ว่าต้นไม้ของเรามีลิงค์ที่ใช้ไม่ได้อยู่หรือไม่ (ช่วยในการตรวจความถูกต้องของโปรแกรม)

การแวะผ่านต้นไม้ไปทำอะไรได้บ้าง

มีการประยุกต์ใช้หลายอย่างที่ต้องการนำเอาข้อมูลทั้งหมดในต้นไม้ออกมาประมวลผล เช่น

1. การค้นหาไฟล์ที่ต้องการในดิสก์ หรือ ในโฟลเดอร์
   (หวังว่าจะจำกันได้ว่า โครงสร้างโฟลเดอร์มักถูกจัดเก็บด้วยทรี)
2. การจัดเก็บและคำนวณนิพจน์ทางคณิตศาสตร์ (Math Expression)
3. งานวิจัยยุคใหม่ ๆ ก็ยังมีการพูดถึงการใช้งานกันอย่างชัดแจ้ง
   Use of tree traversal algorithms for chain formation in the PEGASIS data gathering protocol for wireless sensor networks. โดย Meghanathan, Natarajan
   (http://www.freepatentsonline.com/article/KSII-Transactions-Internet-Information-Systems/226163552.html)
4. การเก็บข้อมูลแบบลำดับชั้นในฐานข้อมูล (storing hierarchical data in a database)
   (Image source: http://articles.sitepoint.com/article/hierarchical-data-database/2)



ในการประยุกต์ใช้จริง อาจจะไม่ต้องแวะผ่านต้นไม้ทั้งหมด แต่อาจจะต้องแวะผ่านต้นไม้ย่อยแทน เช่น การหาว่ามีผลไม้กี่สีและอะไรบ้าง

ค่ายอบรมโอลิมปิกวิชาการ 2 (วันที่ 2)

โครงสร้างข้อมูล: ต้นไม้
Data Structure: Tree

รัชดาพร คณาวงษ์

17-18 มีนาคม 2560

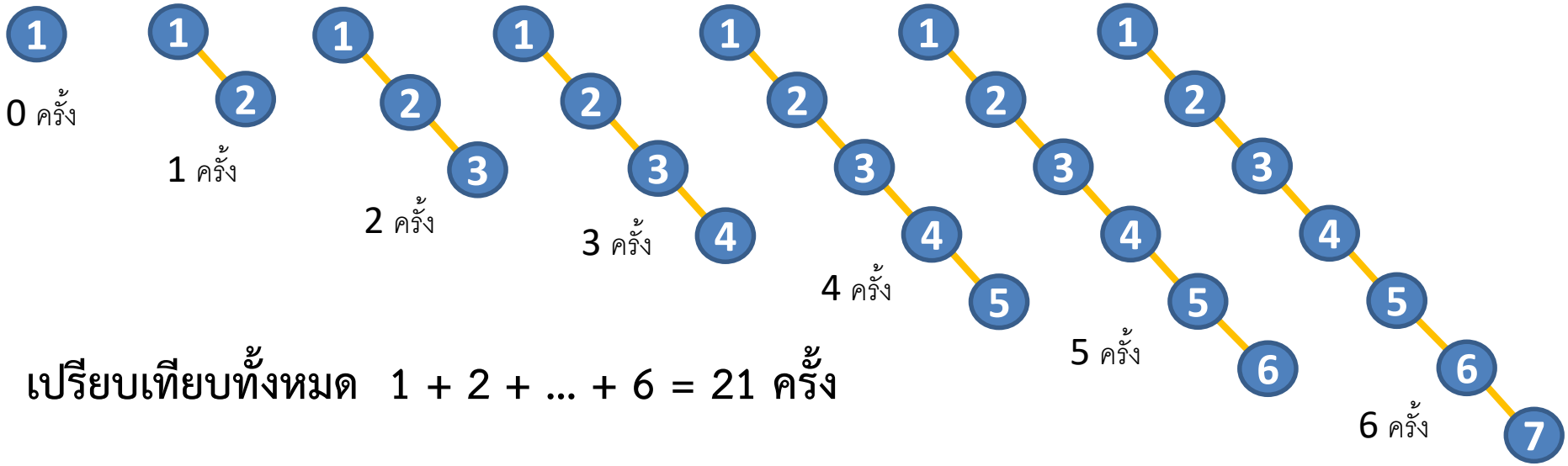
ศูนย์มหาวิทยาลัยศิลปากร

วิเคราะห์การทำงานของ Binary Search Tree

- เราต้องการให้การค้นหา การใส่ข้อมูล การลบข้อมูล มีการเปรียบเทียบตัวเลขให้น้อยครั้งที่สุด

<u>ตัวอย่างที่ไม่ดี</u> ลำดับของข้อมูลที่ใส่เข้าไปในต้นไม้เปล่า 1, 2, 3 ,4, 5, 6, 7

จำนวนการเปรียบเทียบตัวเลข



เปรียบเทียบทั้งหมด 1 + 2 + ... + 6 = 21 ครั้ง

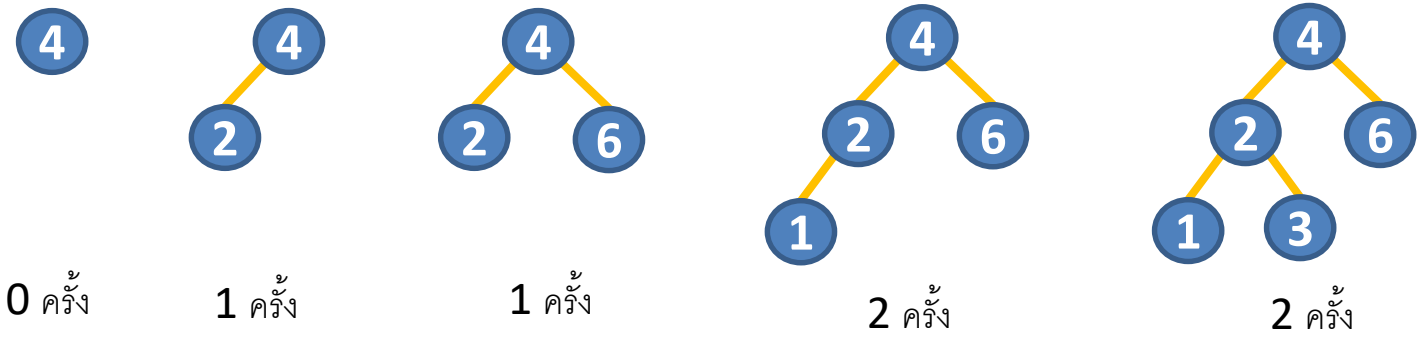
ถ้าตัวเลขมันเรียงกันอยู่แล้ว Binary Search Tree แบบนี้จะทำงานได้ช้ากว่าที่ควรจะเป็นมาก

วิเคราะห์การทำงานของ Binary Search Tree

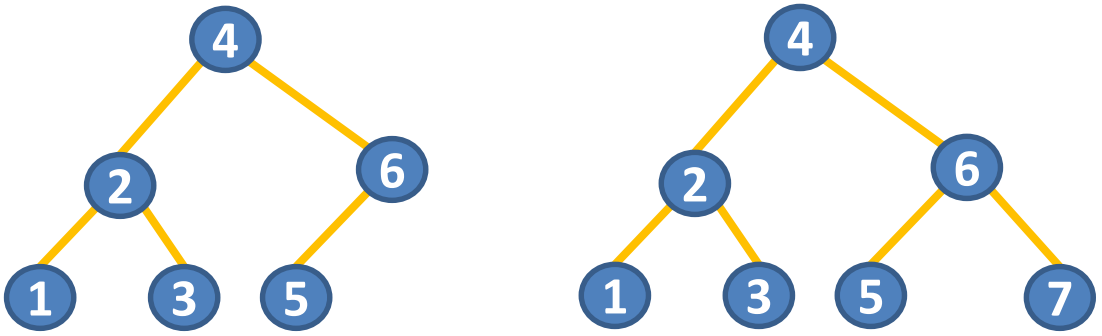
- เราต้องการให้การค้นหา การใส่ข้อมูล การลบข้อมูล มีการเปรียบเทียบตัวเลขให้น้อยครั้งที่สุด

ตัวอย่างที่ดี ลำดับของข้อมูลที่ใส่เข้าไปในต้นไม้เปล่า 4, 2, 6, 1, 3, 5, 7

จำนวนการเปรียบเทียบ



0 ครั้ง 1 ครั้ง 1 ครั้ง 2 ครั้ง 2 ครั้ง



2 ครั้ง 2 ครั้ง

เปรียบเทียบทั้งหมด

1 + 1 + 2 + 2 + 2 + 2

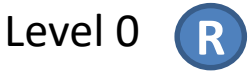
= 10 ครั้ง < 21 ครั้ง

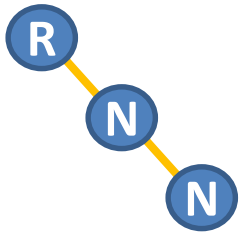
บางทีตัวเลขที่เข้ามาแบบเหมือนสุ่มมาจำทำให้ Binary Search Tree ทำงานได้เร็ว

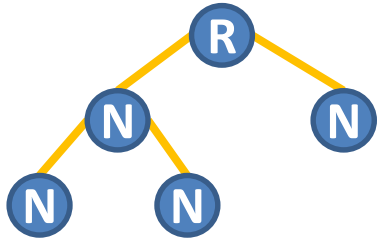
ความลึกของต้นไม้ (Depth of Tree)

- ความลึกของต้นไม้เป็นตัวชี้วัดจำนวนการเปรียบเทียบที่ต้องใช้ในการดำเนินการหลาย ๆ อย่างบนต้นไม้
- ความลึกของต้นไม้วัดจากลำดับชั้น (level) ของลีฟโหนด (leaf node) ที่มากที่สุด
- รากอยู่ที่ลำดับชั้นที่ 0 ดังนั้น ถ้าต้นไม้มีรากแต่เพียงอย่างเดียว ความลึกของต้นไม้ก็คือ 0

ความลึกและการค้นหา







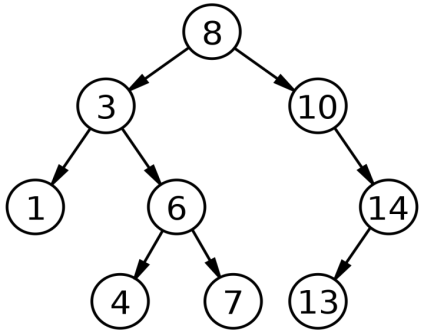
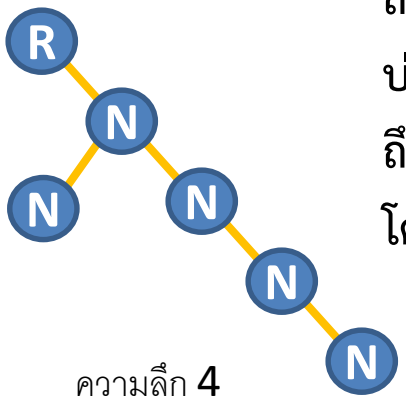


Image source: wikipedia.org



ถ้าต้นไม้มีความลึกมาก กรณีที่โชคร้ายเราต้องเสียเวลาทำการเปรียบเทียบข้อมูลบ่อยครั้ง เช่น ในกรณีของต้นไม้ความลึก 4 ถ้าต้อง findMax ก็จะเสียเวลามาก ถึงแม้ว่า findMin จะทำงานได้อย่างรวดเร็ว

โดยปรกติแล้วเราสนใจเวลาที่ต้องใช้โดยเฉลี่ย หรือเวลาที่ต้องใช้ในกรณีที่แย่ที่สุด

เราสามารถรับประกันได้ว่า binary search tree จะไม่เกิดกรณีที่แย่มาก ๆ หากเราใช้ AVL Tree หรือ Red-Black Tree

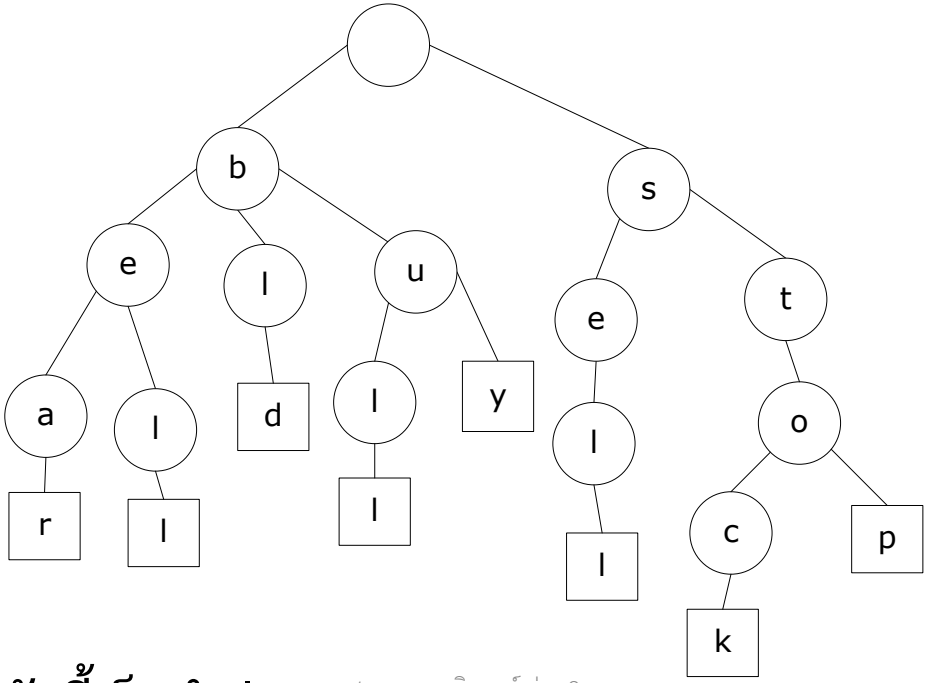
ทรัย (Trie)

- เป็นต้นไม้ที่ออกแบบมาเพื่อเก็บสตริงและค้นหาสตริง
- มาจากคำว่า “retrieval”
- ใช้ในการค้นคืนข่าวสาร เช่นการค้นหาลำดับของ DNA ในฐานข้อมูล genomic
- ถ้าส่วนด้านหน้า (prefix) ของข้อมูลหรือข้อความคือองค์ประกอบหลักในการค้นคืนข้อมูล ทรัยถือได้ว่าเป็นโครงสร้างข้อมูลที่มีความเหมาะสม เช่น ต้องการค้นหาคำทุกคำที่ขึ้นต้นด้วย cat

คุณสมบัติของทรัย

คุณสมบัติของทรัยมีดังนี้ [อ้างอิง: ผศ. นันท์นภัส โตอดีเทพย์]

- ทุกโหนดยกเว้นรากมีเลเบลกำกับเป็นแคแรคเตอร์ในเซตของอักษรทั้งหมด
- ลำดับของโหนดกิ่งเป็นลำดับแบบบัญญัติ (canonical) ของอักษร (หรือออกแบบใหม่)
- โหนดภายนอก (ใบ) แทนสตริง S ที่ได้จากการเชื่อมแคแรคเตอร์จากโหนดราก
- แต่ไม่ใช่โหนดใบก็เก็บสตริงได้เหมือนกัน เช่น กรณีที่มีทั้งคำว่า do, done และ dog อยู่ในทรัย



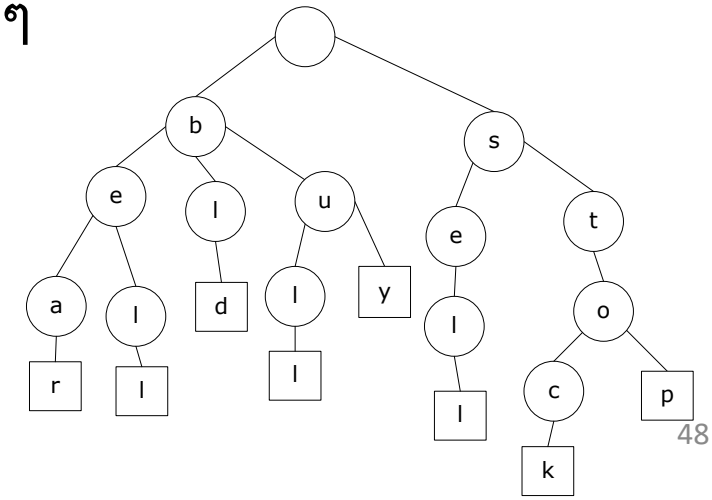
ภาพ: ผศ. นันท์นภัส โตอดีเทพย์

จากภาพข้างบน ทรัยนี้เก็บคำว่า bear, bell, bid, bull, buy, sell, stock, และ stop

ข้อสังเกตเกี่ยวกับทรัย

- ทรัยทำหน้าที่คล้าย search tree ได้เหมือนกัน เช่น การจัดเรียงคำตามแบบบัญญัติ ภาษาอังกฤษเราสามารถเปรียบค่า a, b, c, ..., z เป็น 1, 2, 3, ..., 26 ได้ แล้วกำหนดให้พวกตัวอักษรแรกซ้ายสุด เป็นต้น
- การอ่านค่าของตัวอักษรในแต่ละโหนดแบบ preorder จะทำให้ได้คำที่เก็บไว้ในทรัย (ต้องลบตัวอักษรเวลากลับไปหาโหนดพ่อด้วย)
- การค้นหาคำในทรัยจะเร็วมาก แต่การลบคำจะช้า เพราะต้องตามลบโหนดด้านบนด้วย เช่น การลบคำว่า bear ออกจากทรัยในตัวอย่าง ทั้งโหนด a และ r จะต้องถูกลบด้วย ทรัยจึงไม่เหมาะกับงานที่ต้องลบข้อมูลออกบ่อย ๆ

<u>ทรัยไม่ใช่ binary search tree เพราะสามารถมีโหนดลูกได้มากมายตามจำนวนอักษร</u>



ภาพ: ผศ. นันท์นภัส โตอดิเทพย์

ทดลองสร้างทรัยสำหรับเก็บคำศัพท์

เช่นเดิม เราเริ่มจากที่เก็บข้อมูล

```
#include <string.h>

using namespace std;

class TrieNode {
public:
    bool end;
    char key;
    TrieNode* parent;
    TrieNode* link[26];

    TrieNode(char* word) {
        this->key = word[0];
        end = false;
        parent = NULL;
        for (int i = 0; i < 26; ++i)
            link[i] = NULL;
    };
};
```

ตัวระบุว่ามีคำที่สิ้นสุดในโหนดนี้หรือไม่

แต่เดิมเก็บตัวเลข ตอนนี้เก็บตัวอักษร

ลิงค์ไปโหนดลูกทั้งหมด เนื่องจากตัวอักษรภาษาอังกฤษมี 26 ตัว จึงเตรียมไว้ทั้งหมด 26 ชุด

เก็บตัวอักษรตัวแรกของคำไว้ในฐานะ key ของโหนด

คล้ายกับการกำหนดค่าเริ่มต้นleft กับ right ซึ่งเป็นลิงค์ไปหาโหนดลูกใน binary search tree

แล้วตัวอักษรสัมพันธ์กับตัวเลขยังไง

วิธีเก็บตัวอักษรมีอยู่หลายมาตรฐาน เช่น ASCII, ISO 8859-11, และ Unicode

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

เราจะเลือกแบบ ASCII เพราะง่ายและเข้ากันได้กับหลายระบบ

Source: www.LookupTables.com

แค่จะหาลิงค์ไปโหนดลูกก็เป็นเรื่องที่ต้องคิด


```
int getSlot(char key) {
    if (key >= 97) // อักษรตัวเล็กเริ่มจากค่า 97 ขึ้นไป
        // convert to uppercase.
        key -= 32;

    if (key < 65 || key > 90)
        return -1;  // invalid slo
    else
        return (key - 65);
}
```

อักษรตัวใหญ่จะมีค่าน้อยกว่าตัวเล็กอยู่ 32

อักษรตัวใหญ่จะมีค่าตั้งแต่ 65 ถึง 90

แปลงค่าให้กลายเป็น 0 - 25

ถึงคราวต้อง insert ข้อมูลกันแล้ว

แบ่งออกเป็นสองฟังก์ชันเพราะโหนดรากของทรัยไม่ได้เก็บตัวอักษรอะไรไว้และทำหน้าที่พิเศษ

ตัวจัดการโหนดราก

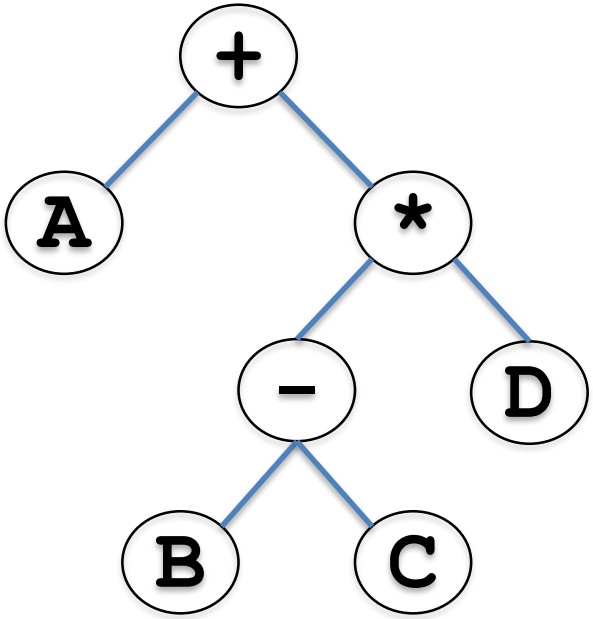
```
void insert(char* word,
TrieNode*& root) {
    if (root == NULL)
        root = new
TrieNode("*");

    int slot = getSlot(word[0]);
    if (slot == -1)
        return;

    insert2(word, root-
>link[slot], root);
}
```

```
void insert2(char* word, TrieNode*& subtree, TrieNode* parent)
{
  char key = word[0];
  if (key == '\0') {
    parent->end = true;
  } else {
    int slot = getSlot(word[0]);
    if (slot == -1)
      return;      // invalid character
    else {

      // need new node
      if (subtree == NULL) {
        subtree = new TrieNode(word);
        subtree->parent = parent;
      }

      if (word[1] == '\0') {
        subtree->end = true;
      } else {
        int nextSlot = getSlot(word[1]);
        insert2(word+1, subtree->link[nextSlot], subtree);
      }
    }
  }
}
```

เทคนิคการบวกลบ pointer ไม่ค่อยมีในภาษายุคใหม่เพราะทำให้โปรแกรมมีช่องโหว่ด้านความปลอดภัยได้ง่าย

การประยุกต์ต้นไม้กับนิพจน์การคำนวณ

- ให้เครื่องหมายการคำนวณ (Operator) เป็นโหนดพ่อของโอเปอร์แรนด์ทั้งสองที่เป็นลูกทางซ้ายและลูกทางขวา [อ้างอิง: ผศ. นันท์นภัส โตอดีเทพย์] จากรูป แทนนิพจน์ A+(B-C)*D ด้วยต้นไม้

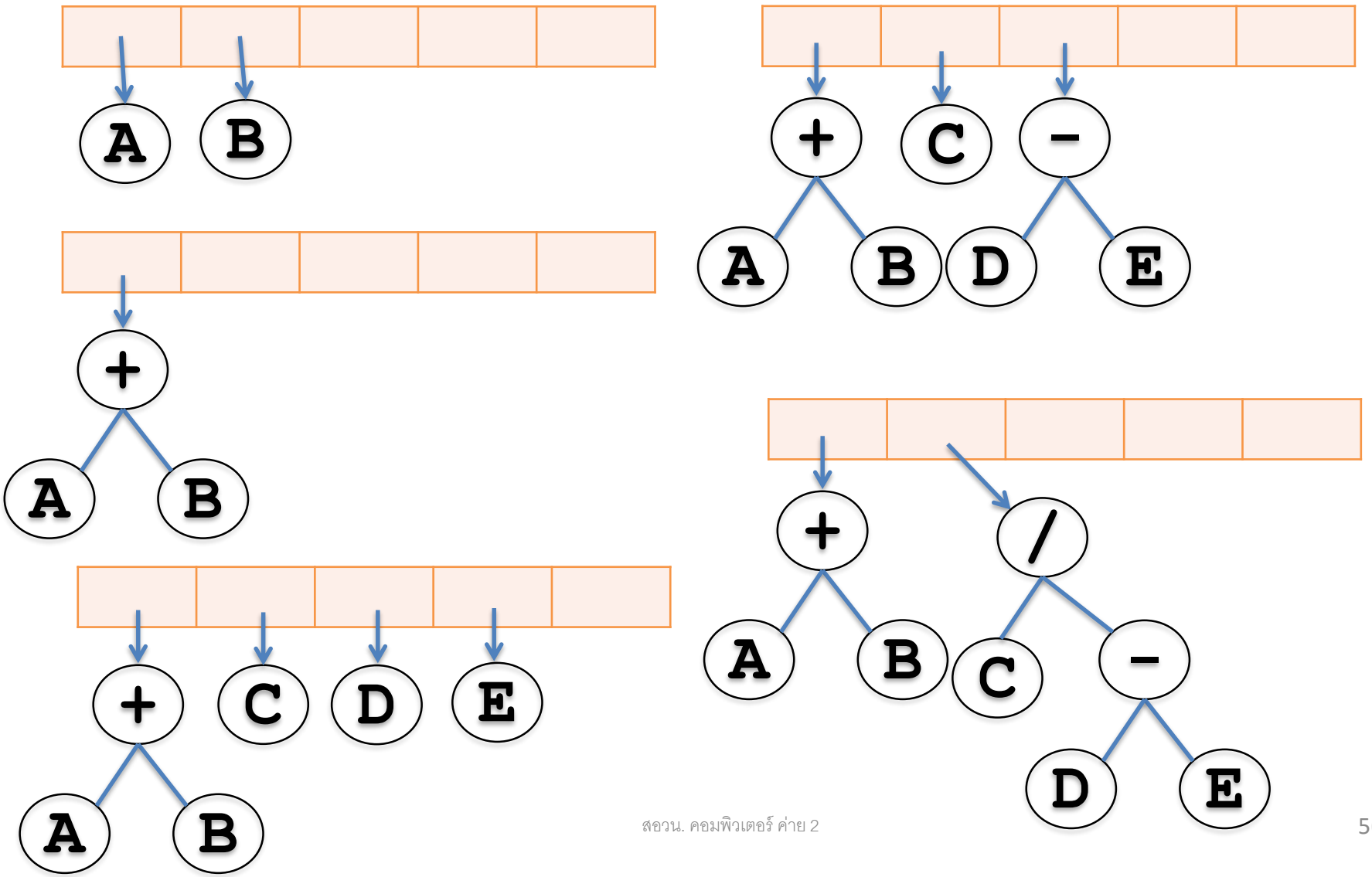


ซึ่งเราสามารถแปลงนิพจน์ infix notation เป็น postfix notation โดยการแวะผ่านต้นไม้แบบ inorder

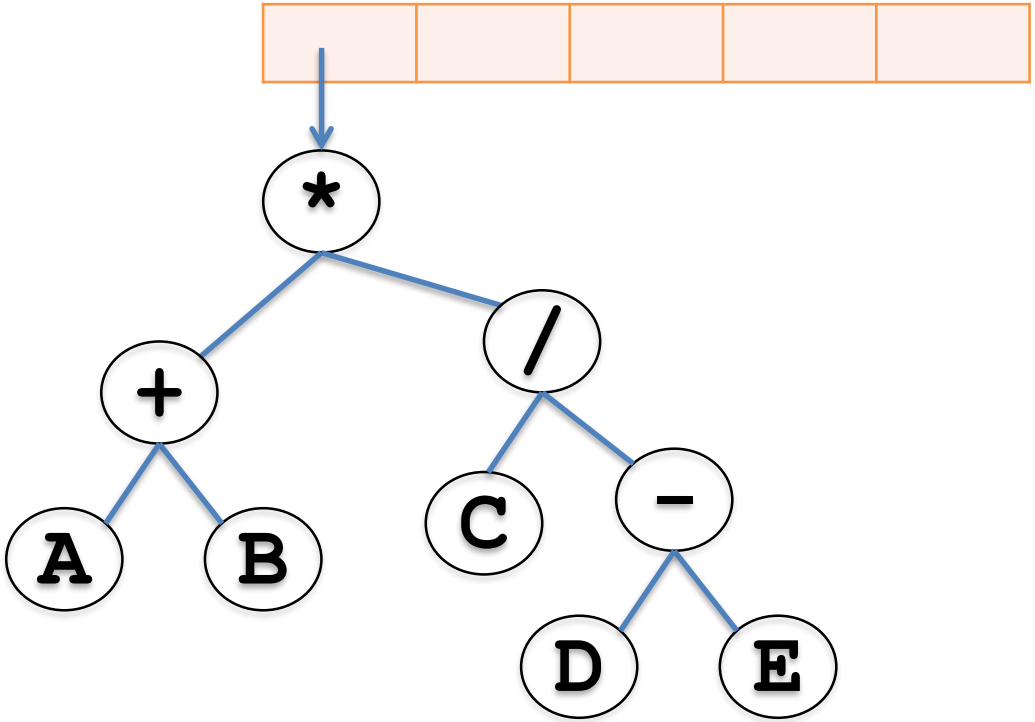
การสร้างต้นไม้เก็บนิพจน์จากนิพจน์ postfix

- อ่านนิพจน์ postfix ครั้งละ 1 สัญญลักษณ์จนหมด
  - ถ้าเป็นโอเปอร์แรนด์ สร้างโหนด และพุชค่าพอยเตอร์ของโหนดนั้นลงสแตค
  - ถ้าเป็นโอเปอร์เรเตอร์ ให้สร้างต้นไม้โดยมีโอเปอร์เรเตอร์เป็นโหนดแรกและต้นไม้ย่อยขวาและซ้ายได้จากพอยเตอร์ที่ป๊อปจากแสตคและพุชพอยเตอร์ของต้นไม้ใหม่ลงสแตค

postfix expression: A B + C D E - / *



postfix expression: A B + C D E - / *



เรื่องน่ารู้

- ในหลาย ๆ ปัญหาทรีของเรามักจะไม่ต้องยุ่งกับการลบโหนด แต่เราก็ใช้การลบโหนดเพื่อช่วยในการอัพเดตคีย์ได้
- ต้นไม้ที่สมดุลอย่าง AVL กับ Red-Black Tree รับประกันความเร็วในการทำงาน

  แต่ก็ไม่เหมาะที่จะเอาไปใช้ในการแข่ง เพราะใช้เวลาสร้างนาน

  อย่างไรก็ตามต้นไม้สองแบบนี้อาจจะเหมาะในการใช้งานจริง
- การรู้ข้อกำหนดของโจทย์ เช่น “จะมีข้อมูลเข้าไม่เกิน 1000 บรรทัด” จะทำให้เราสามารถสร้างที่เก็บข้อมูลแบบตายตัว เช่น Titem item[1000]; ขึ้นมาได้ โดยไม่ต้องกังวลว่าจะต้องไปหาขนาดของจำนวนข้อมูลก่อน

  ➔ โปรแกรมจะเขียนง่าย เหมาะกับการแข่งที่มีเวลาน้อยอย่างโอลิมปิกวิชาการ

Self-balancing Binary Search Tree

- โครงสร้างที่การันตีความสูงของต้นไม้จะเป็น O(log2 *n*) แม้จะมีการปรับเปลี่ยนข้อมูลในโครงสร้างแบบไดนามิก
- ตัวอย่างของโครงสร้างแบบนี้เช่น
  - AVL Trees
  - B-trees
  - Red-black Trees

  เป็นต้น

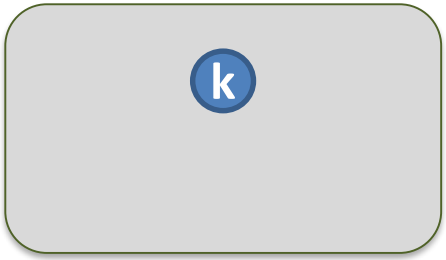
AVL Trees

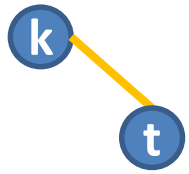
- ถึง AVL Trees ไม่เหมาะกับการแข่งแต่เหมาะสำหรับทำแอปริเคชัน
- เพื่อให้การค้นหาใช้เวลาน้อยที่สุด เราจำเป็นต้องทำให้โครงสร้างต้นไม้มีความสมดุลด้านความสูงให้มากที่สุด ซึ่งมีนักคณิตศาสตร์ชาวรัสเซียสองคนคือ G.M. ADEL'SON-VEL'SKII และ E.M. LANDIS ได้คิดค้นรูปแบบไว้แล้ว ซึ่งวิธีการก็ได้ให้ชื่อเพื่อเป็นเกียรติกับพวกเขาคือ AVL Trees
- AVL Trees จะใช้เวลาในการค้นหา แทรกข้อมูล และลบข้อมูลในต้นไม้ขนาด n โหนด เสร็จในเวลา O(log n) เท่านั้น แม้แต่ในกรณีที่แย่ที่สุดก็ตาม

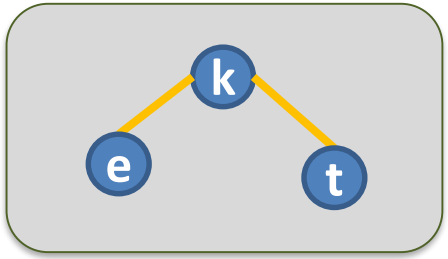
AVL Trees (ต่อ)

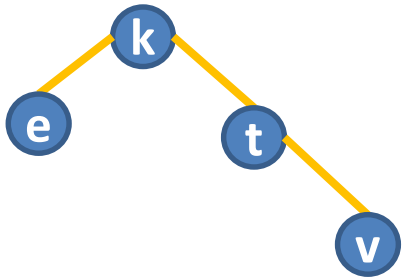
- AVL Trees คือ binary search tree ที่มีความสูงของต้นไม้ย่อยทางขวาและต้นไม้ย่อยทางซ้ายมีความแตกต่างไม่เกิน 1 และต้นไม้ย่อยทั้งซ้ายและขวาต้องเป็นโครงสร้าง AVL Trees เช่นกัน
- การดำเนินการกับโครงสร้าง AVL Trees จะต้องทำให้คงไว้ซึ่งโครงสร้าง AVL Trees เช่นเดิม ดังนั้นเราต้องมาพิจารณาว่าทำสิ่งต่อไปนี้อย่างไร
  - Insertion of a node
  - Deletion of a node

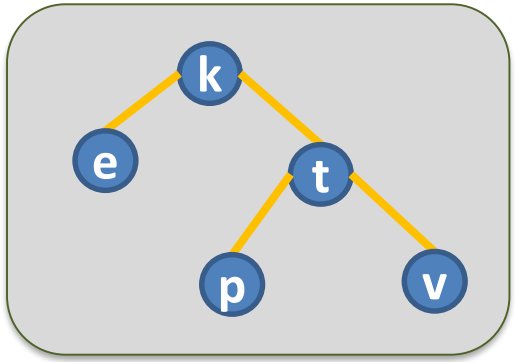
Insertion of a node

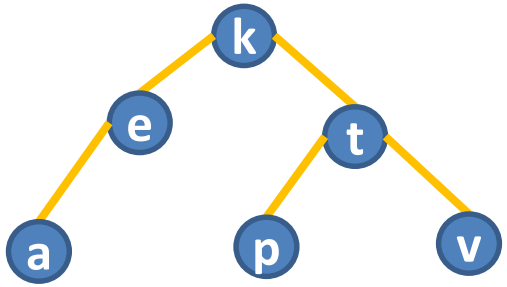


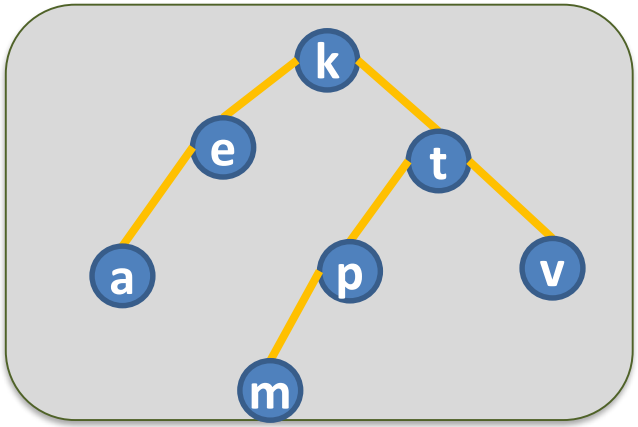


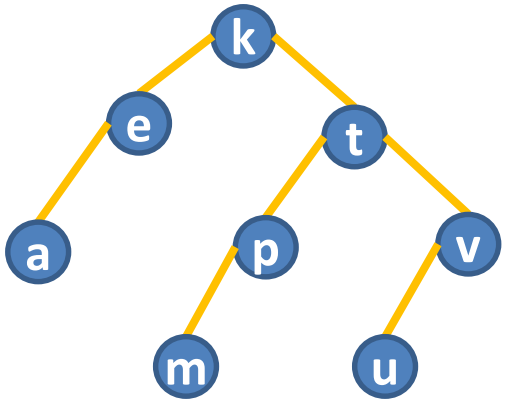


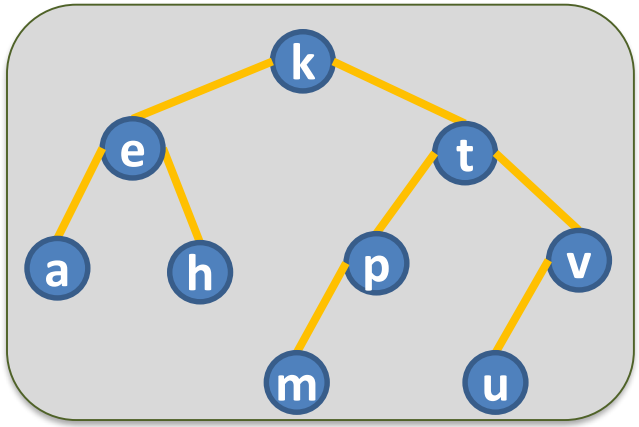










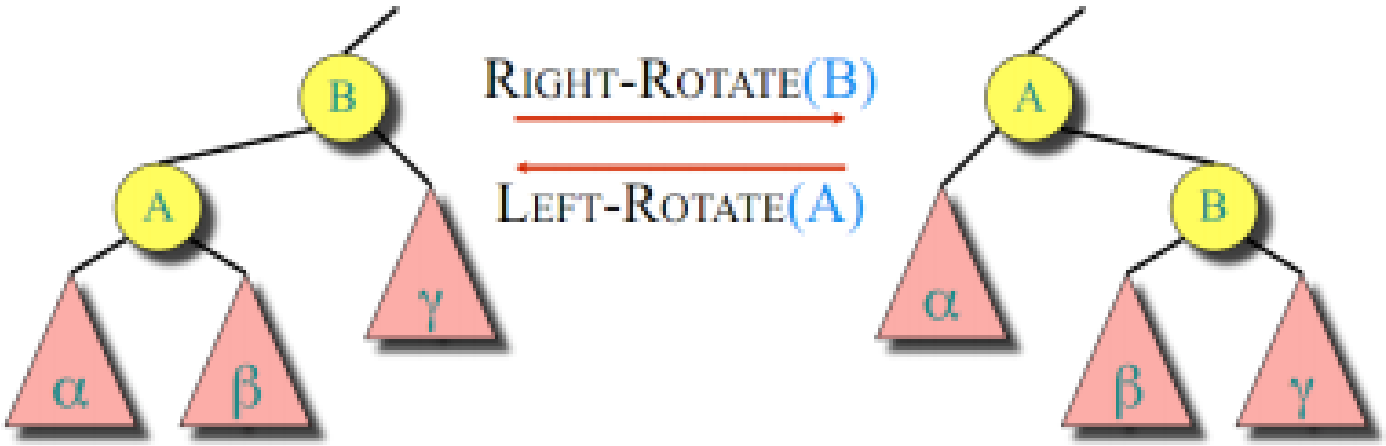


การดำเนินงาน 2 อย่างที่ช่วยรักษาสมดุล

- การหมุนเวียน (rotation) เป็นการจัดลำดับโหนดใหม่เพื่อให้ความสูงของต้นไม้ของต้นไม้ย่อยเปลี่ยนตำแหน่ง ทำให้รักษาสมดุลได้
- Rotation ต้อนการเปลี่ยนตำแหน่งแค่ left, right และ parent ของบางโหนดเท่านั้น
- มี Rotation ที่น่าสนใจ 2 คือ Left rotation และ Right rotation

Rotation

ภาพแสดงการดำเนินงาน Rotation ทั้ง 2 แบบ

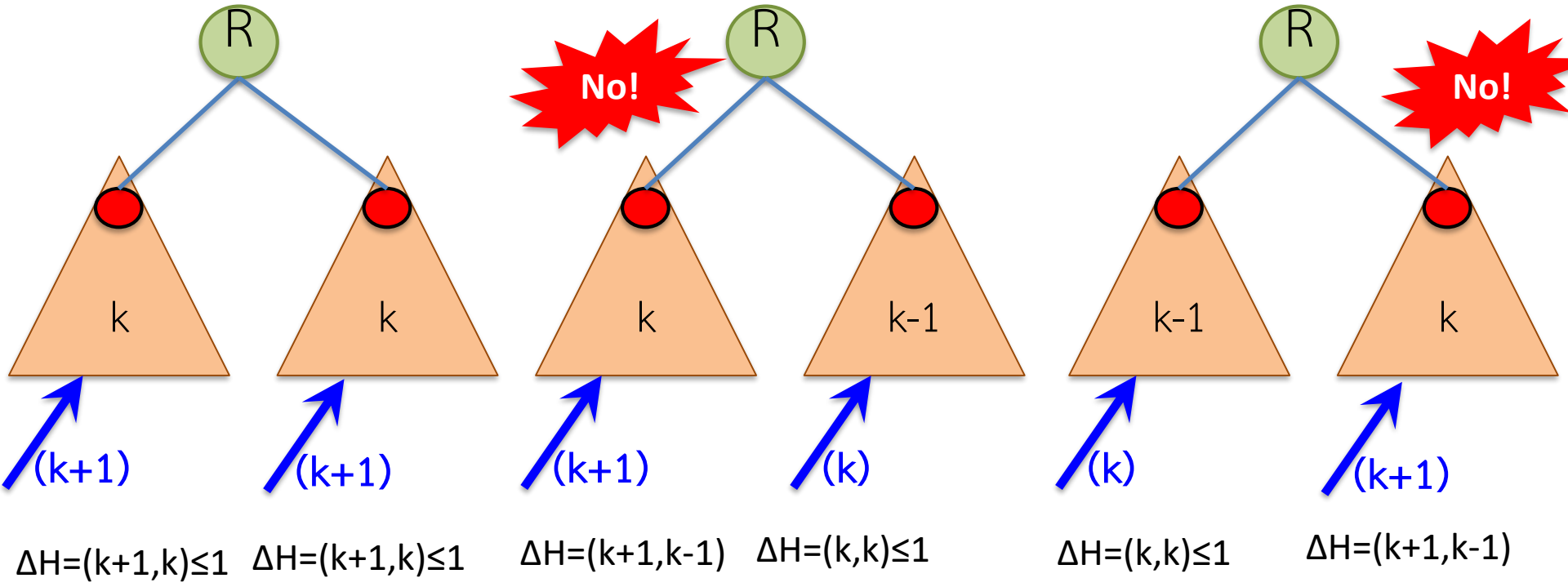


ที่มาภาพ : http://courses.csail.mit.edu/6.006/spring11/

Right_Rotate (B) กลับไปเป็นลูกทางขวา (ต้องไปอยู่ใต้ลูกทางซ้าย)

Left_Rotate(A) กลับไปเป็นลูกทางซ้าย (ต้องไปอยู่ใต้ลูกทางขวา)

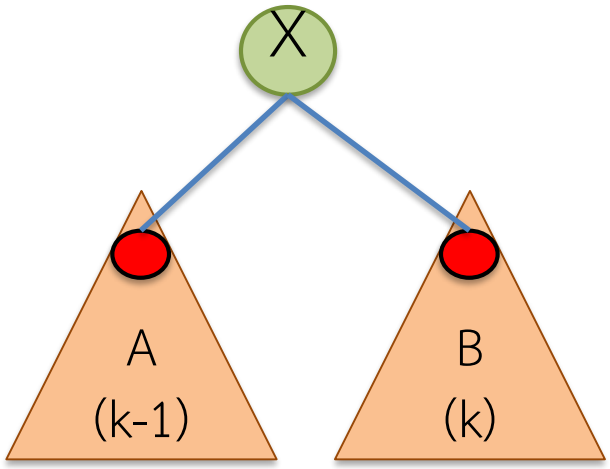
AVL Tree ก่อน insert ในสถานการณ์ต่างๆ



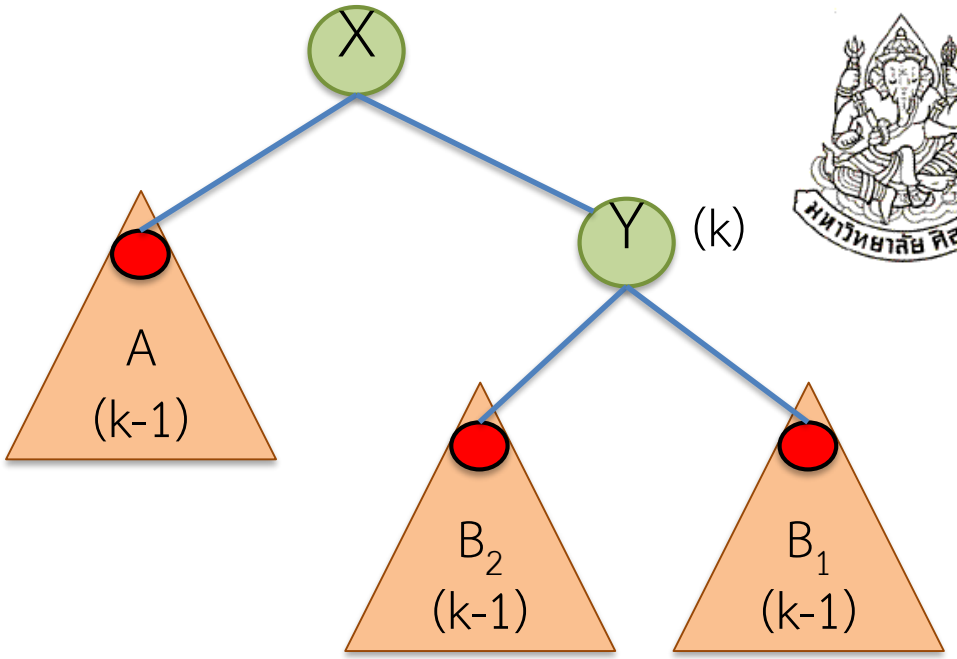
ข้อสังเกตุ ถึงแม้ว่าต้นไม้ย่อยจะมีความสูง k แต่เมื่อเราแทรกโหนดใหม่ ไม่จำเป็นว่าความสูงของต้นไม้ย่อยจะต้องเปลี่ยนไปเป็น k+1 เสมอไป

มีความเป็นไปได้ 3 กรณี

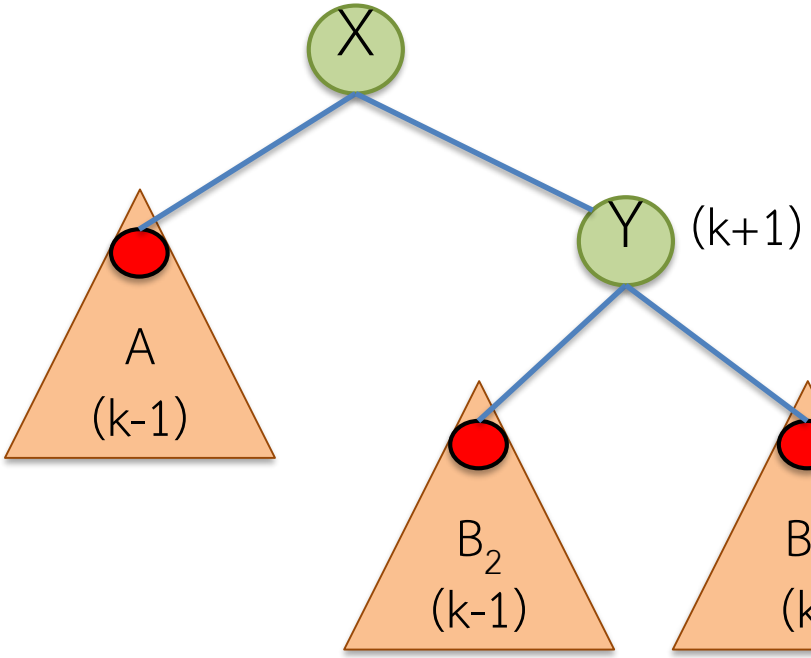
- โหนดลูกของ R มีความสูงเท่ากันทั้งสองด้าน ในกรณีนี้ ไม่ว่าโหนดใหม่จะถูกแทรกไว้ที่ด้านใดก็ไม่ทำให้ความสูงของต้นไม้ย่อยทั้งสองต่างกันเกินกว่า 1 ไปได้
- โหนดลูกของ R ด้านซ้ายมีความสูงมากกว่าโหนดลูกของ R ด้านขวาอยู่ 1 ถ้าโหนดใหม่ถูกเพิ่มที่ด้านซ้ายจะทำให้ต้นไม้ไม่สมดูล
- โหนดลูกของ R ด้านขวามีความสูงมากกว่าโหนดลูกของ R ด้านซ้ายอยู่ 1 ถ้าโหนดใหม่ถูกเพิ่มที่ด้านขวาจะทำให้ต้นไม้ไม่สมดูล
- เมื่อใดก็ตามที่ AVL Tree ไม่สมดุล เราจะต้องทำการ rotation เพื่อจัดโหนดให้ต้นไม้เกิดความสมดุล



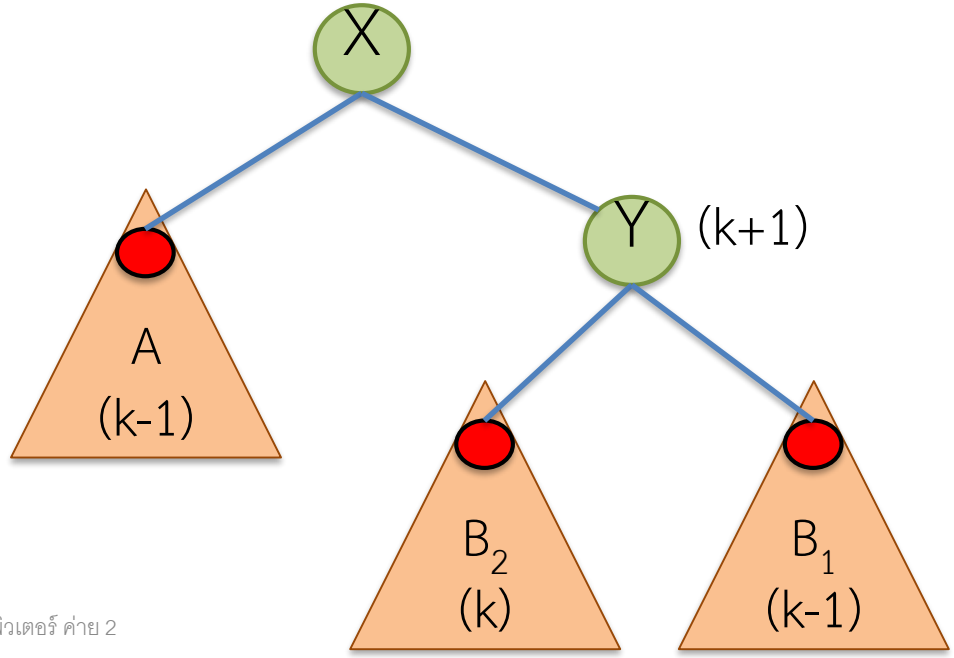
≅



CASE 1 : โหนดใหม่แทรกที่ B_1

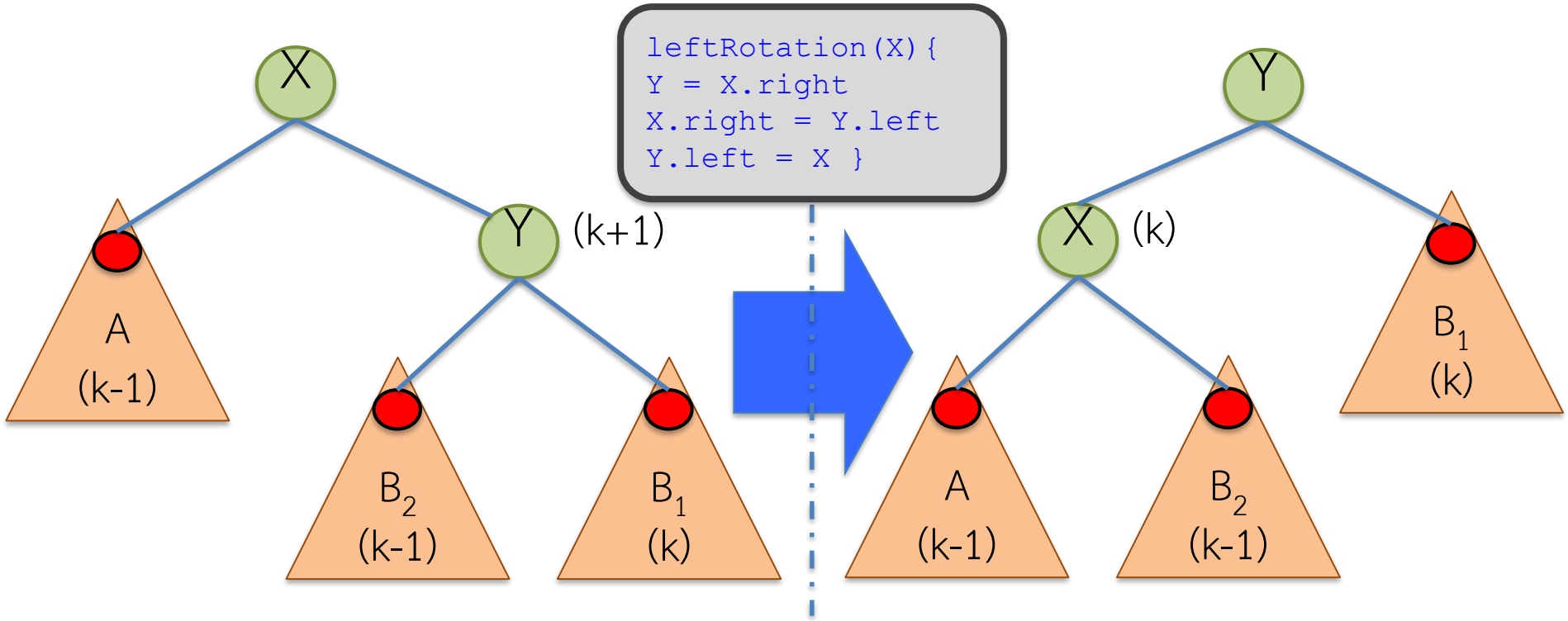


CASE 2 : โหนดใหม่แทรกที่ B_2



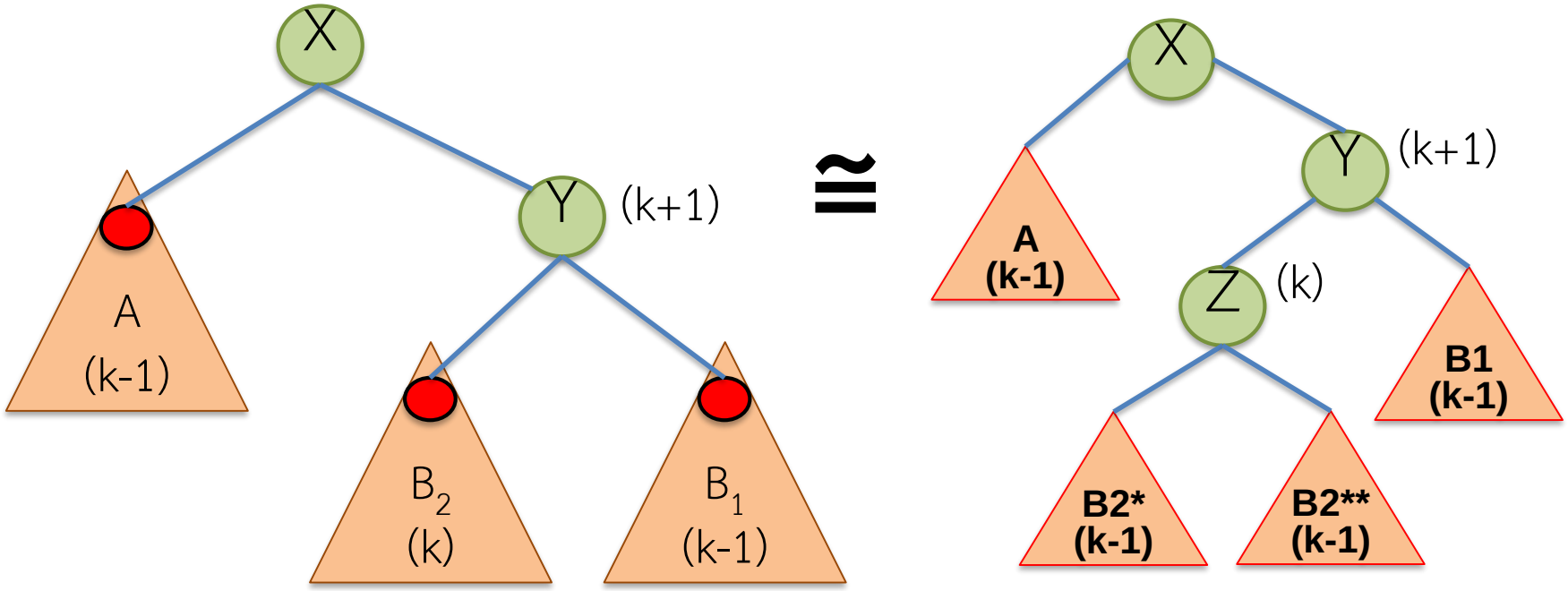
Case 1: โหนดใหม่ถูกแทรกที่ต้นไม้ย่อยขวาสุด

- สามารถใช้ left rotation เพื่อช่วยจัดลำดับในต้นไม้ใหม่

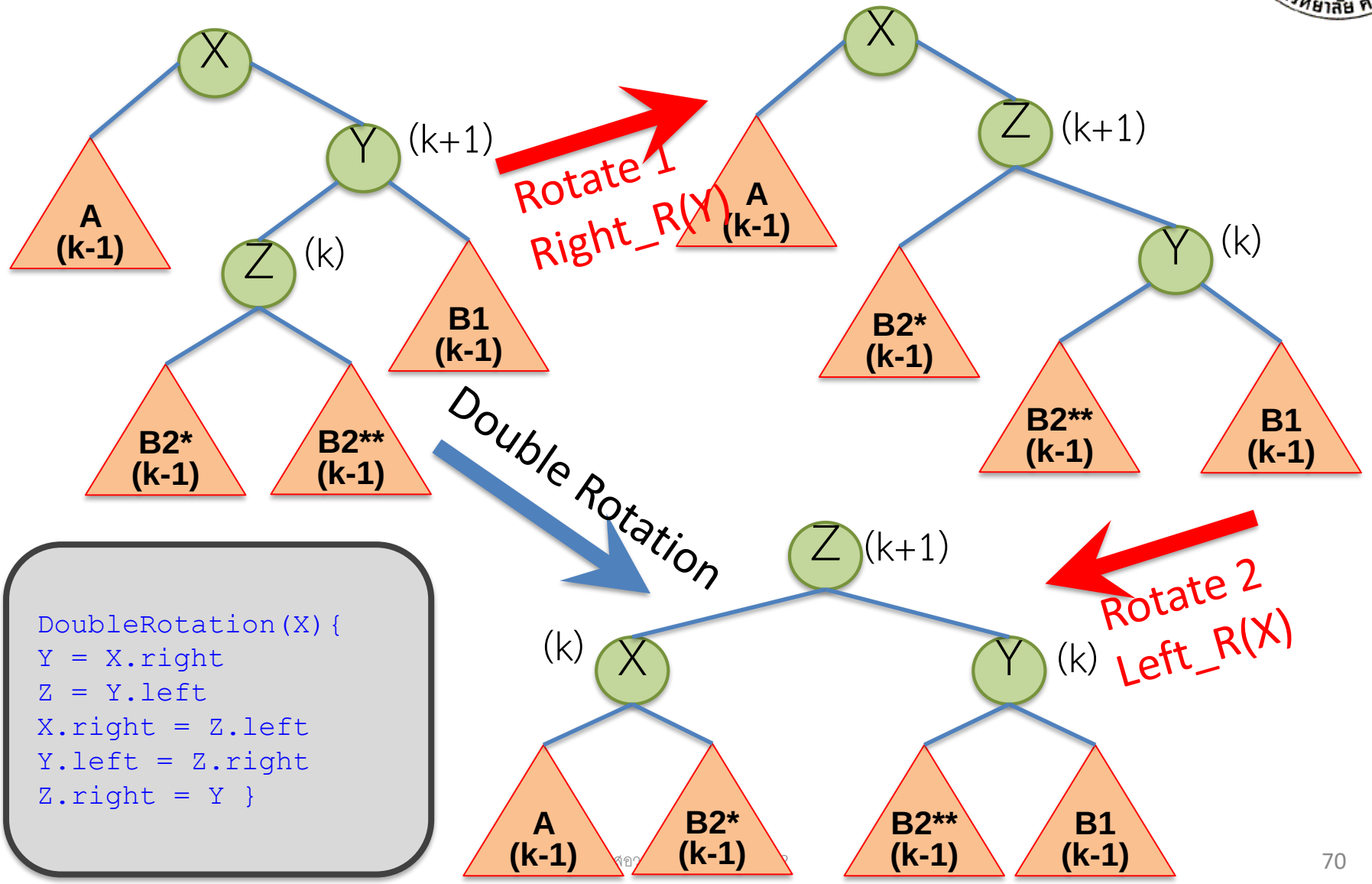


ลำดับของสมาชิกยังคงเหมือนเดิมคือ $A < X < B_2 < Y < B_1$

CASE 2:โหนดใหม่ถูกแทรกที่ต้นไม้ย่อยซ้ายของ Y



Case 2: ต้องใช้ double ratation

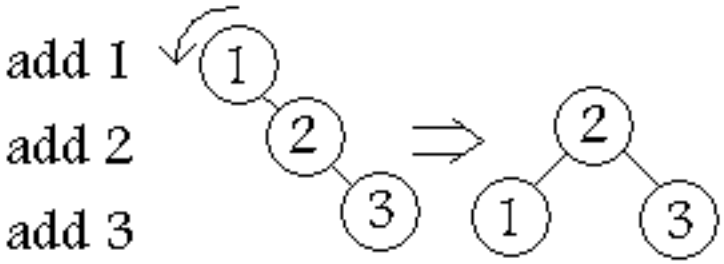



```
DoubleRotation(X){
Y = X.right
Z = Y.left
X.right = Z.left
Y.left = Z.right
Z.right = Y }
```

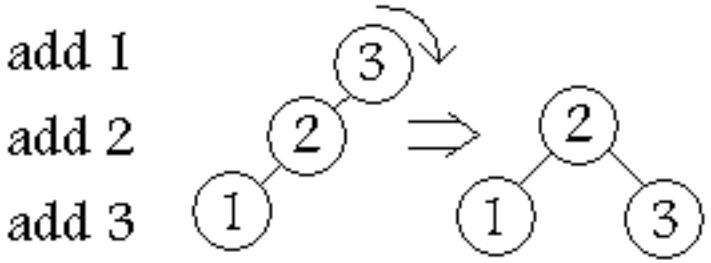
การลบโหนดออกจาก AVL Tree

- การลบโหนดใดๆ ออกจาก AVL Tree สามารถใช้วิธีการลบโหนดเหมือน BST แต่ต้องมีการทำสมดุลให้ต้นไม้ โดยใช้วิธีการ rotation ซึ่งสิ่งที่ต้องคำนึงถึงคือเมื่อเราทำสมดุลให้กับโหนดใดแล้วโหนดพ่อของโหนดดังกล่าวอาจเกิดความไม่สมดุลได้ เราจึงต้องตามไปปรับสมดุลด้วย left rotation หรือ double rotation ไปเรื่อยๆ ตามแต่สถานการณ์ว่าต้นไม้ย่อยด้านใดมีความสูงมากกว่ากัน ซึ่งก็ใช้วิธีการเหมือนกับการเพิ่มโหนดนั่นเอง

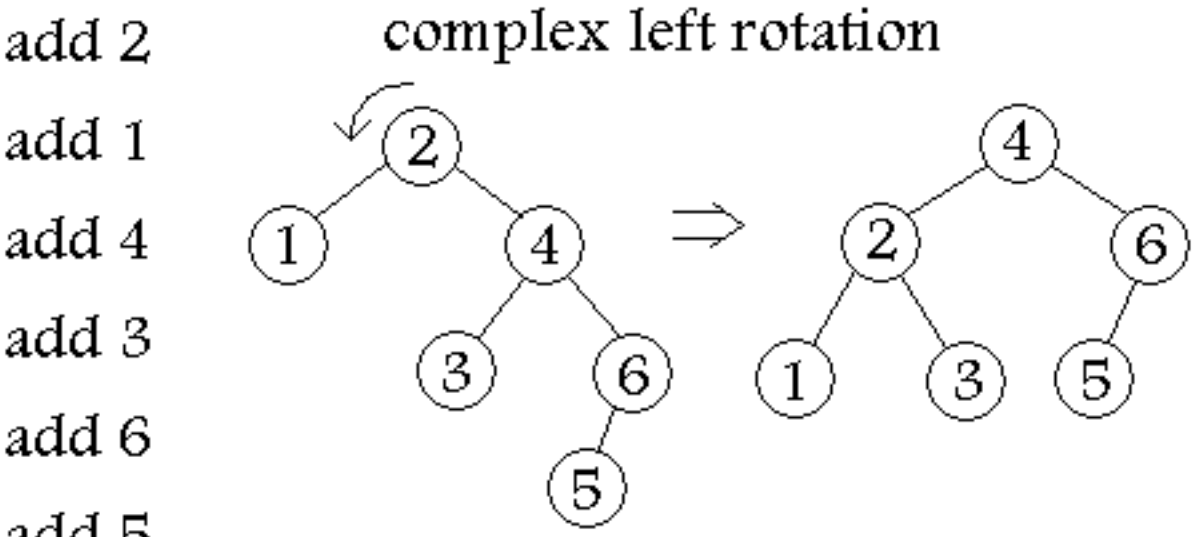
Single Rotations



simple left rotation

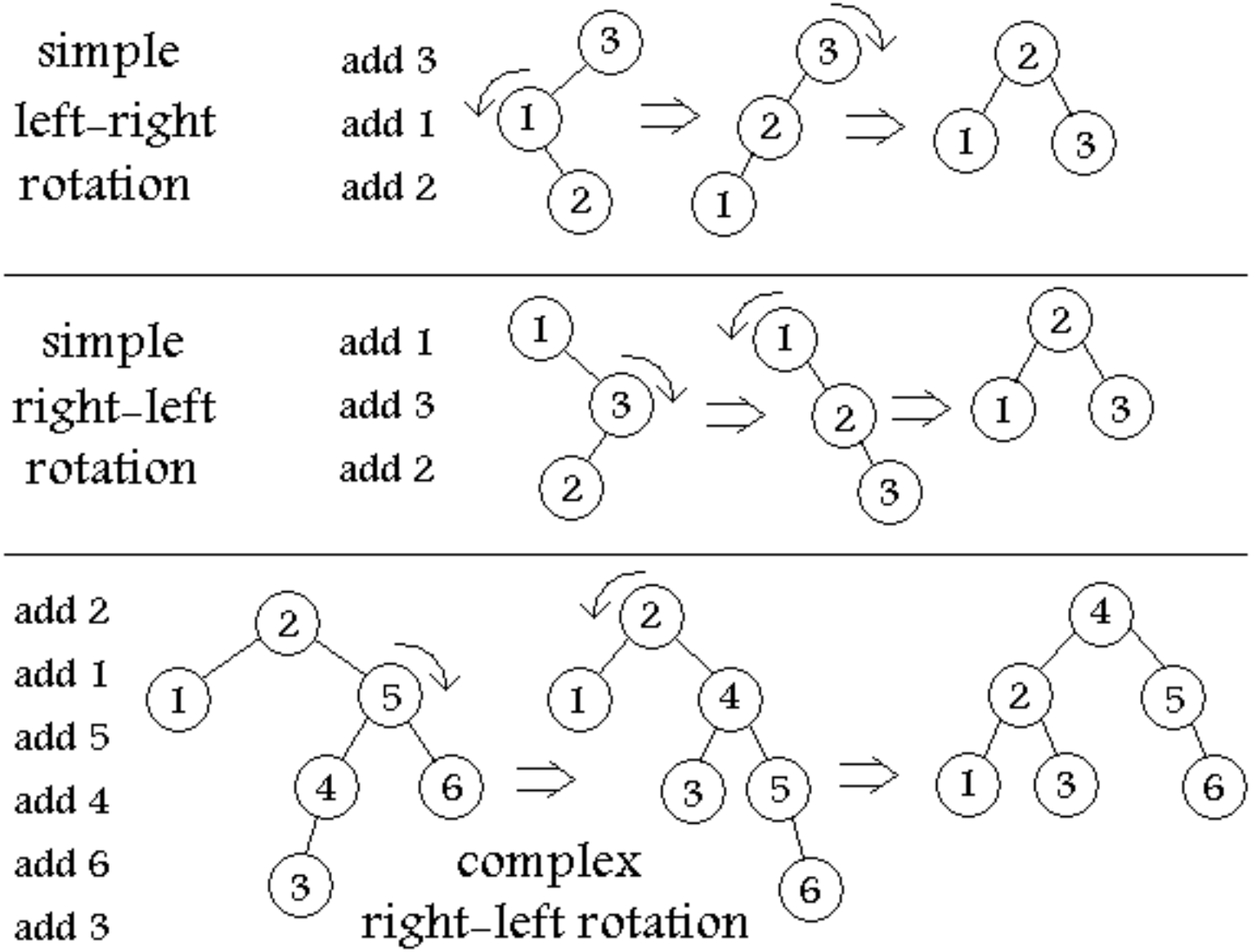


simple right rotation



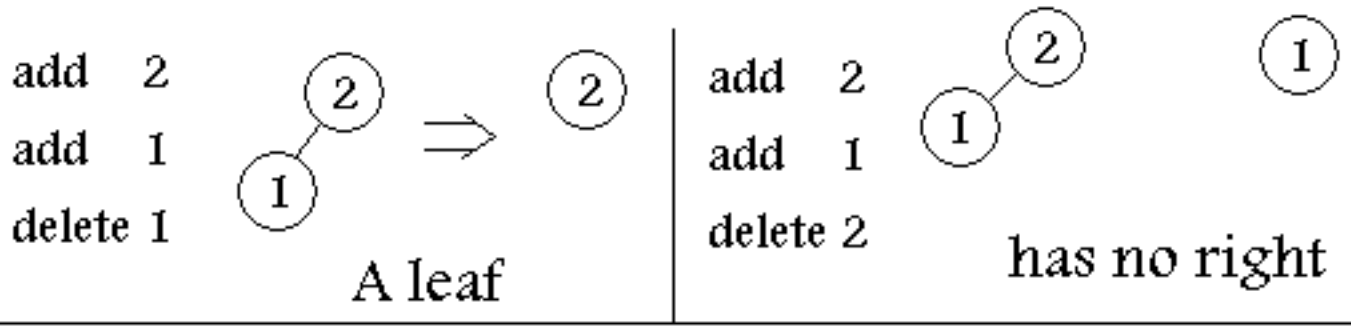
ที่มาภาพ : http://www.coe.utah.edu/~clillywh/CS2420

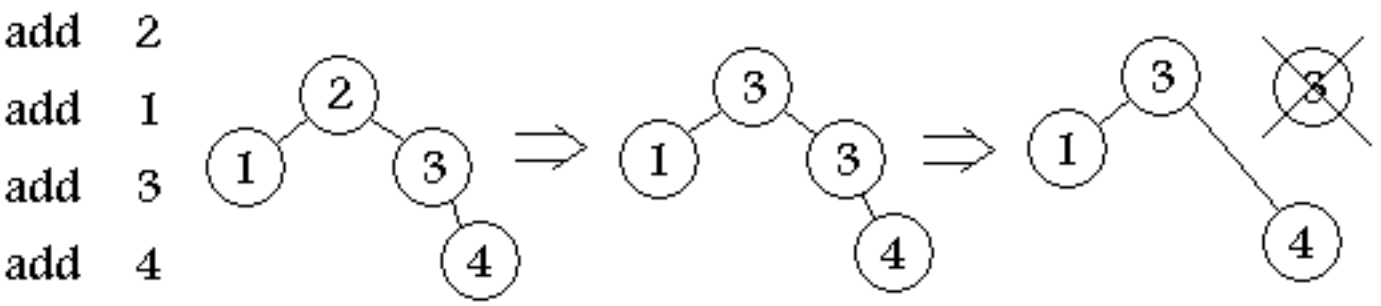
Double Rotations

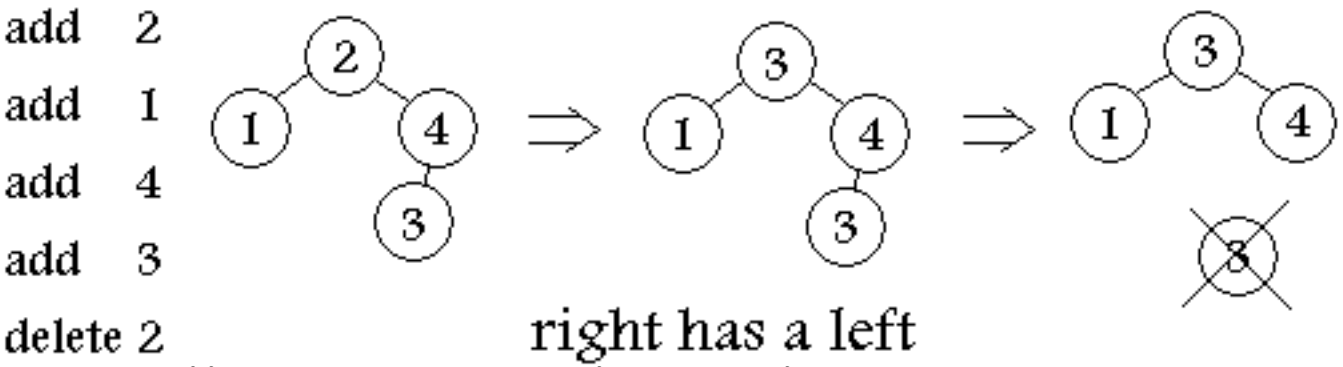


ที่มาภาพ : http://www.coe.utah.edu/~clillywh/CS2420

Simplified Deletion







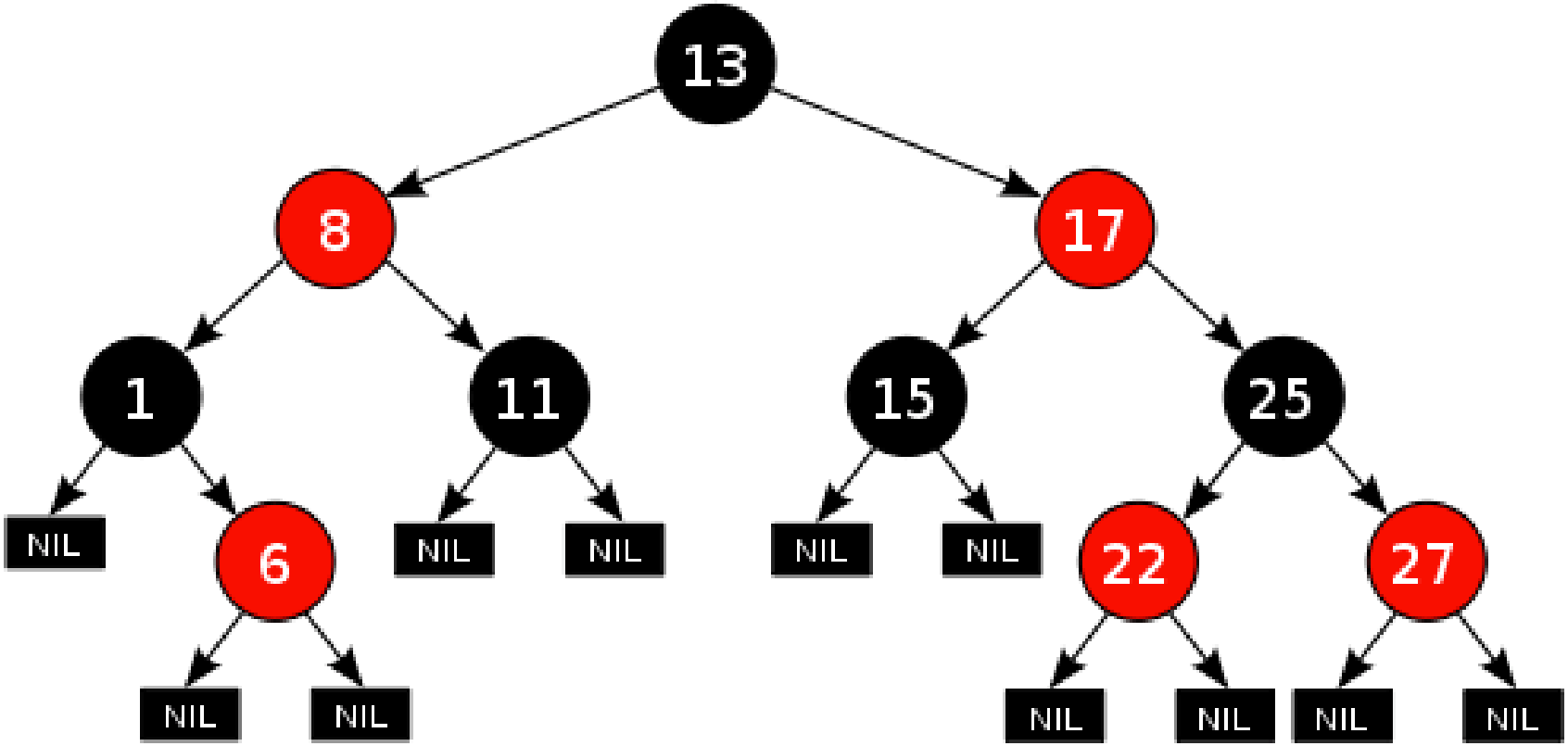
ที่มาภาพ : http://www.coe.utah.edu/~clillywh/CS2420

Red-black trees

- ต้องเพิ่มฟิล์ดพิเศษลงในโหนดนั่นคือสี โดยกำกับให้แต่ละโหนดต้องมีสี ซึ่งจะเป็นได้แค่ 2 สีคือสีดำ (black) หรือสีแดง (red)
- ต่อไปนี้คือคุณสมบัติของ Red-black trees

1. ทุกโหนดจะต้องมีสีแดงหรือสีดำเท่านั้น
2. รากและใบ (คือค่า NULL เพราะถือเป็นโหนดสิ้นสุดจริงๆ) มีสีดำ
3. ถ้าโหนดใดเป็นสีแดง โหนดลูกจะต้องเป็นสีดำทั้งสองโหนด
4. เส้นทางใดๆ (path) จากโหนด x ไปโหนดลูกที่เป็นใบของมันจะมีจำนวนโหนดสีดำเท่ากันเสมอ เรียก black-height(x)

ตัวอย่าง Red-black Trees



Trees in STL

- น่าเสียดาย The Standard Template Library ไม่ได้มีเทมเพลตต้นไม้ในชื่อว่า tree แต่มี container ที่เมื่อใส่ข้อมูลไป ตัวเทมเพลตได้สร้างการเก็บข้อมูลเป็นแบบ self-balancing binary search tree และเนื่องจาก the self-balancing BST จะรักษาสมดุลของต้นไม้ ดังนั้นจึงมั่นใจได้ว่าเวลาในการค้นหาจะมีค่า $O(\log_2 n)$ เสมอ แม้จะมีการเปลี่ยนโครงสร้างข้อมูลภายในก็ตาม
- ตัวคอนเทนเนอร์ที่ว่าคือ map<T1,T2>

std::map<T1,T2> container

- The STL map<T1,T2> บางครั้งถูกเรียกว่า associative array เพราะถูกออกแบบมาให้ทำการแมบค่าจะชนิดข้อมูล T1 ไปยังชนิดข้อมูล T2
- เริ่มจากอาร์เรย์ธรรมดา ซึ่งจริงๆ แล้วเป็นการแมปค่าจะจำนวนเต็มไปยังชนิดข้อมูลที่กำหนด ตัวอย่างเช่น

```
string fruits[3] = {"Apple", "Orange", "Banana"};
```

เมื่อเราต้องการอ้างถึงผลไม้ก็ใช้ตัวเลขเป็นตัวชี้ ดังนี้

```
cout << fruits[0];
cout << fruits[1];
```

0	1	2
"Apple"	"Orange"	"Banana"

std::map<T1,T2> Container (2)

- จากอาร์เรย์ เราสามารถอ้างถึงชื่อผลไม้จากตัวเลข

```
string fruits[3] = {"Apple", "Orange", "Banana"};
```

0	1	2
"Apple"	"Orange"	"Banana"

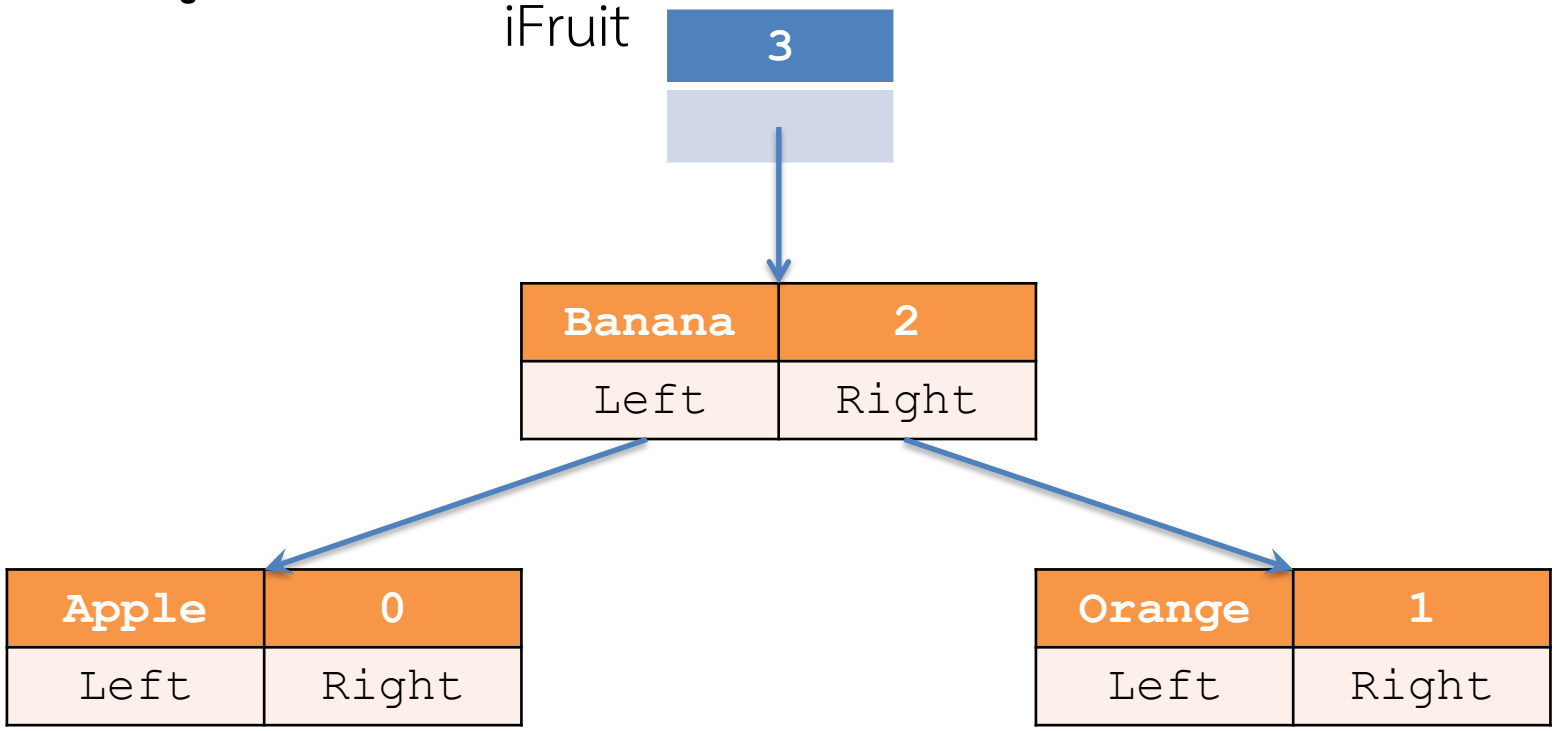
- แต่ถ้าเราต้องการใช้ชื่อผลไม้อ้างตัวเลขหล่ะ อาร์เรย์ทำไม่ได้ แต่ map<T1,T2> สามารถทำได้ โดย

```
map<string, int> iFruit;
iFruit["Apple"] = 0;
iFruit["Orange"] = 1;
iFruit["Banana"] = 2;
```

["Apple"]	["Orange"]	["Banana"]
0	1	2

std::map<T1,T2> container

- ซึ่งความจริงแล้วโครงสร้างภายในของ iFruit จะเป็น self-balancing BST ดังรูป



std::map<T1,T2> & Operator

ตัวดำเนินการ	ความหมาย
bool empty()	เป็นจริงเมื่อไม่มีค่า
Int size()	จำนวนค่าที่มีใน map
Int erase(T1 aValue)	ลบค่า aValue
Void clear()	ลบค่าทุกค่าใน map
Iterator find (T1 aValue)	คือค่าที่คู่กับ aValue ทุกค่า
Int count(T1 aValue)	นับค่าชนิดข้อมูล T1 มีค่า aValue
Iterator begin()	คือค่าแรกของ map
Iterator end()	คือพอยเตอร์ของค่าสุดท้ายของ map

ตัวอย่างโปรแกรม


```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {
    map <string, string> mascots;
    mascots[“China”] = “Panda”;
    mascots[“Thailand”] = “Elephant”;
    mascots[“Malaysia”] = “Tiger”;
    cout << “enter the name of country:”;
    string country;
    getline(cin, country);
    map<string, string>::iterator it = mascots.find(country);
    if (it != mascots.end())
       cout << “Answer ” << mascots[country] << endl;
   else
       cout << “missing country from database” << endl;
}
```

มุมคำถามและทบทวน

สิ่งที่เรียนไปทั้งหมด

- นิยามต้นไม้ และส่วนประกอบต่างๆ ของต้นไม้
- Binary Search Tree
- Self-balancing BST
- std::map<T1,T2>

คำถาม

- จงสร้าง self-balancing BST จากลำดับต่อไปนี้

23 44 20 2 30 56 60 32 65 48 46