ค่ายอบรมโอลิมปิกวิชาการ 2

โครงสร้างข้อมูล: ต้นไม้

Data Structure: Tree

รัชดาพร คณาวงษ์

13 มีนาคม 2561

ศูนย์มหาวิทยาลัยศิลปากร

ต้นไม้ (Tree) สำหรับนักคอมพิวเตอร์

- เป็นโครงสร้างข้อมูลที่แสดงถึงความสัมพันธ์ของข้อมูลแบบมีลำดับชั้น โดยเปรียบเทียบจากส่วนประกอบต่างๆ ของต้นไม้ในโลกความจริง

ส่วนประกอบที่สำคัญคือ

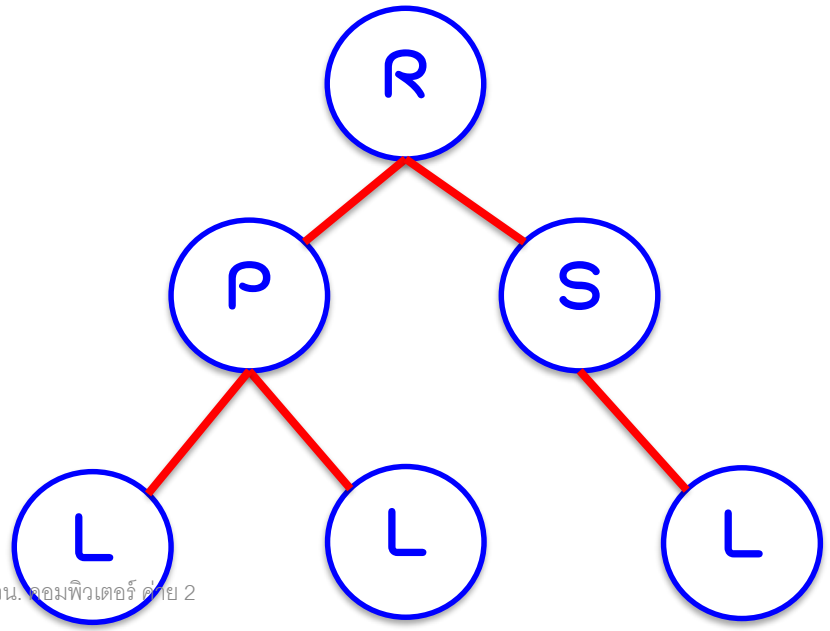
- ✧ ราก
- ✧ กิ่งก้าน
- ✧ ใบ

องค์ประกอบของต้นไม้

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

หมายเหตุ

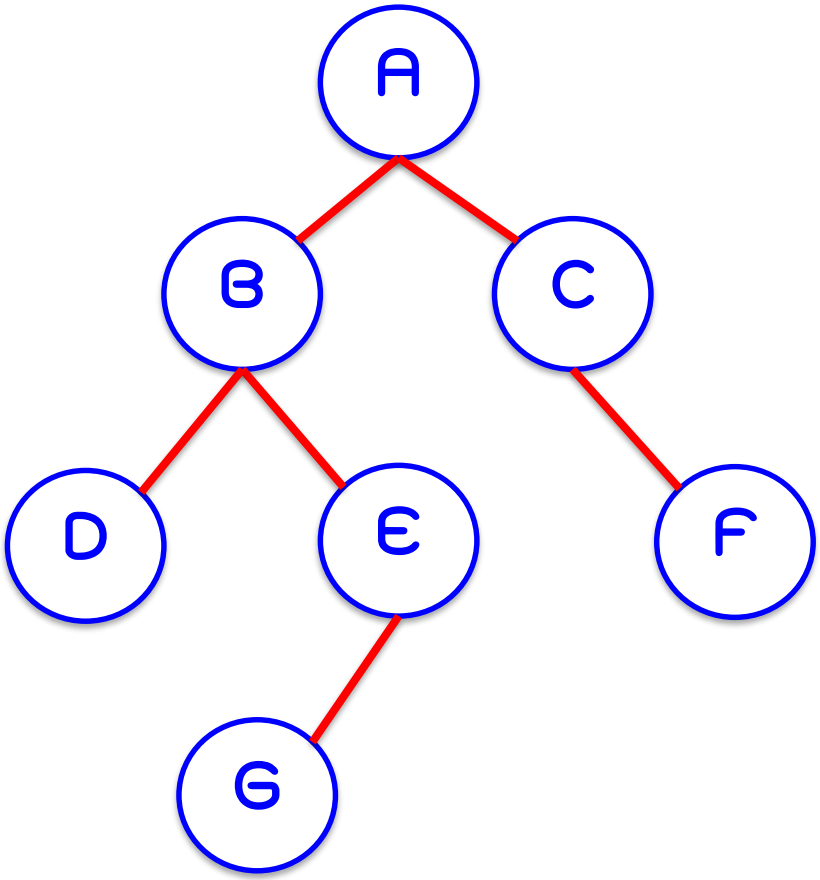
เพื่อความง่ายต่อการวาดต้นไม้ จึงนิยมวาดต้นไม้คว่ำ เริ่มจากให้รากอยู่บนสุด และวาดเส้นแทนกิ่งก้านแตกแขนงเป็นลำดับลงมาเรื่อยๆ



โหนดของต้นไม้

- โหนดบางตำแหน่งจะมีบทบาทแตกต่างกัน เช่น
  - ราก (root) คือโหนดที่อยู่บนสุด
  - ลูก (child) คือโหนดที่มีโหนดด้านบน
  - พ่อหรือแม่ (parent) คือโหนดที่มีโหนดเชื่อมต่อด้านล่าง (root ไม่ถือเป็น parent)
  - ใบ (leaf) คือโหนดที่ไม่มีโหนดเชื่อมต่ออยู่ด้านล่าง
  - 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

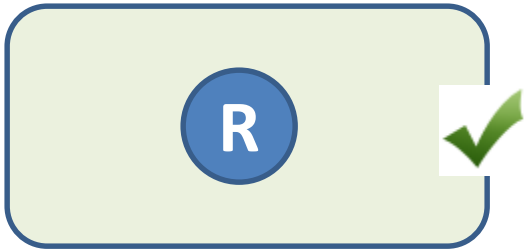
ลักษณะของต้นไม้ (Tree)

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

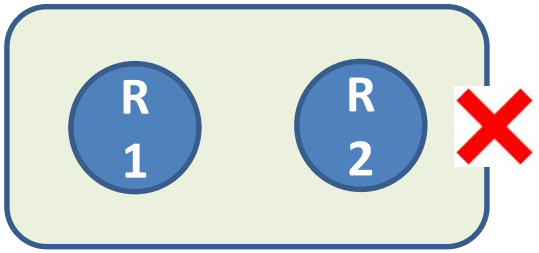
อะไรที่ใช่และไม่ใช่ทรี (สำคัญมากห้ามสับสน)

- ทรีทุกอันเป็นกราฟ แต่กราฟอาจไม่ใช่ทรี
- ไม่มีอะไรเลยก็เรียกว่าทรี (Empty tree)

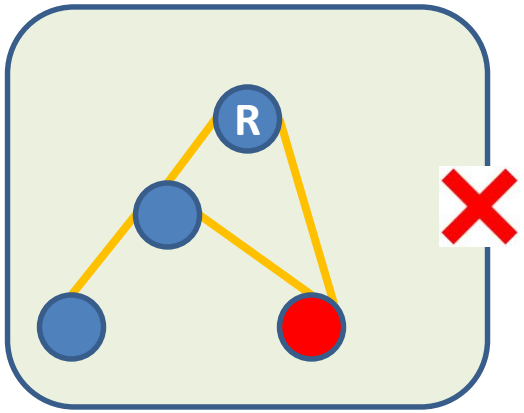
- มีแต่รูทก็เรียกว่าทรี



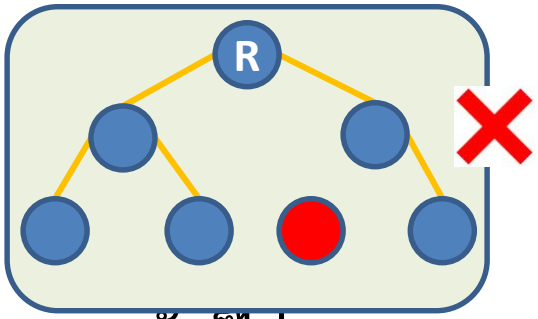
- แต่มีสองรูทถือว่าไม่ใช่ทรี



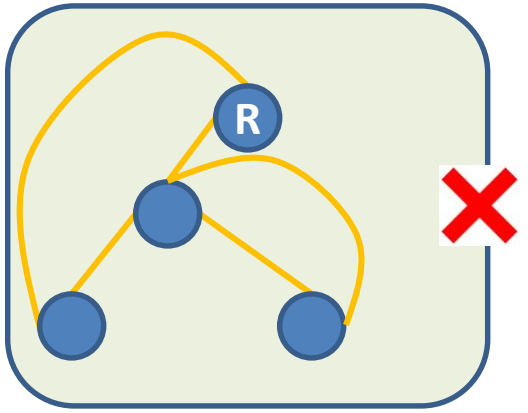
- มีโหนดที่มีหลายพ่อก็ไม่ใช่



- มีโหนดกำพร้าก็ไม่ได้ (โหนดกำพร้าที่จริงคือรูทอีกตัว)



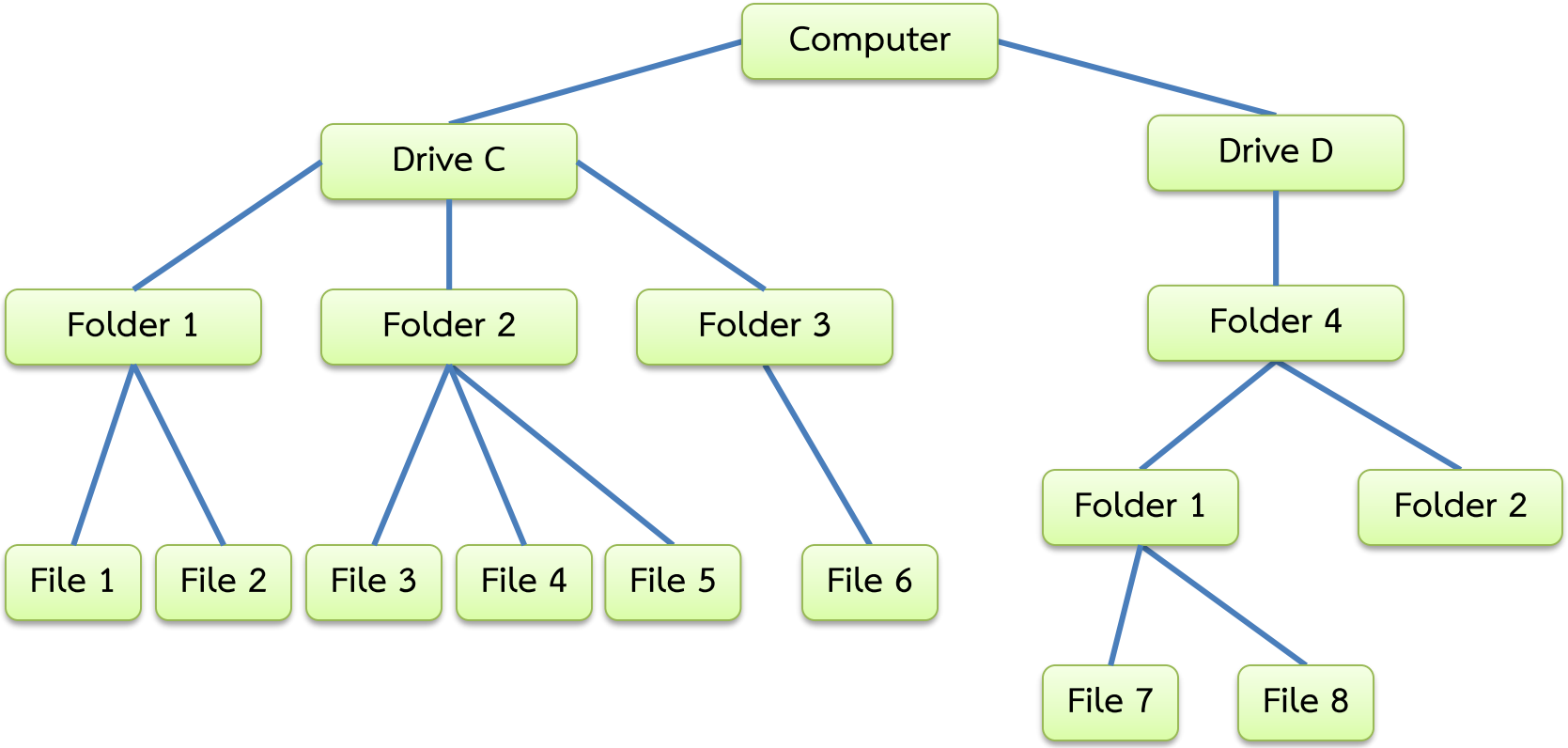
- วนกลับก็ไม่ยอม (การวนกลับทำได้ในกราฟบางประเภท แต่ไม่ใช่ต้นไม้)



****การดำเนินการ (operation) ของโครงสร้างข้อมูลแต่ละรูปแบบมีลักษณะเฉพาะของมัน จึงจำเป็นที่ต้องสร้างโครงสร้างข้อมูลที่ถูกต้อง**

ตัวอย่างโครงสร้างต้นไม้หรือทรี (tree)

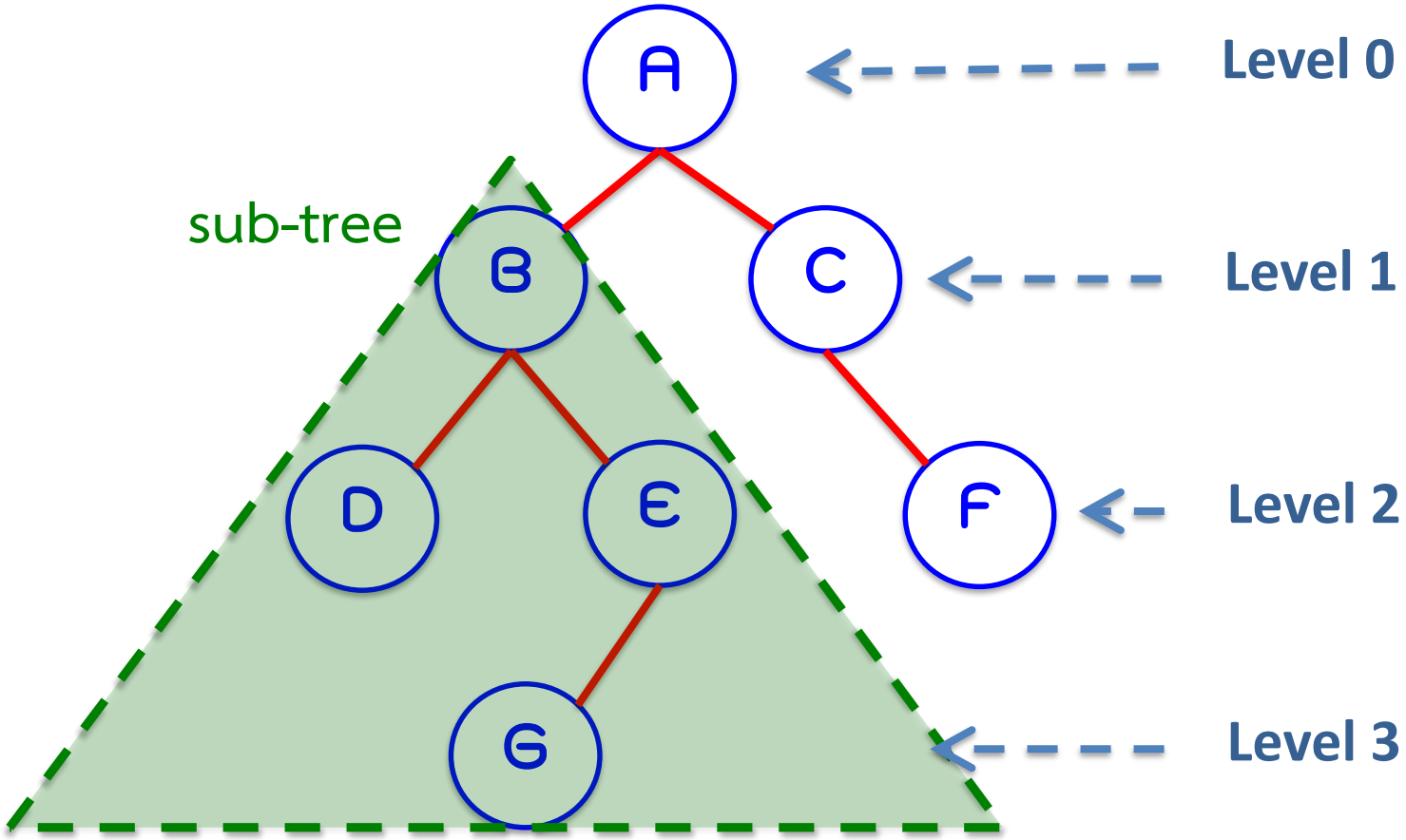
- โครงสร้างไฟล์และโฟลเดอร์



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

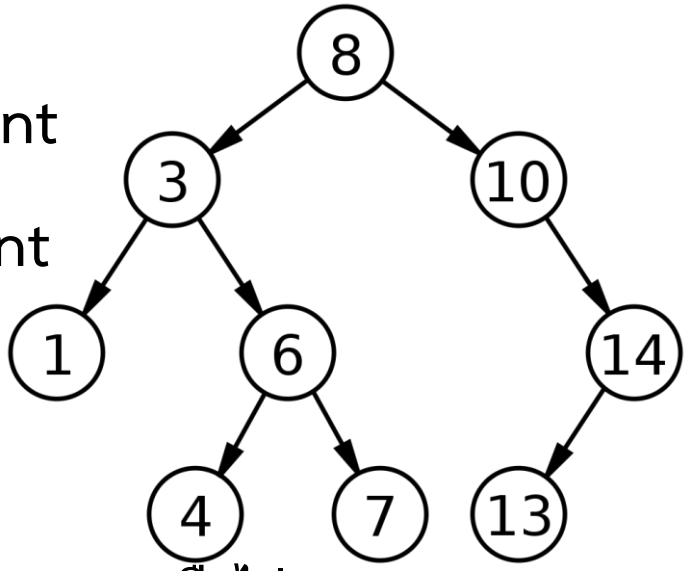


Image source: wikipedia.org

หมายเหตุ เนื่องจากทรีถูกเขียนด้วย linked-list จึงมีข้อดีกว่าการค้นหาด้วยการเรียงข้อมูลเก็บไว้ในอาร์เรย์ ในการแทรกค่าใหม่และลบค่าเดิมได้อย่างรวดเร็ว

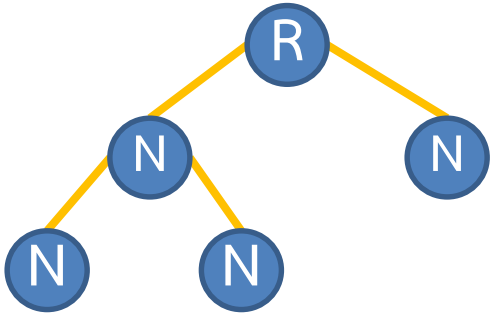
เราสามารถค้นได้อย่างรวดเร็วว่าค่า (value) ที่สนใจมีอยู่ในระบบหรือไม่

การประยุกต์ขั้นสูงขึ้นจะนำไปสู่โครงสร้างข้อมูลที่เรียกว่า Trie (ทรัย)

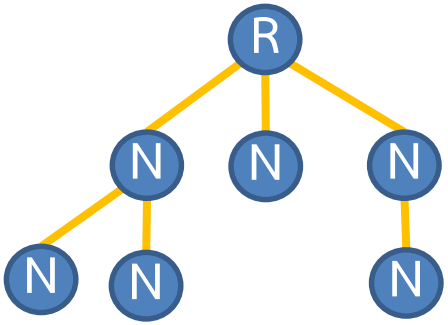
ทรัยมักถูกใช้กับการสร้างพจนานุกรมและการวิเคราะห์เอกสารข้อความ

ทรีแบบต่าง ๆ

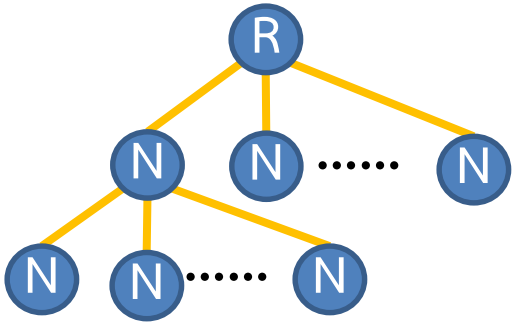
- แบ่งตามดีกรี (จำนวนโหนดลูกสูงสุดที่ยอมให้มีได้): Binary, Ternary, N-ary



Binary



Ternary



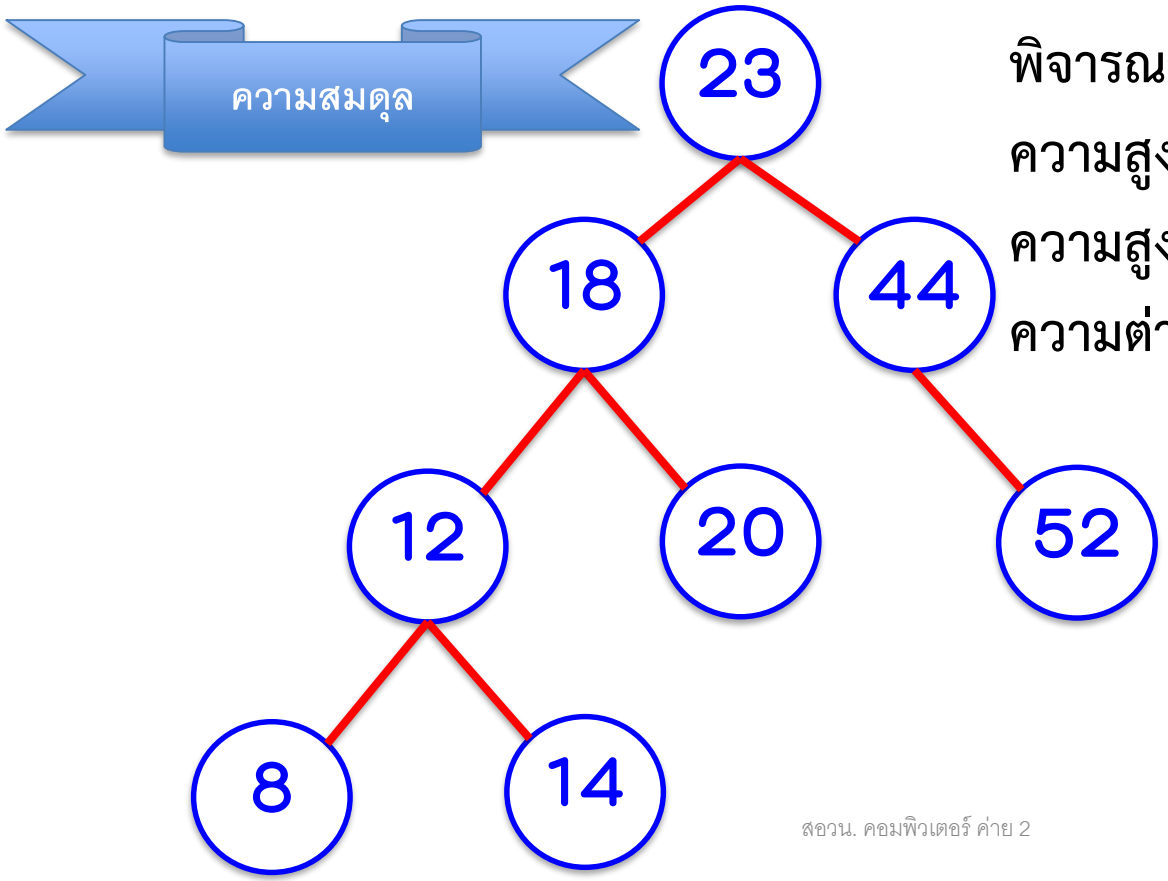
N-ary

** 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

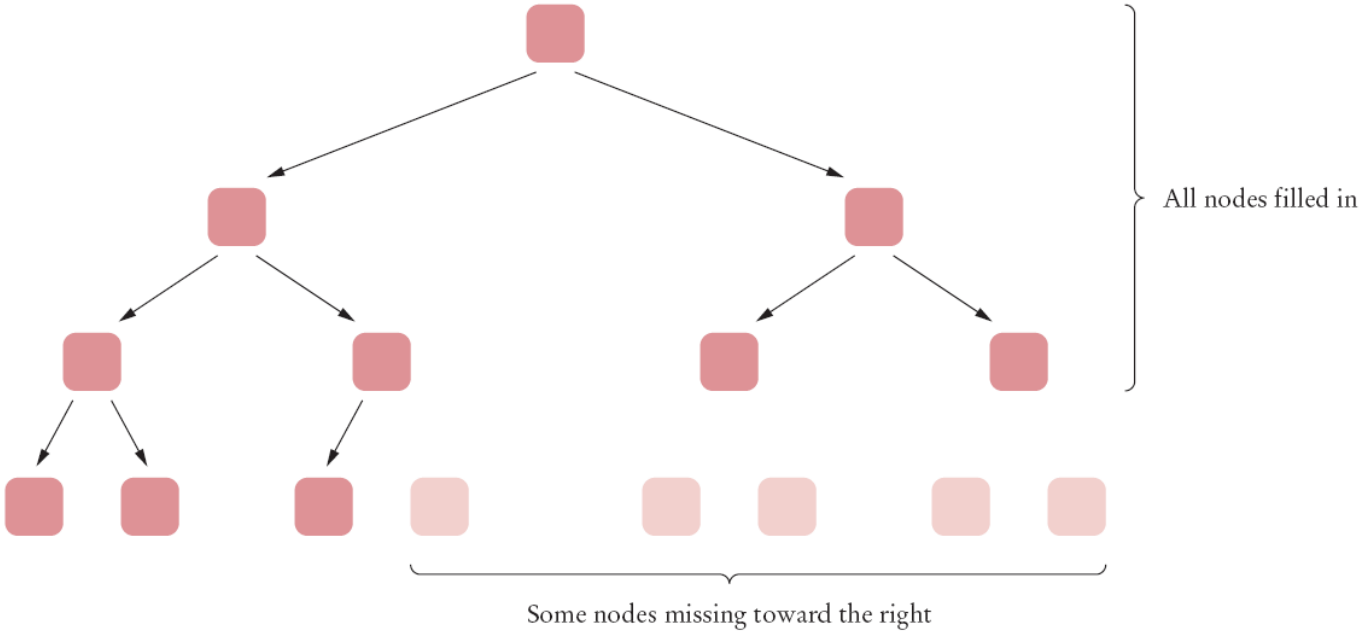
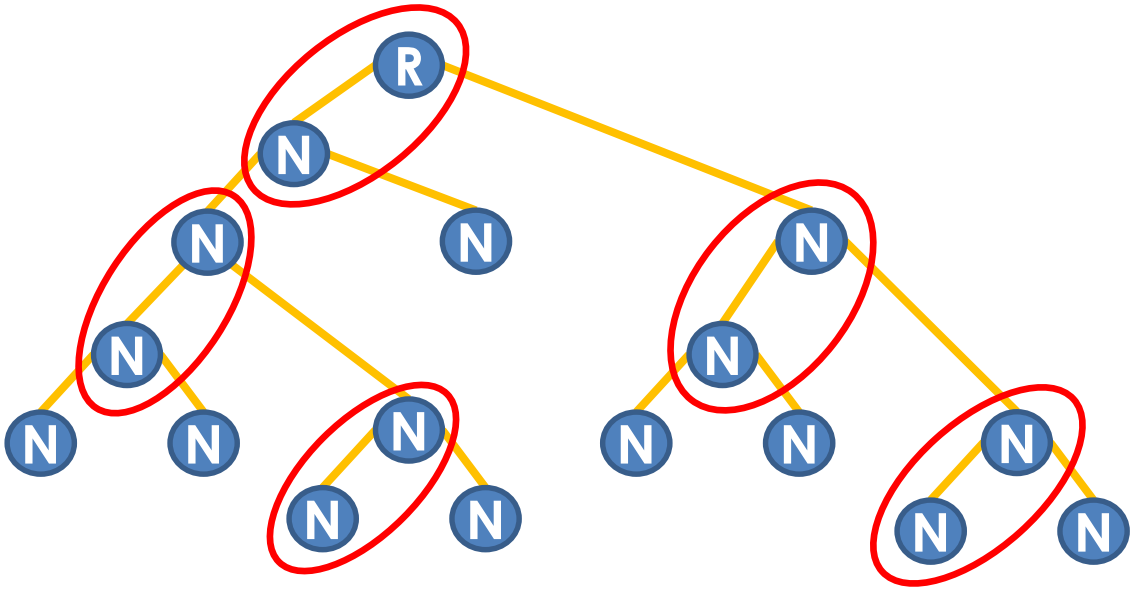


Figure 16 An Almost Complete Tree

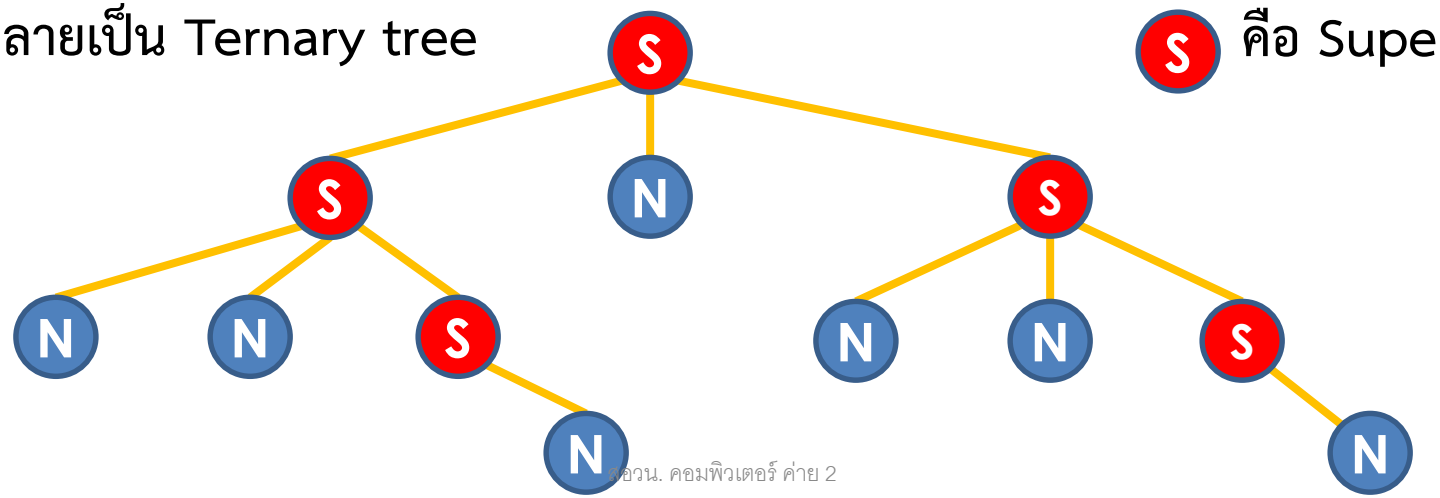
Image source: http://www.iro.umontreal.ca/~pift1025/bigjava/Ch21/ch21.html

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

ทำงานเร็ว โค้ด เข้าใจง่าย

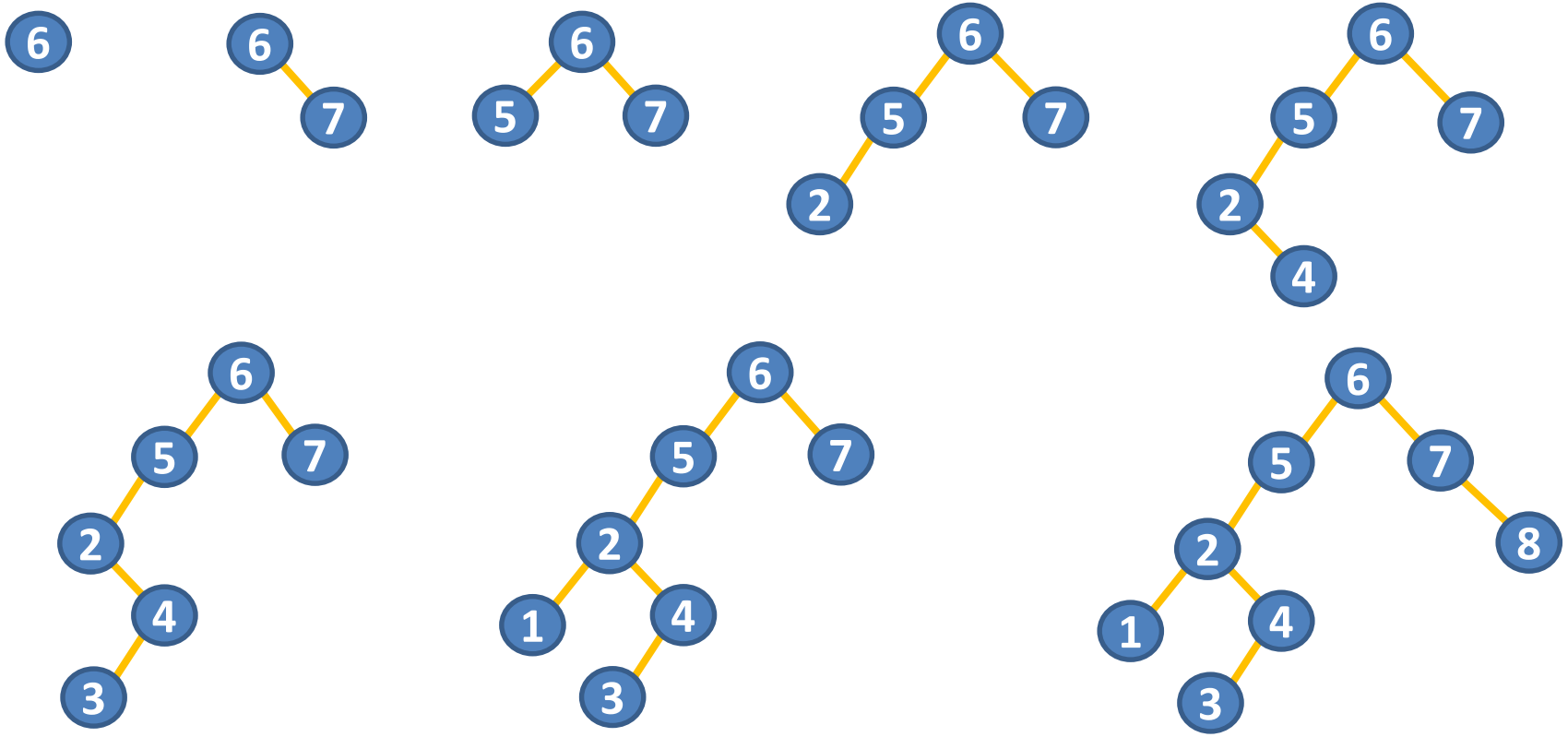
- เพาะรากขึ้นมาด้วยการ insert ค่าตัวแรกเข้าไป
  - ว่าแต่ต้องทำอย่างไง ถึงจะใส่ค่าต่าง ๆ เข้าไปใน binary search tree ได้อย่างถูกต้อง ?
  - อย่าลืมว่า binary search tree จัดลำดับตามค่าที่ใส่เข้าไป ค่าน้อยไปด้านซ้าย ค่ามากไปด้านขวา

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);
}
```

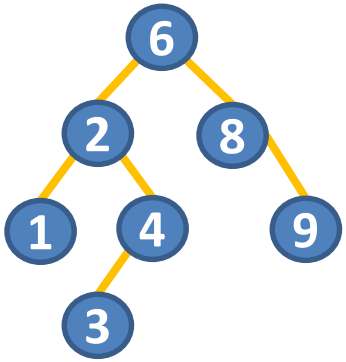
```
TreeNode* Tree::findN(int key) {
  if (root == NULL)
    return NULL;

  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



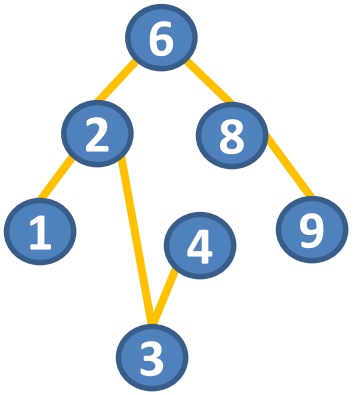
ต้องการลบ 4 ออกจากต้นไม้

มันเป็นเรื่องง่าย ถ้าโหนดที่ถูกลบมีลูกแค่อันเดียว

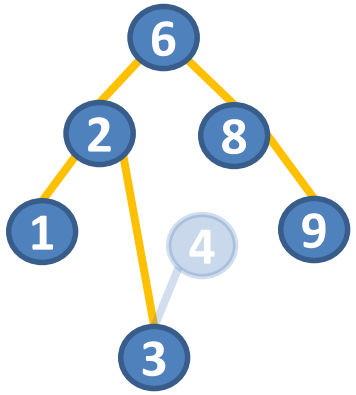
➔ โยงลิงค์ใหม่ได้เลย

โหนดทางต้นไม้ย่อยทางขวา ยังไงก็มีค่ามากกว่าโหนดทางซ้าย

➔ ลบโหนดที่ไม่ต้องการทิ้งไปได้เลย



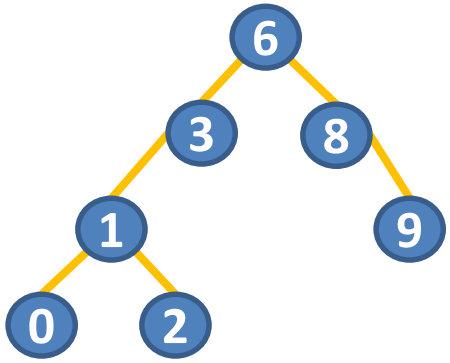
โยงลิงค์ใหม่



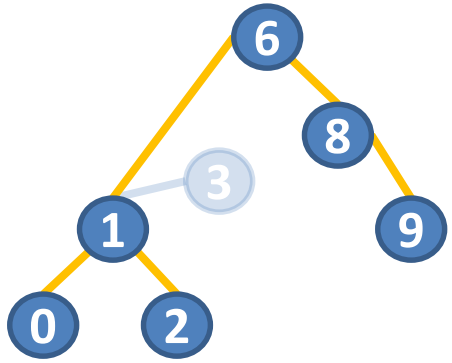
ลบโหนดทิ้ง

การลบโหนด (Remove Node) (2)

- ถ้าโหนดที่โดนลบมีลูกแค่โหนดเดียวถึงแม้จะมีทั้งลูกและหลาน ยังไงก็ง่าย

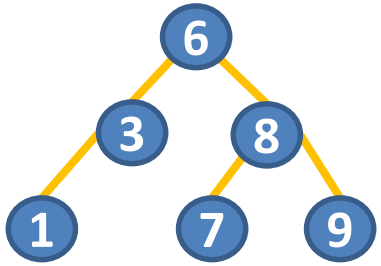


ต้องการจะลบ 3 ออกจากต้นไม้

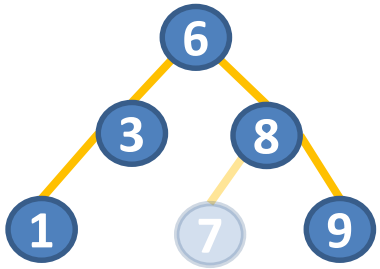


เปลี่ยนลิงค์ และลบออกได้เหมือนเดิม แทบไม่มีอะไรต่างกันเลย

- ยิ่งง่ายเข้าไปอีก ถ้าโหนดที่ถูกลบเป็นใบ (leaf) คือไม่มีโหนดลูก



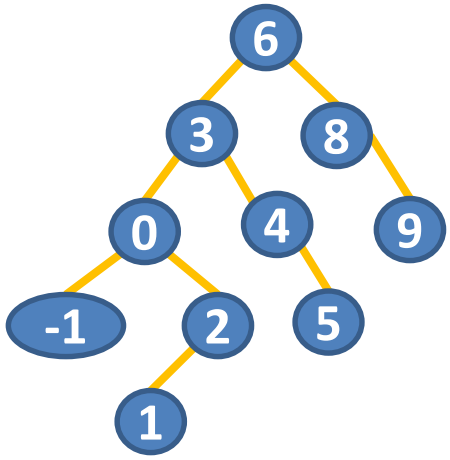
ต้องการจะลบ 7 ออกจากต้นไม้



อย่าลืมอัพเดตลิงค์ของโหนด 8 ให้กลายเป็น NULL

การลบโหนด (Remove Node) (3)

- ถ้าโหนดที่จะลบมีลูกอยู่สองโหนด การดำเนินงานจะกลายเป็นเรื่องซับซ้อนขึ้นมาทันที

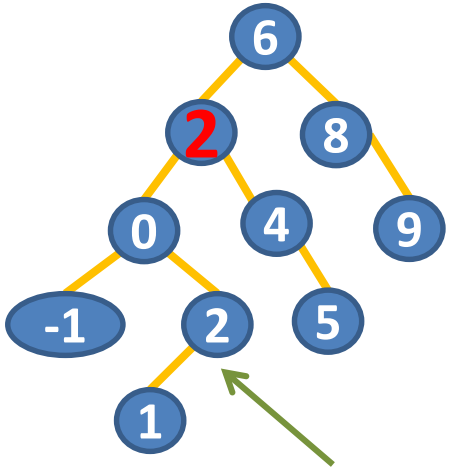


ต้องการจะลบ 3 ออกจากต้นไม้

ถ้าเราแทนค่าโหนด 3 ด้วยค่าในโหนด 2

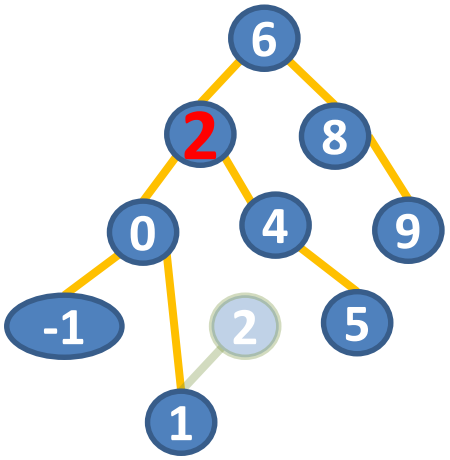
จะเปรียบได้ว่าโหนด 3 ถูกลบออก

(copy key ของโหนด 2 ไปทับโหนด 3)



มีโหนดลูกอันเดียว ลบด้วยวิธีเดิม ๆ ได้

ผลลัพธ์จากการลบโหนด 2



ผลลัพธ์ที่ได้รักษาคุณสมบัติของ binary search tree ไว้ได้ทุกประการ

“แล้วรู้ได้ไงว่าต้องเลือกโหนด 2 มีหลักการเลือกหรือเปล่า ?”

การลบโหนด (Remove Node) (4)

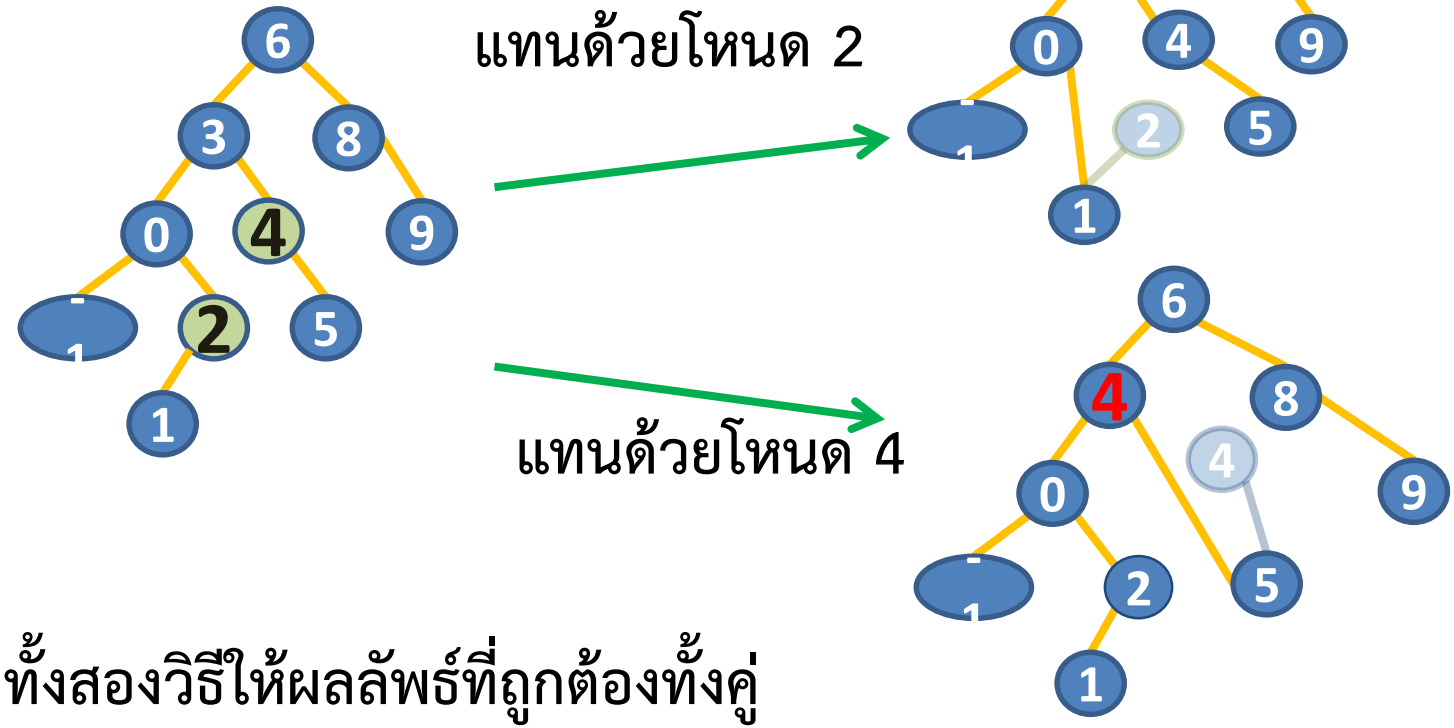
- ถ้าเลือกโหนดที่มีค่ามากสุดในต้นไม้ย่อยด้านซ้าย หรือ เลือกโหนดที่มีค่าน้อยสุดในต้นไม้ย่อยทางขวา

  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;
}
```

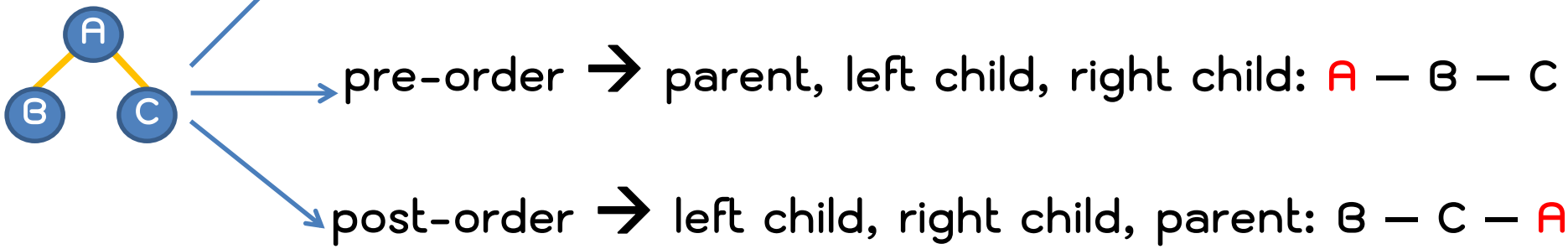
การ insert กับ remove โหนดก็ต้องเปลี่ยนไปจากเดิมด้วย

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

การแวะผ่านต้นไม้ (Tree Traversal)

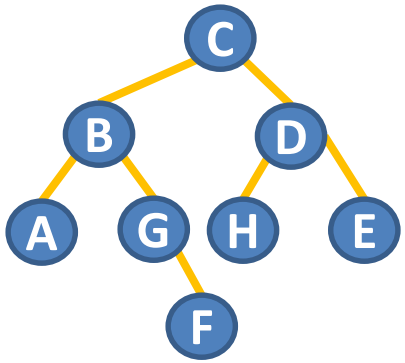
- เป็นการเดินเยี่ยมโหนดทุกโหนดในต้นไม้ (visit all nodes in a tree)
- มีอยู่สามลักษณะคือแบบ In-order (ตามลำดับ), pre-order (ก่อนลำดับ), และ post-order (หลังลำดับ)
- มุมมองของการนับลำดับดูที่ parent node เป็นตัวอ้างอิง และมองซ้อนแบบเดิมไปเรื่อย ๆ

ตัวอย่างแบบง่าย (ยังไม่จำเป็นต้องมองแบบ recursive)



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

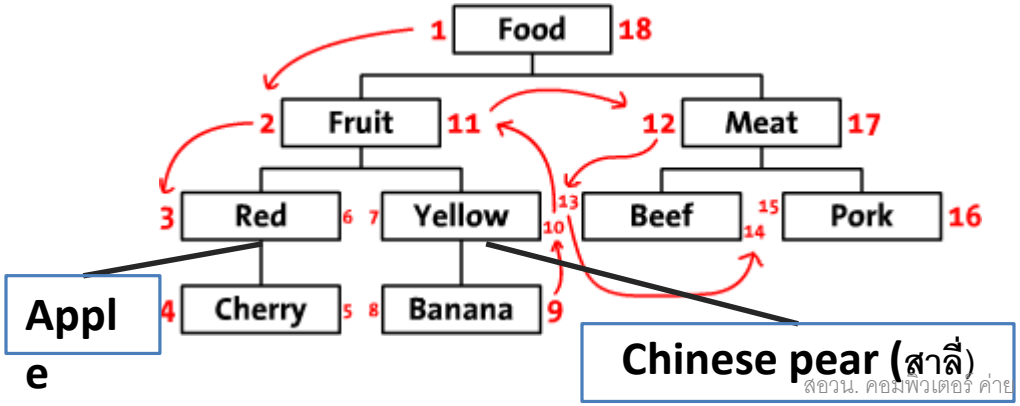
```
void inorder(TreeNode* current) {
    if (current == NULL)
        return;
    else {
        inorder(current->left);
        print(current);
        inorder(current->right);
    }
}
```

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)



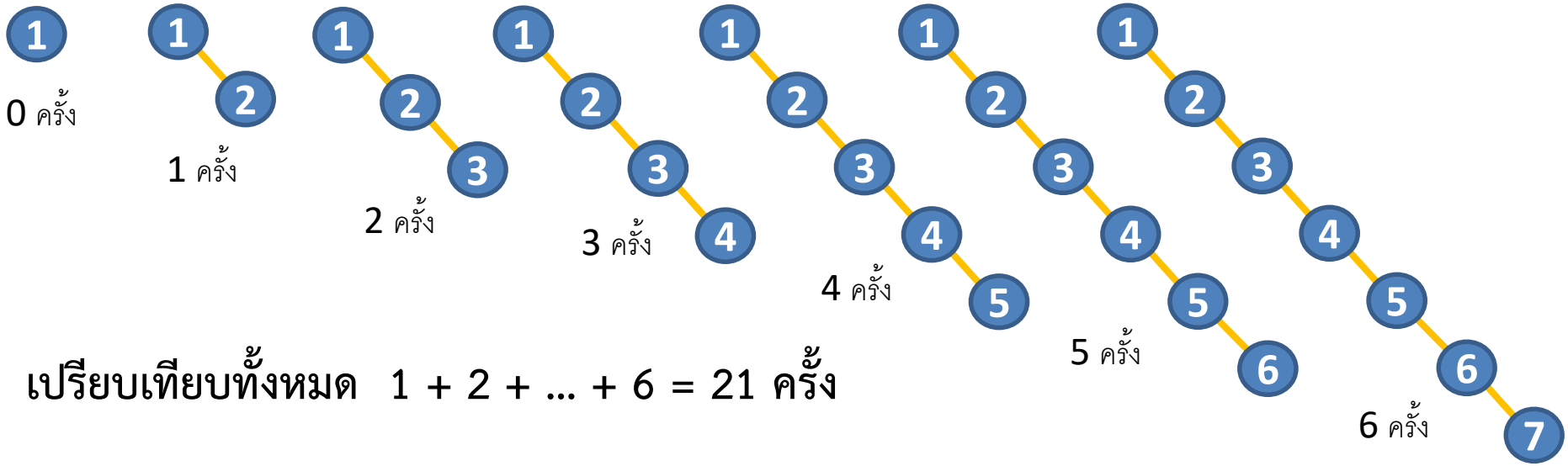
ในการประยุกต์ใช้จริง อาจจะไม่ต้องแวะผ่านต้นไม้ทั้งหมด แต่อาจจะต้องแวะผ่านต้นไม้ย่อยแทน เช่น การหาว่ามีผลไม้กี่สีและอะไรบ้าง

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

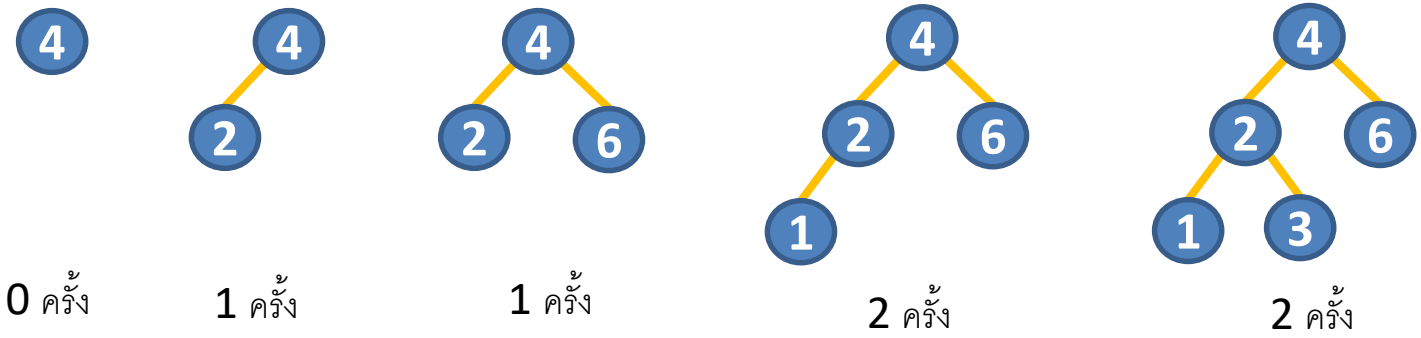
ถ้าตัวเลขมันเรียงกันอยู่แล้ว Binary Search Tree แบบนี้จะทำงานได้ช้ากว่าที่ควรจะเป็นมาก

วิเคราะห์การทำงานของ Binary Search Tree

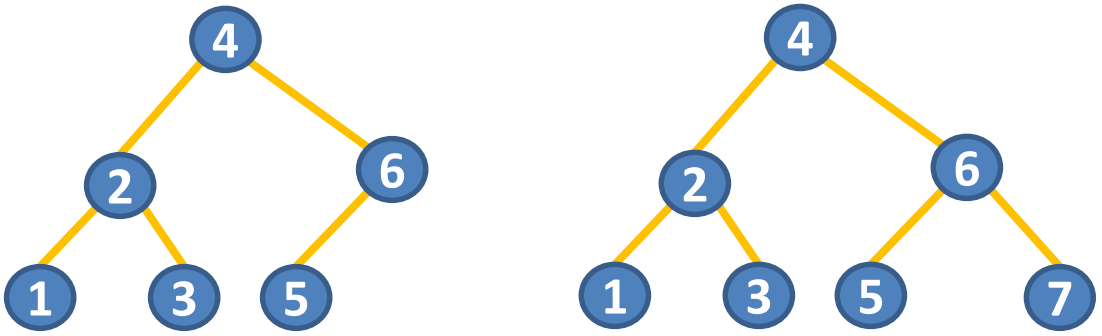
- เราต้องการให้การค้นหา การใส่ข้อมูล การลบข้อมูล มีการเปรียบเทียบตัวเลขให้น้อยครั้งที่สุด

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

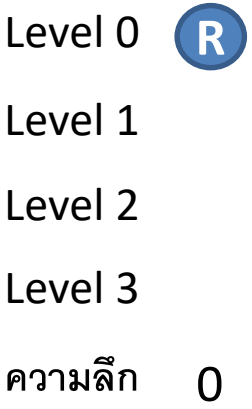
= 10 ครั้ง < 21 ครั้ง

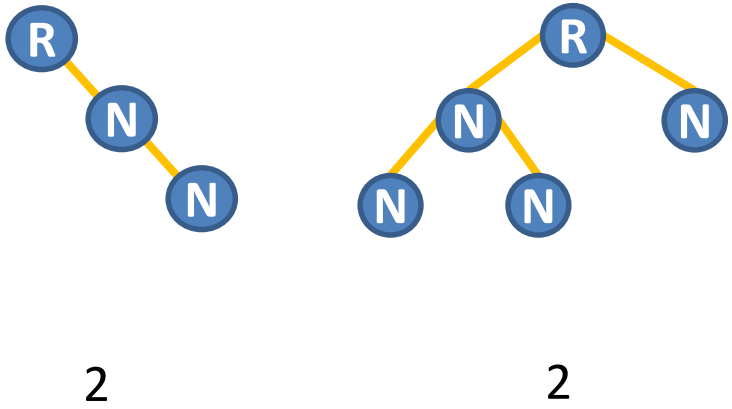
บางทีตัวเลขที่เข้ามาแบบเหมือนสุ่มมาจำทำให้ Binary Search Tree ทำงานได้เร็ว

ความลึกของต้นไม้ (Depth of Tree)

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

ความลึกและการค้นหา





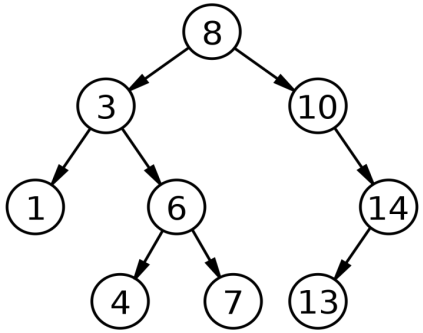
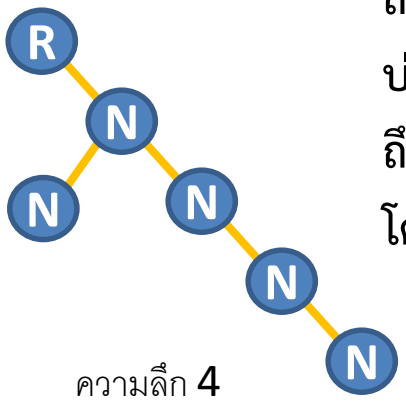


Image source: wikipedia.org



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