Tree 2

Tree

1. Tree Definitions
2. Binary Tree
   - Traversals
   - Binary Search Tree
   - Representations
   - Application : Expression Tree

3. AVL Tree
4. Which Represention ?
5. n-ary Tree
6. Generic Tree
7. Multiway Search Tree
8. B-Trees

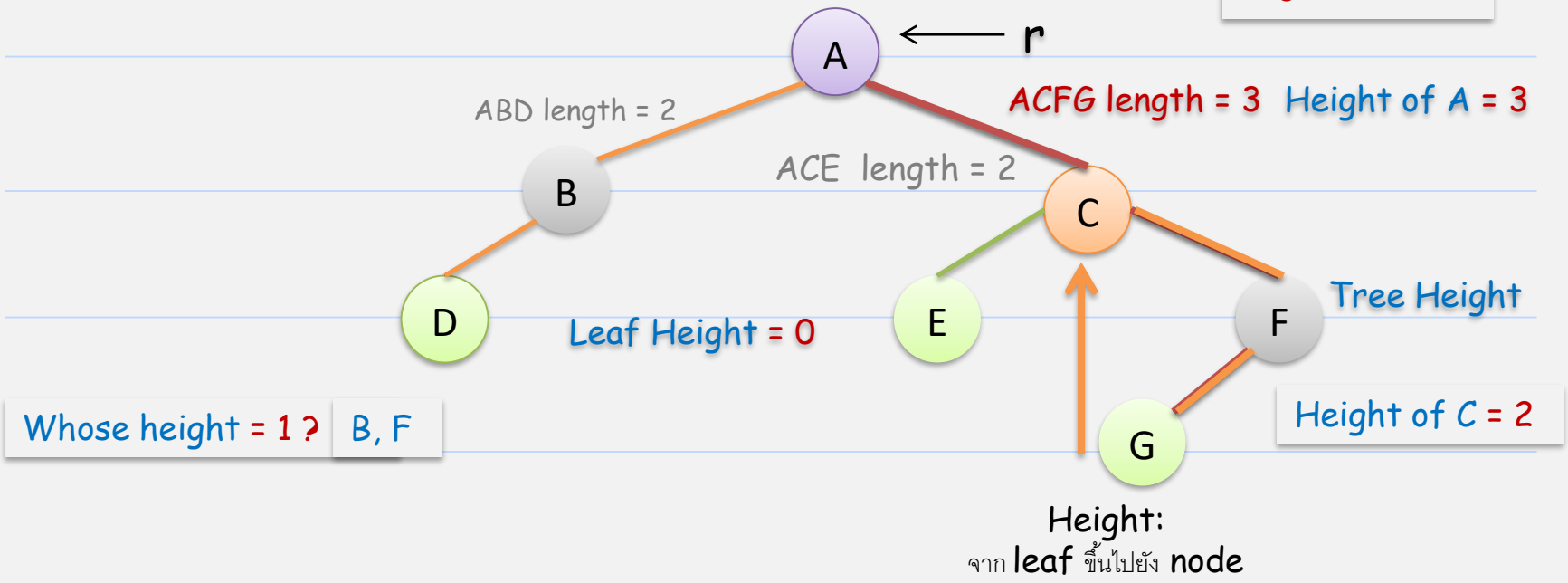
Height

Height of node = longest path length from node to leaf
path ที่ยาวที่สุดจาก node นั้น ถึง leaf

Empty Tree Height = -1

Tree Height
= Root Heigth
= 3

Height of A = ?



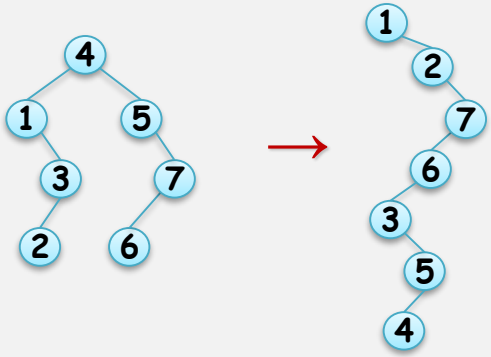
Whose height = 1 ? B, F

Height:
จาก leaf ขึ้นไปยัง node

Why Balanced Tree ?

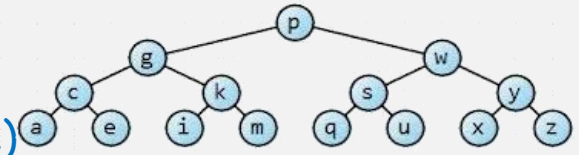
Tree Efficiency (searching, inserting, deleting,...) → O(Height)

Binary Search Tree
sometimes turn degenerated



Linear Linked list or near
Linear Search O(n)

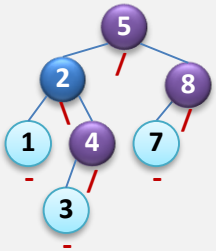
Binary Search Tree
(Perfect) Balanced Tree
Every leaf is at the same level (very rear case)



Balanced Binary Search Tree

$O(\log_2 n)$

Binary Search Tree
Hight Balanced Tree → AVL Maximum height <= 1.44 $\log_2 n$
Every node's subtrees' heights differ <= 1



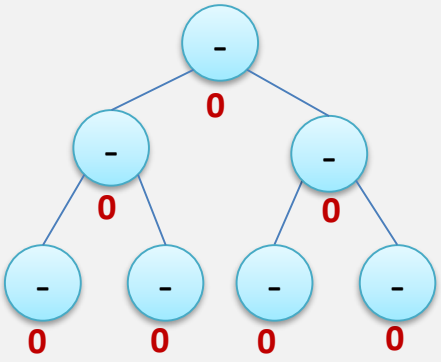
Height Balanced

$O(\log_2 n)$

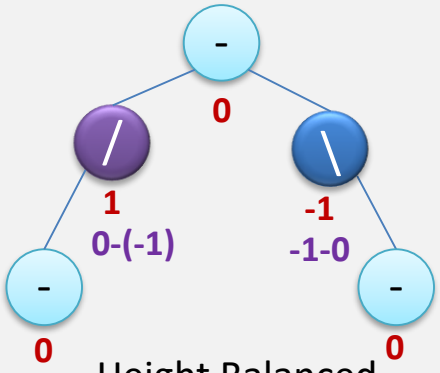
Balance Factor

Height Balanced Tree : Every node's subtrees' heights differ <= 1
(Nearly Balanced) Balance factor of every node is 0 or 1 or -1

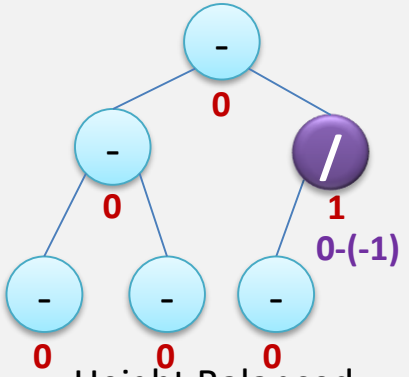
Balance factor of a node : different of left & right subtrees' hights
= Height(Left Subtree) – Height(Right Subtree)



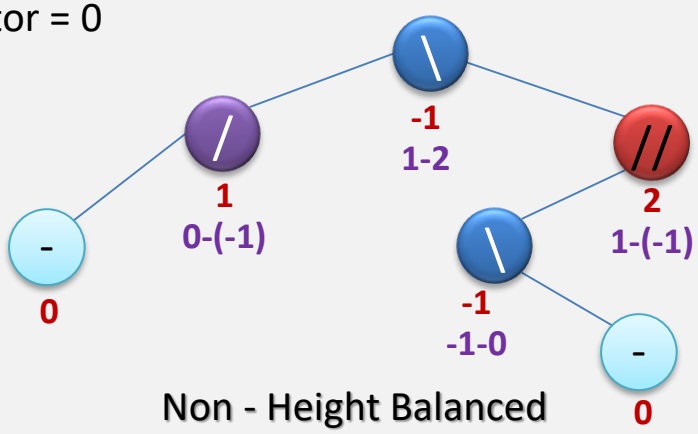
Height Balanced
Every node's balance factor = 0



Height Balanced



Height Balanced



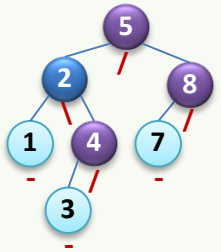
Non - Height Balanced

AVL (Height Balanced) Tree

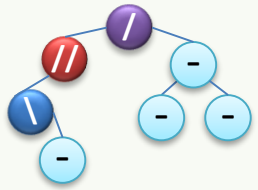
AVLTree Self- rebalancing binary search tree to keep height balance.

- Maximum height <= 1.44 $\log_2 n$: near the ideal complete binary tree.
- Search, insertion, deletion (both average & worst cases) $O(\log_2 n)$
- Insertions & deletions may require 1 or > 3 rotations.
- Worst case not much slower than random binary search tree.

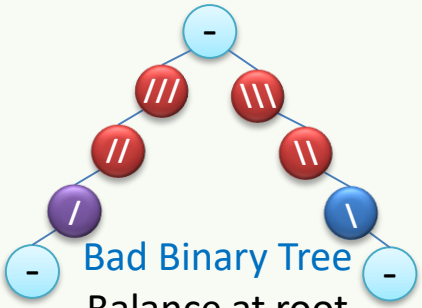

Adelson-Velskii Landis
(1962 Soviet inventors)



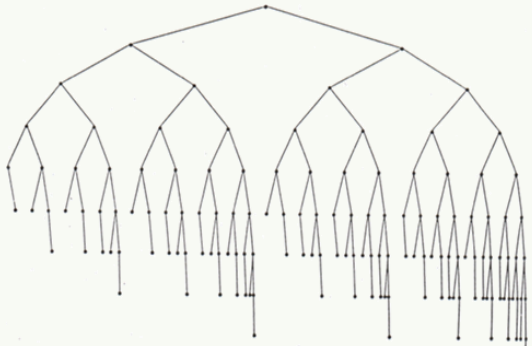
Height Balanced



Non-Height Balanced



Bad Binary Tree
Balance at root
is not enough.



Smallest AVL tree of height 9

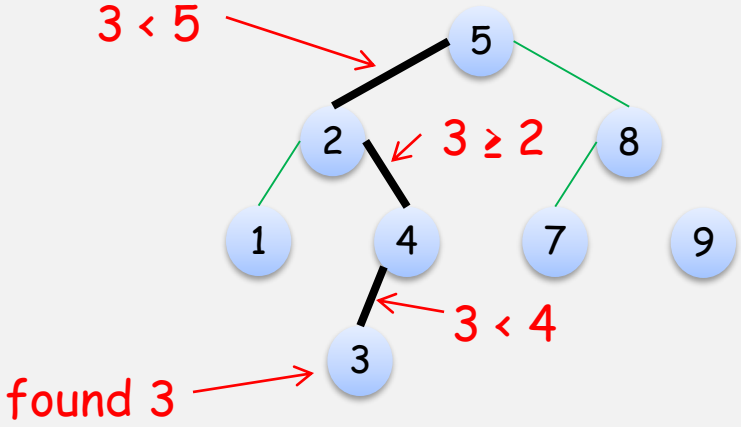
Searching in AVL = Searching in BST

AVL is a binary search tree.

Review Searching a BST (Binary Search Tree).

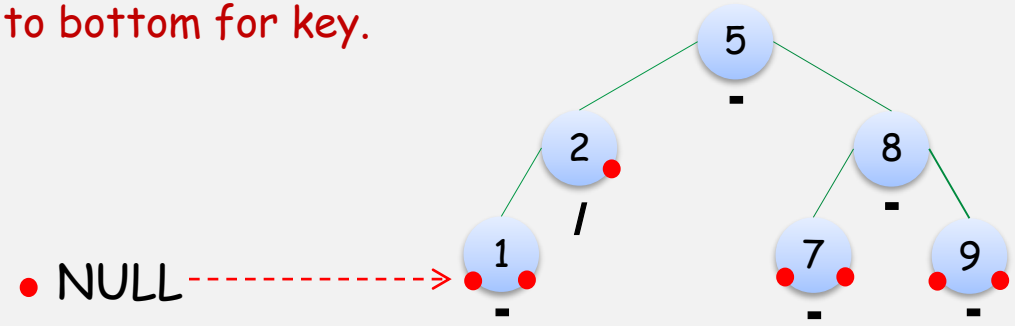
- Compare search key against key in node start from root.
- Follow associated link (recursively).
  - Left link if search key < key in node
  - Right link if search key ≥ key in node

Ex. Search for 3

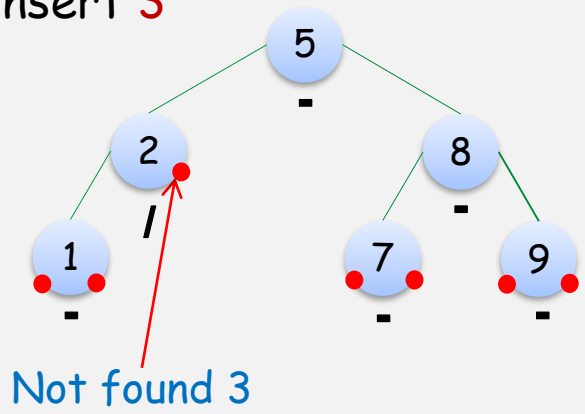


Review Inserting in BST

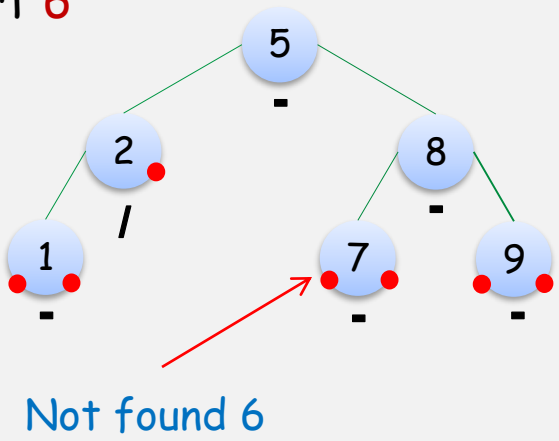
- Search to bottom for key.



Ex. Insert 3

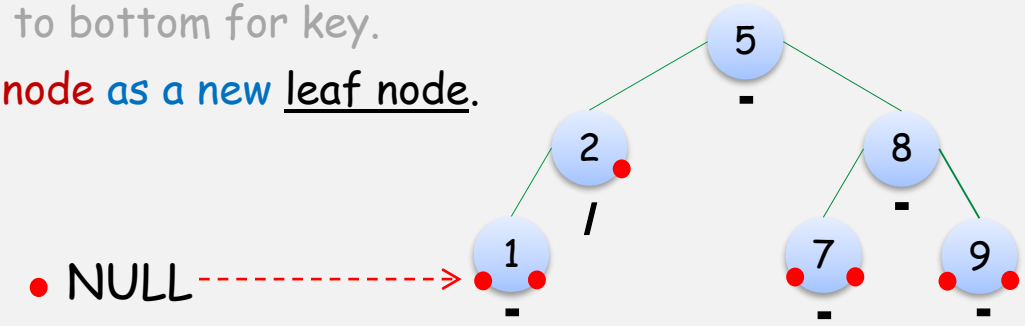


Ex. ert 6

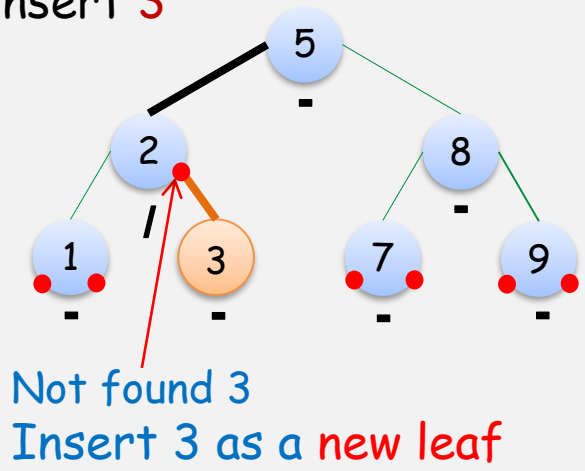


Review Inserting in BST

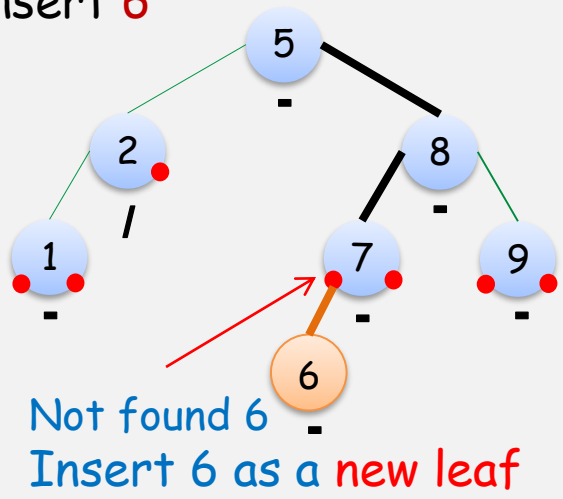
- Search to bottom for key.
- Insert node as a new leaf node.



Ex. Insert 3



Ex. Insert 6



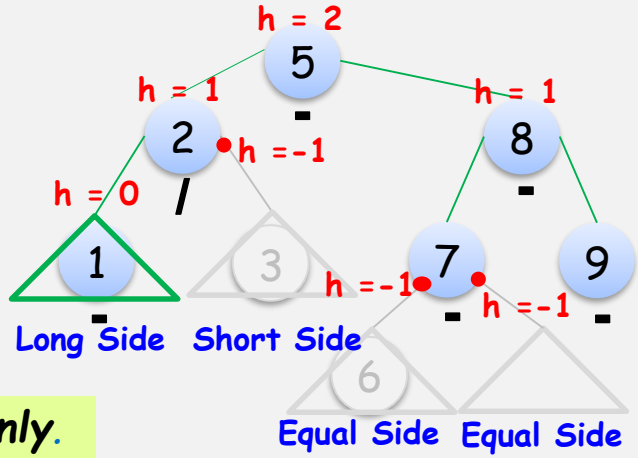
Inserting in AVL without Rebalancing

Inserting at short side or equal side

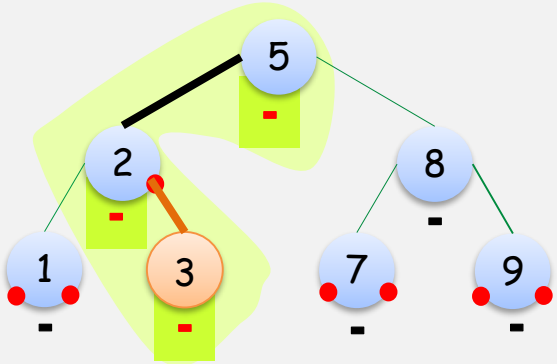
→ AVL 's still height balanced

- Insert BST new leaf node.
- Inserting node will
  change the height of some nodes (*increase* by 1)
  on the path from the newly inserted node to the root ***only***.

Change balance factors along the path to the root.

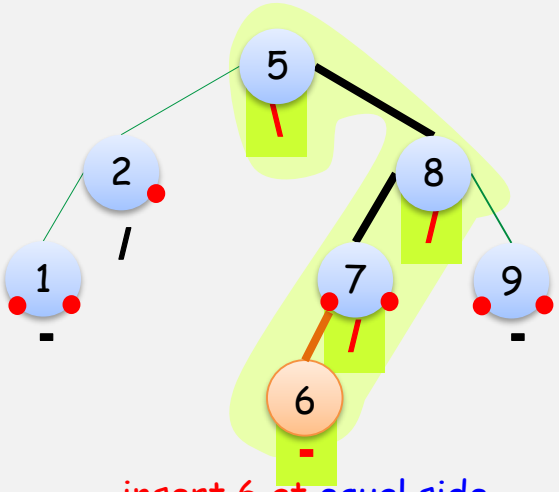


Ex. Insert 3 at short side.



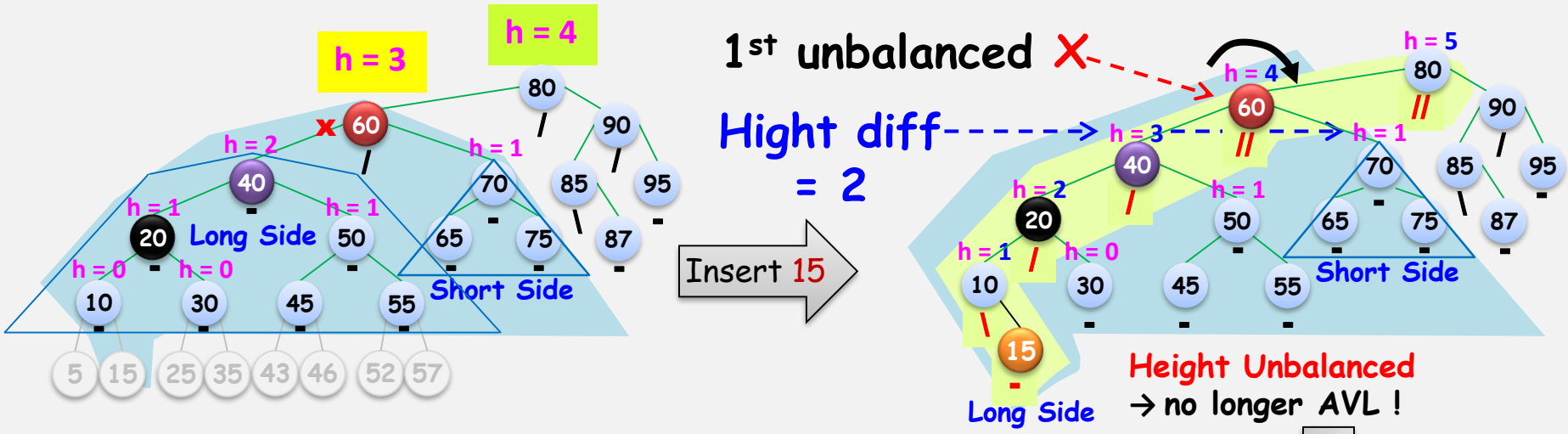
insert 3 at short side
still height balance

Ex. Insert 6 at equal side.



insert 6 at equal side
still height balance

Insertion Re-balancing in AVL



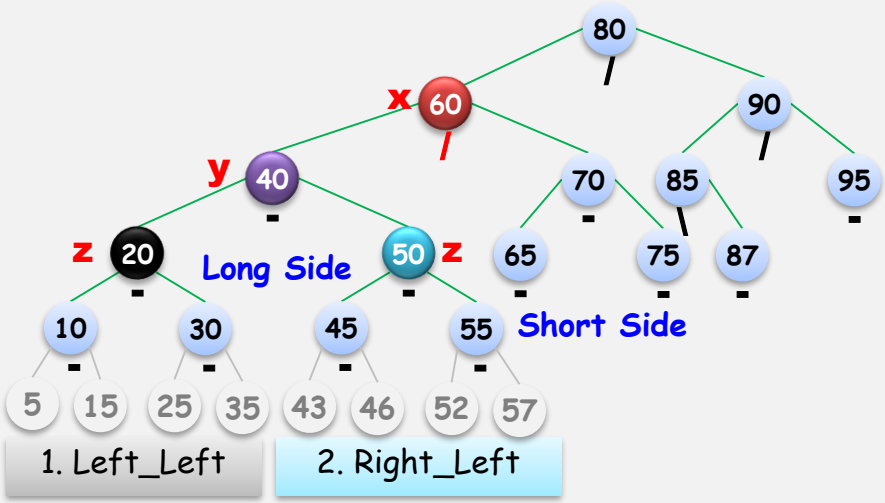
Inserting AVL in a **long side**.

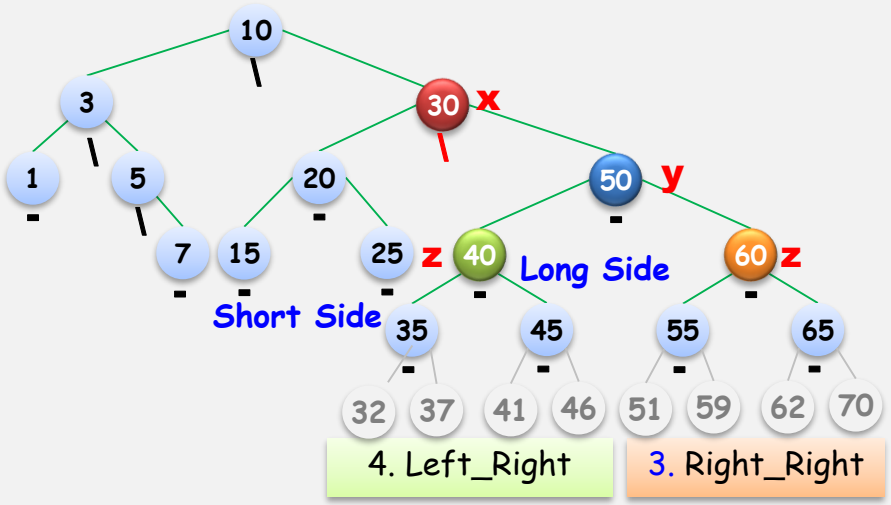
1. Causes some node(s) unbalanced (height diff = 2) ***only*** up the inserted path.
   - x = 1st unbalanced node
2. Need rebalancing the (blue) shaded subtree.

   Rebalancing does not change :
   - height of the subtree
   - → ie. Nor height of root.
   - → Only some part of the shaded subtree's balance facters need changing.

4 Insertions Re-balancing in AVL

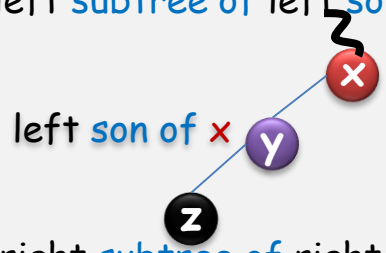
Inserting AVL in a **long side** : ให้ x = node แรกที่ไม่ balance , ตาม path ที่ insert ให้ y เป็นลูกของ x และ z เป็นลูกของ y



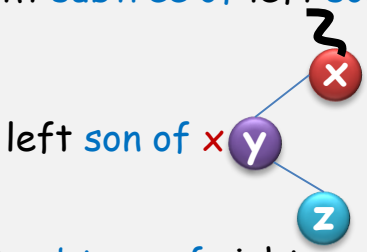


มี 4 กรณี เมื่อ **insert** ที่

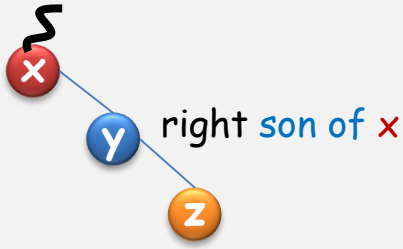
1. left subtree of left son of x



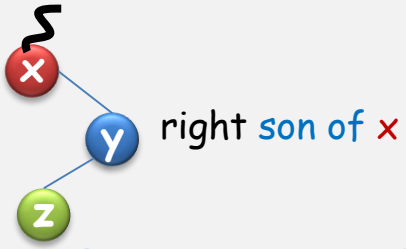
2. right subtree of left son of x



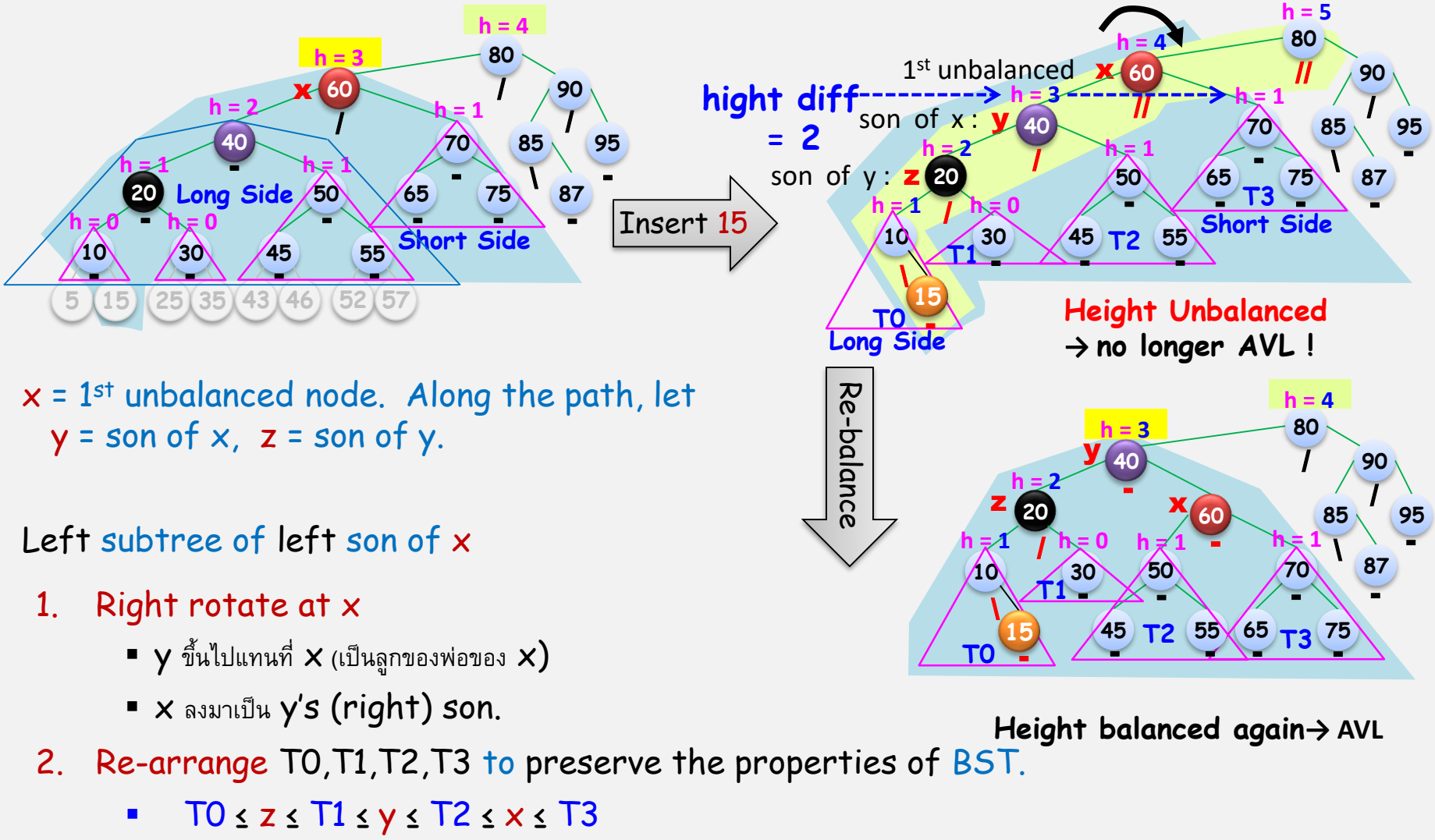
3. right subtree of right son of x



4. left subtree of right son of x



Rebalancing Left Subtree of Left Son of x (1st Unbalanced Node)



x = 1st unbalanced node. Along the path, let
y = son of x, z = son of y.

Left subtree of left son of x

1. Right rotate at x
   - y ขึ้นไปแทนที่ x (เป็นลูกของพ่อของ x)
   - x ลงมาเป็น y's (right) son.
2. Re-arrange T0,T1,T2,T3 to preserve the properties of BST.
   - T0 ≤ z ≤ T1 ≤ y ≤ T2 ≤ x ≤ T3
3. Fixing some part of the shaded subtree's balance facters.
4. Height of the blue shaded subtree doesn't change -> also the whole AVL's.

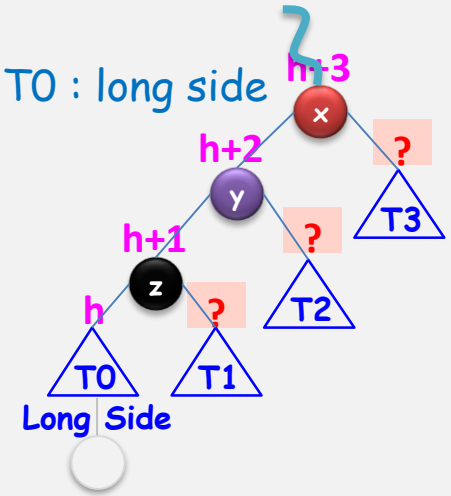
Proof : Rebalancing Area' Height's unchange (Left Subtree of Left son of x)

Insert at a long side **T0**

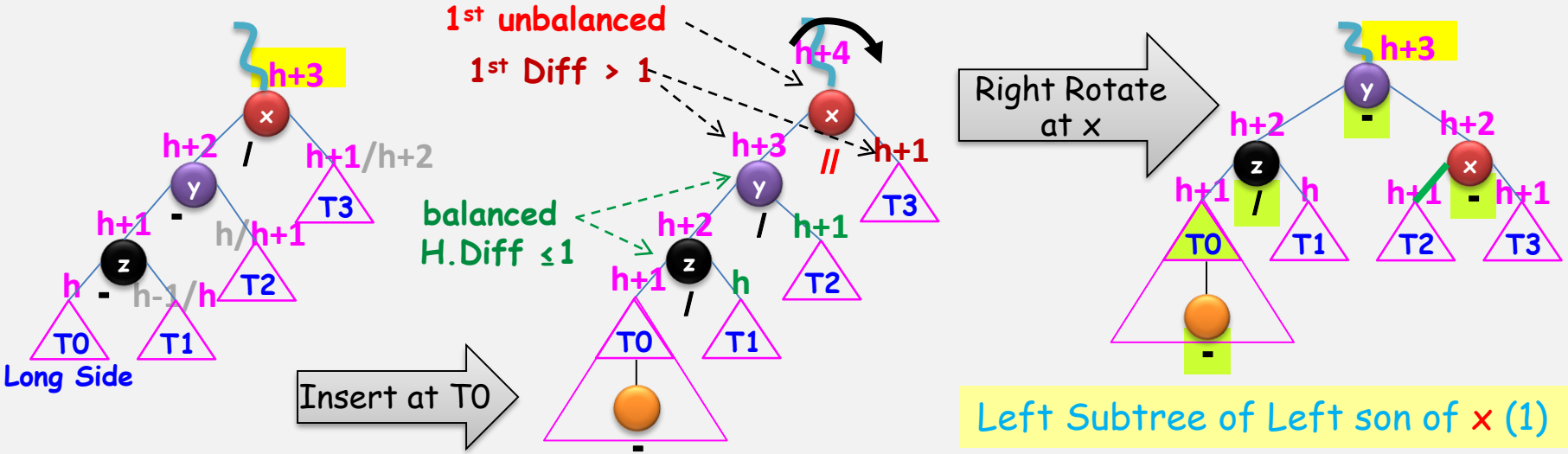
x = 1st unbalanced node, y = son of x, z = son of y. ก่อน insert:

- height(T1) = ? When height(T0) = h.
  = h-1 หรือ h ∵ heightDiff (T0,T1) ≤1 & ≠ h+1 ∵ T1 : short side, T0 : long side
  = h ∵ หลัง insert , z : balanced ∴ heightDiff ≤1
- height(T2) = h+1 ∵ หลัง insert , y : balanced
- height(T3) = h+1 ∵ หลัง insert , x : unbalanced

1. Rebalancing ไม่เปลี่ยน heights ของ subtree → ie. รวมทั้ง height ของ root.



2. ดังนั้นจึงแค่ แก้ balance facters ตาม inserted path



Left Subtree of Left son of x (1)

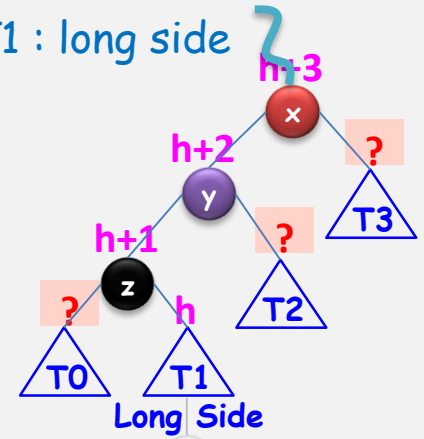
Proof : Rebalancing Area' Height's unchange (Left Subtree of Left son of x)2

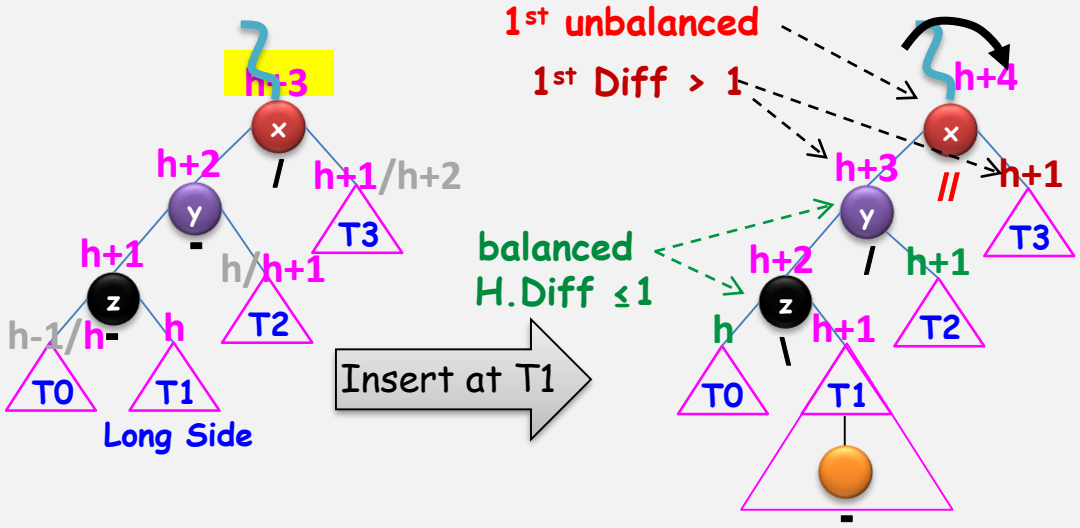
Insert at a long side **T1**

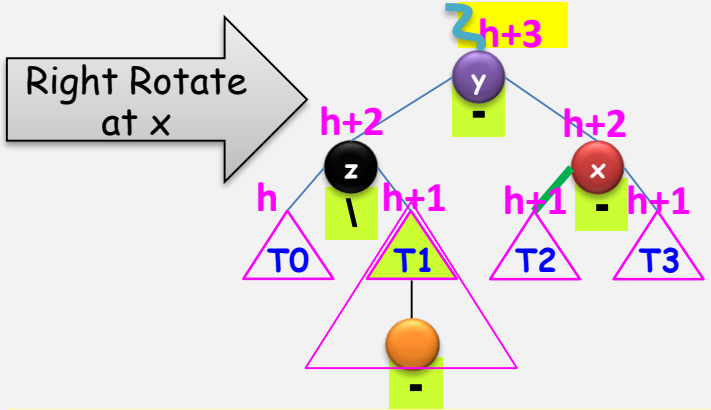
x = 1st unbalanced node, y = son of x, z = son of y. Before insertion :

- height(T0) = ? When height(T1) = h.
  = h-1 or h ∵ heightDiff (T0,T1) ≤1 & ≠ h+1 ∵ T0 : short side, T1 : long side
  = h ∵ After inserting , z : balanced ∴ heightDiff ≤1
- height(T2) = h+1 ∵ After inserting , y : balanced
- height(T3) = h+1 ∵ After inserting , x : unbalanced

1. Rebalancing does not change heights of the subtree → ie. Nor of root.
2. So only fix balance facters of inserted path to y and of x.





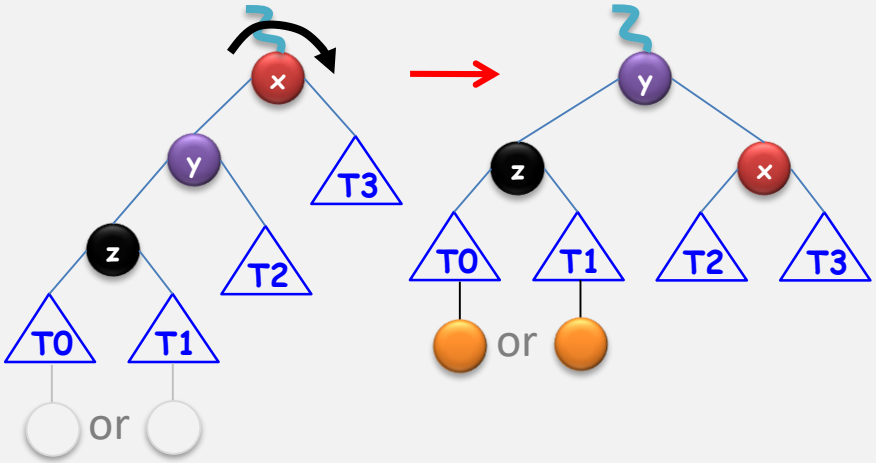


Left Subtree of Left son of x (2)

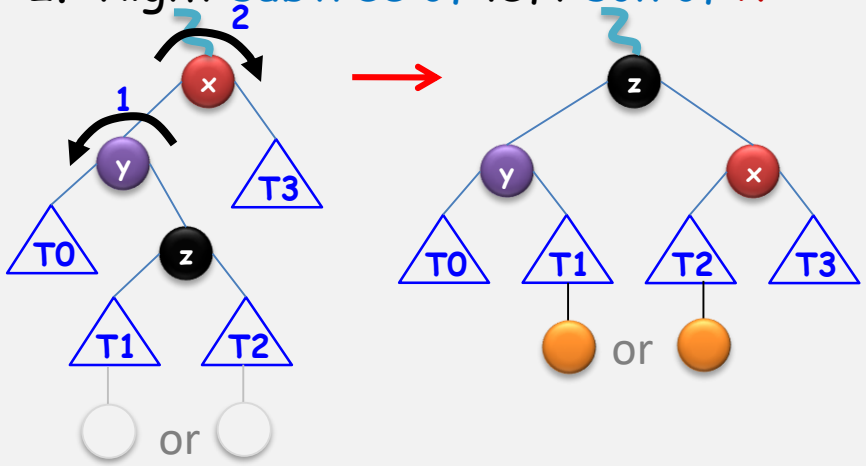
4 Kinds of Rebalancing in AVL

x = 1st unbalanced node, y = son of x, z = son of y

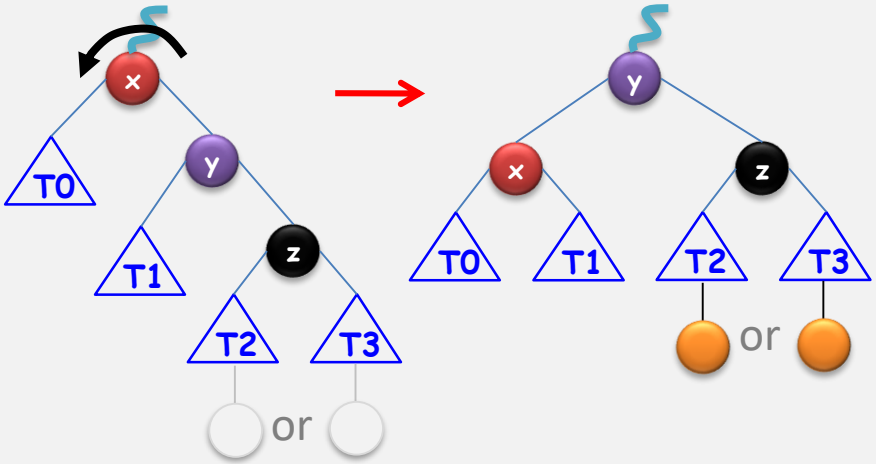
1. Left subtree of left son of x



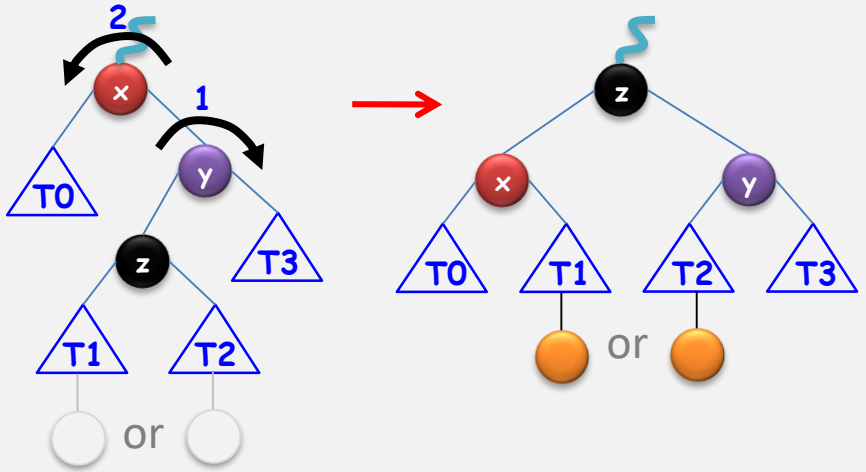
2. Right subtree of left son of x



3. Right subtree of right son of x



4. Left subtree of right son of x



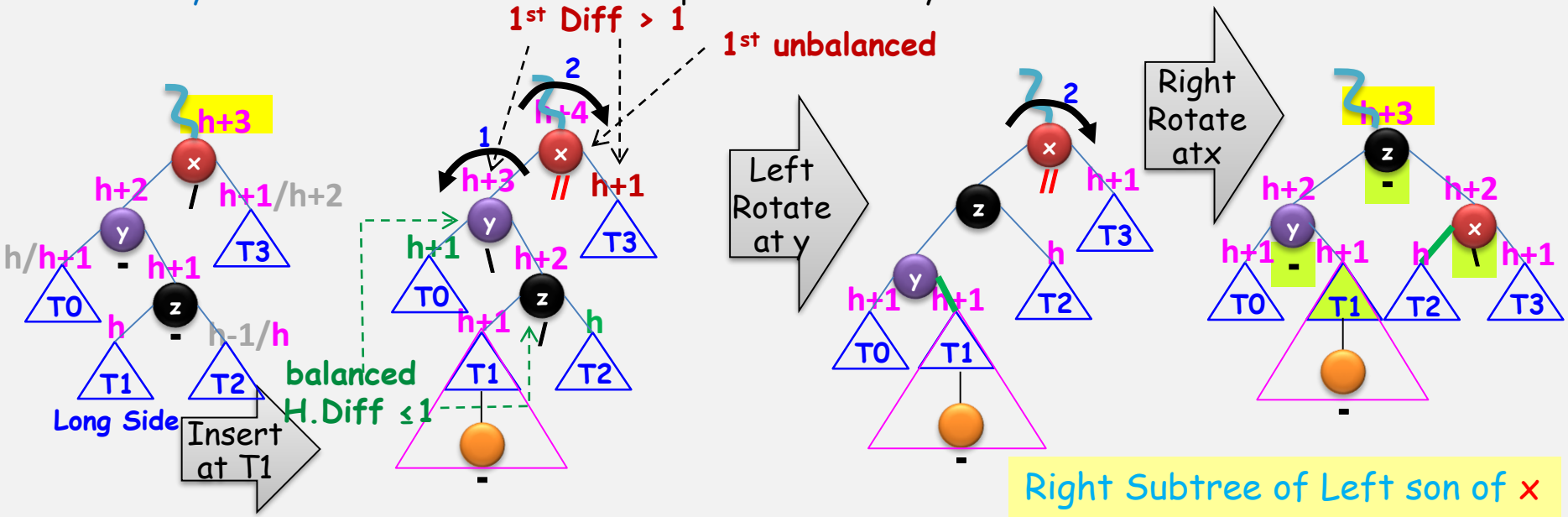
Proof : Rebalancing Area' Height's unchange (Right Subtree of Left son of x)

Insert at a long side **T1**

x = 1st unbalanced node, y = son of x, z = son of y. Before insertion :

- height(T2) = ? When height(T1) = h.
  = h-1 or h ∵ heightDiff (T2,T1) ≤1 & ≠ h+1 ∵ T2 : short side, T1 : long side
  = h ∵ After inserting , z : balanced ∴ heightDiff ≤1
- height(T0) = h+1 ∵ After inserting , y : balanced
- height(T3) = h+1 ∵ After inserting , x : unbalanced

1. Rebalancing does not change heights of the subtree → ie. Nor of root.
2. So only fix balance facters of inserted path to z and of y.



Right Subtree of Left son of x

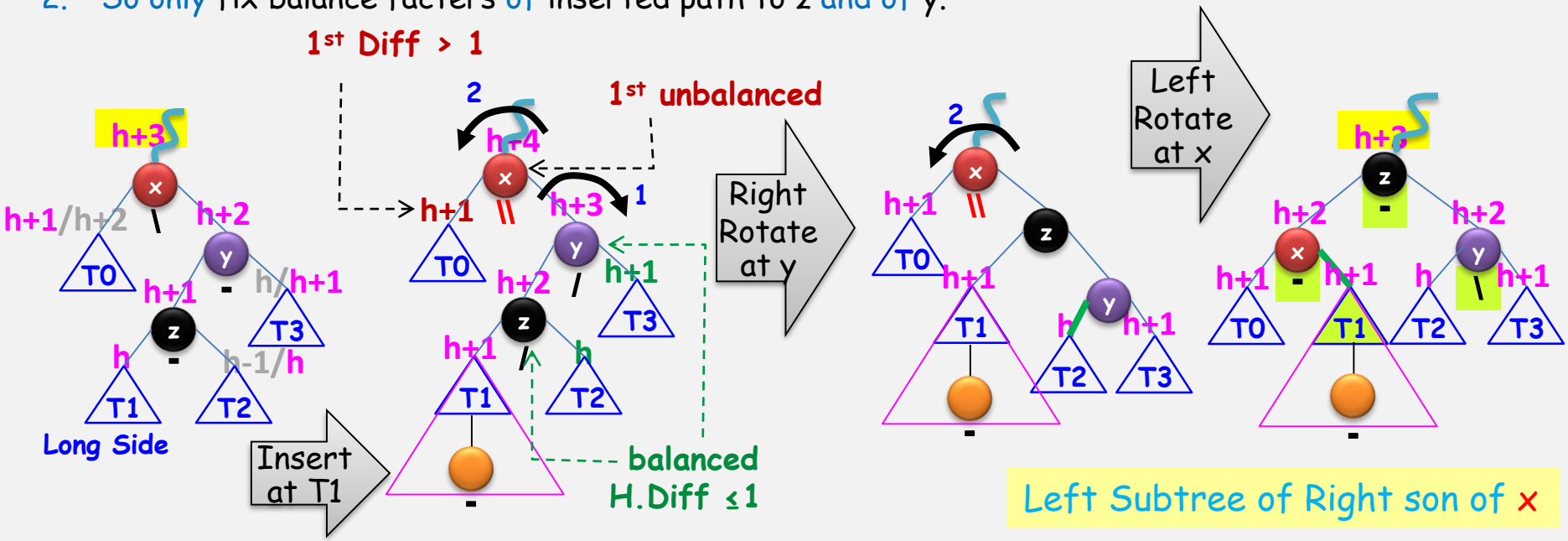
Proof : Rebalancing Area' Height's unchange (Left Subtree of Right Son of x)

Insert at a long side **T1**

x = 1st unbalanced node, y = son of x, z = son of y. Before insertion :

- height(T2) = ? When height(T1) = h.
  = h-1 or h ∵ heightDiff (T1,T2) ≤1 & ≠ h+1 ∵ T2 : short side, T1 : long side
  = h ∵ After inserting , z : balanced ∴ heightDiff ≤1
- height(T3) = h+1 ∵ After inserting , y : balanced
- height(T0) = h+1 ∵ After inserting , x : unbalanced

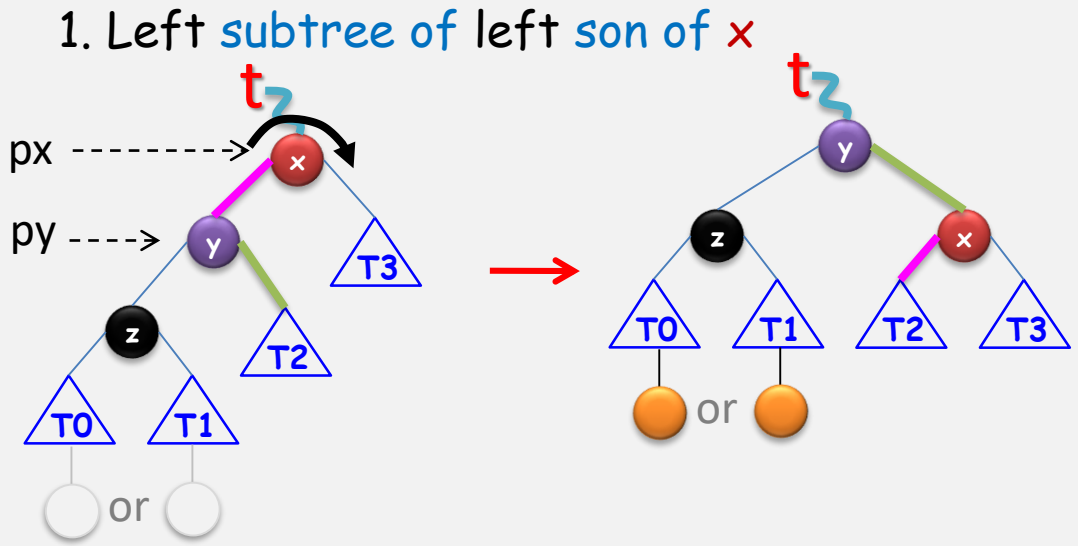
1. Rebalancing does not change heights of the subtree → ie. Nor of root.
2. So only fix balance facters of inserted path to z and of y.



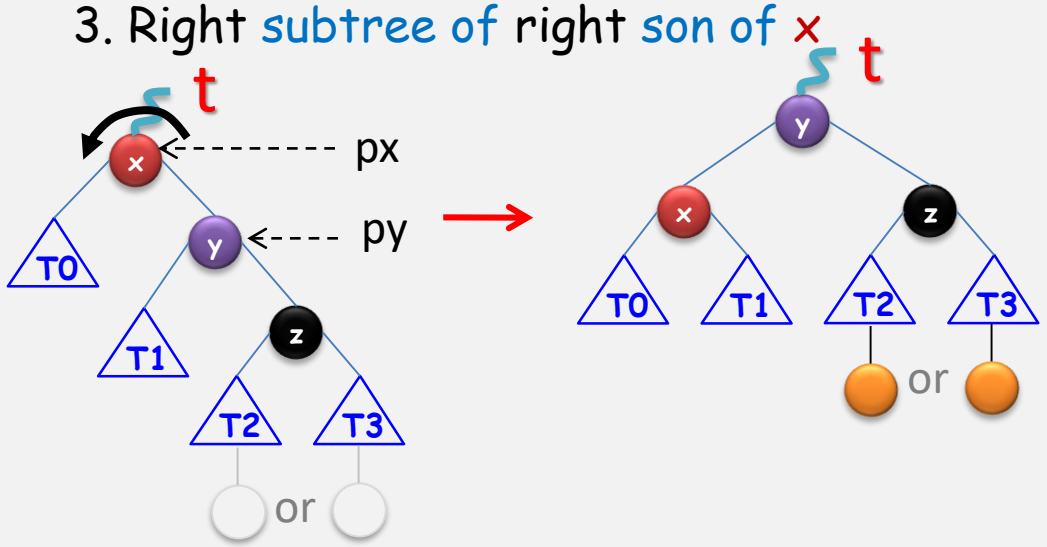
Left Subtree of Right son of x

Single Right Rotation & Single Left Rotation Algorithms

```
// Single Right Rotation
// LeftLeftRotate(t)
LeftLeftRotate(node *px)
    node* py = px.left  //px->left
    px.left = py.right
    py.right = px
    return py
```



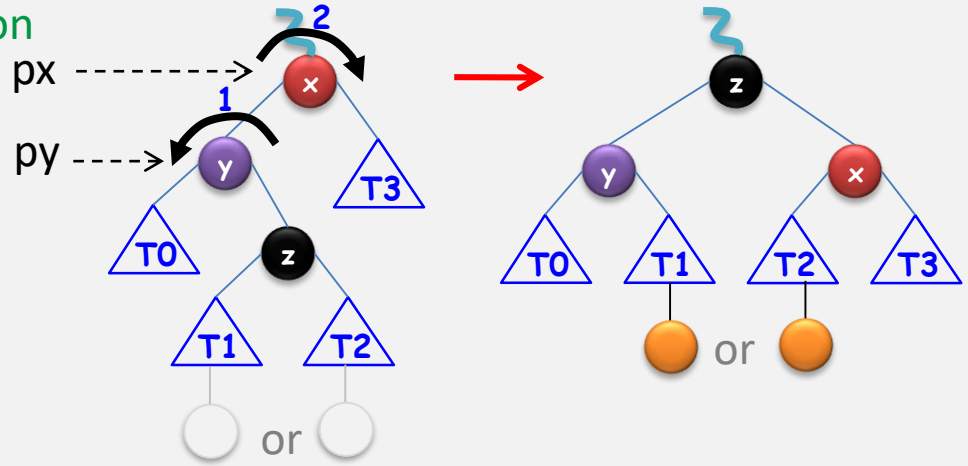

```
// Single Left Rotation
// RightRightRotate(t)
RightRightRotate(node *px)
    node* py = px.right
    px.right= py.left
    py.left = px
    return py
```



Double Rotations Algorithms

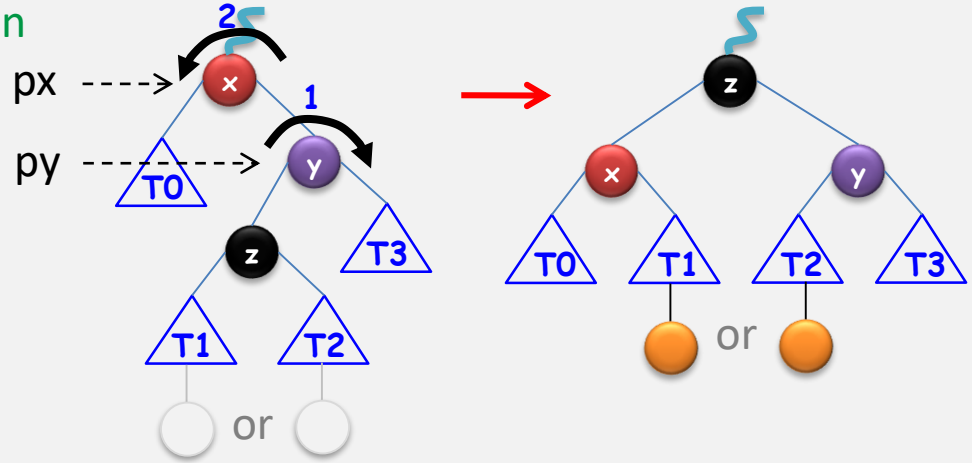
```
// Double Rotation :
// Case 2 : Left Rotation then Right Rotation
RightLeftRotate(node *px)
        node* py = px.left
        px.left = RightRightRotate(py)
        py = LeftLeftRotate(px)
        return py
```

2. Right subtree of left son of x




```
// Double Rotation :
// Case 4 : Right Rotation then Left Rotation
LeftRightRotate (node *px)
        node *py = px.right
        px.right = LeftLeftRotate(py)
        py = RightRightRotate(px)
        return py
```

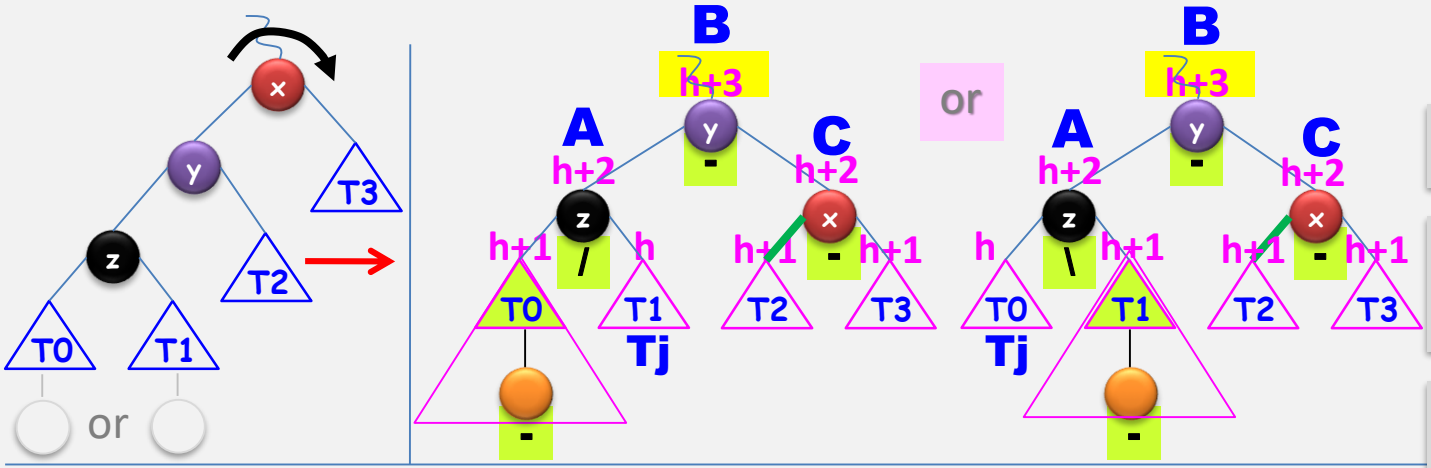
4. Left subtree of right son of x



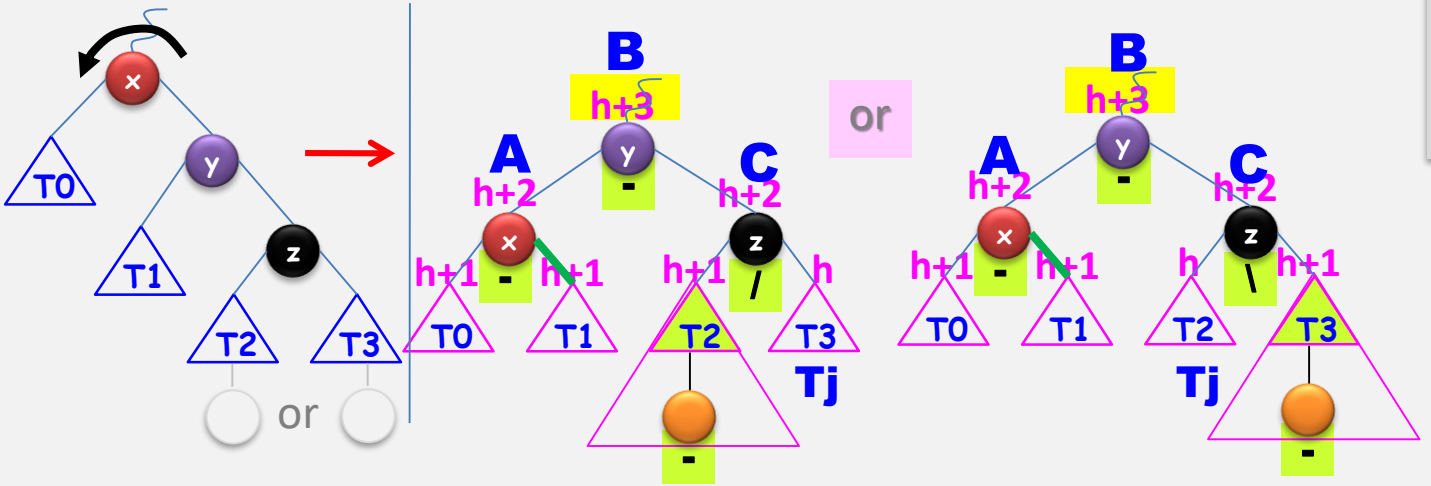
Single Rotation in AVL

x = 1st unbalanced node, y = son of x, z = son of y **A < B < C** **T0<T1<T2<T3**

1. Left subtree of left son of x



2. Right subtree of right son of x



สรุปรูปผลลัพธ์

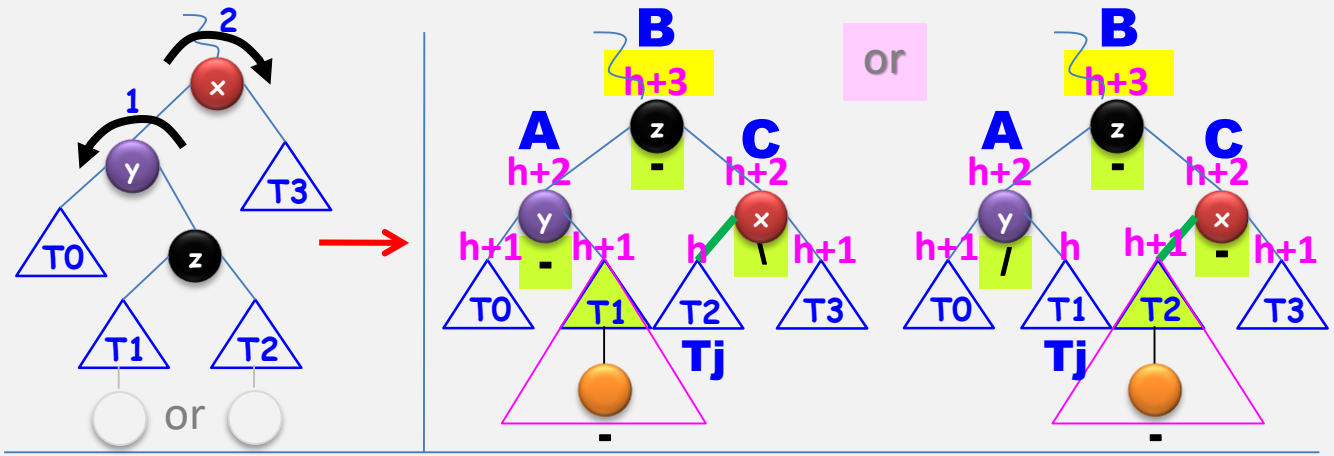
B ตัวค่ากลาง เป็น root ใหม่ เสมอ

- **Height(Tj)** = h ตัวเดียว **Tj** คือตัวที่คู่กับ **Ti** ที่ insert node ใหม่เข้าไป
- นอกนั้น height(T) = h+1 หมด

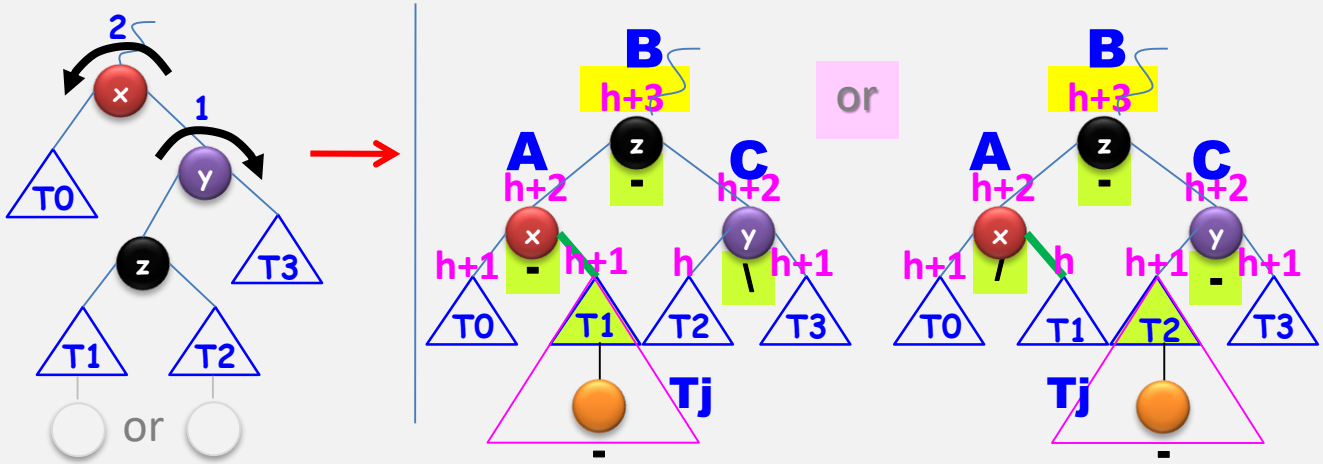
Double Rotations in AVL

x = 1st unbalanced node, y = son of x, z = son of y **A < B < C** **T0<T1<T2<T3**

1. Right subtree of left son of x



3. Left subtree of right son of x



สรุปรูปผลลัพธ์

B ตัวค่ากลาง เป็น root ใหม่ เสมอ

- **Height(Tj)** = h ตัวเดียว **Tj** คือตัวที่คู่กับ **Ti** ที่ insert node ใหม่เข้าไป
- นอกนั้น height(T) = h+1 หมด

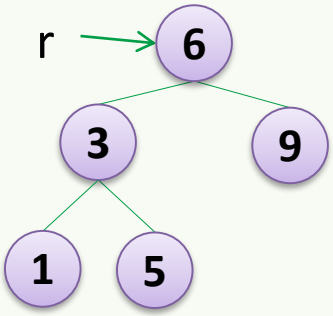
Tree 2

3. AVL Tree
   Height Balanced Tree
4. Which Representions ?
5. n-ary Tree
6. Generic Tree
7. Multiway Search Tree
8. Top Down Tree
9. B-Tree

1. Tree Definitions
2. Binary Tree
   - Traversals
   - Binary Search Tree
   - Representations
   - Application : Expression Tree

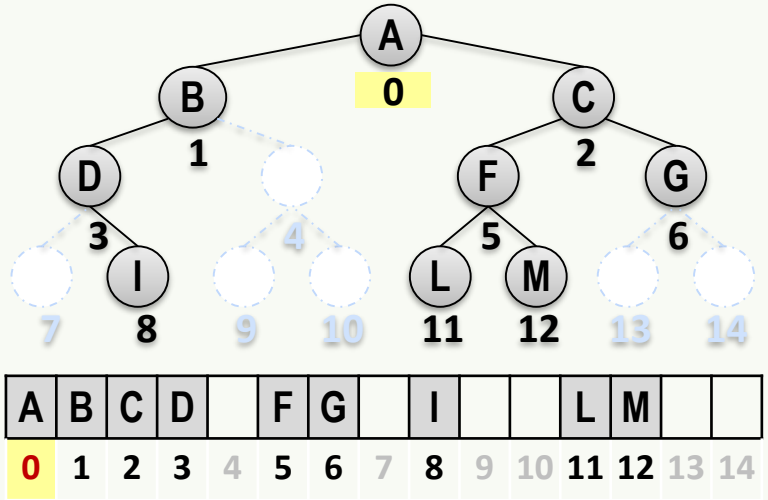
Binary Tree Representations

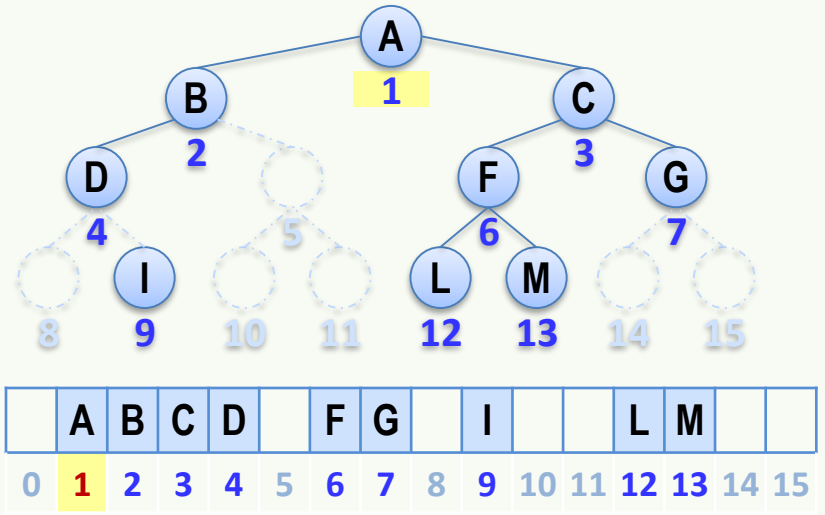
1. Dynamic



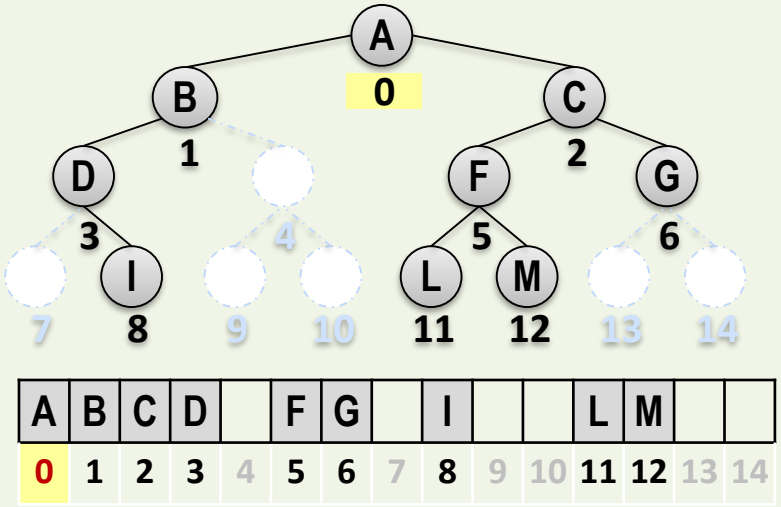

```
class node:

    def __init__(self,  data, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right
```

2. Sequential (Implicit) Array



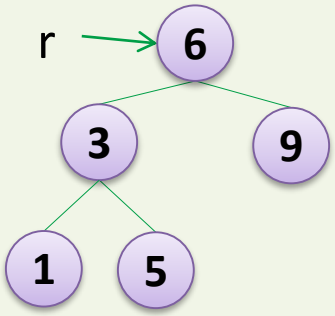


Sequential Array (Implicit Array)



- ง่าย
- ต้องกะขนาด array
- เพิ่มขึ้นอีก 1 level # ที่เพิ่มมากกว่าที่มีอยู่เดิมอยู่
  (คิดเต็มที่ full binary tree)
- ถ้ากะขนาด array ถูก และ tree ใช้พื้นที่ส่วนใหญ่ของ array เช่น almost complete binary tree) จะ save space เพราะไม่ต้องมีฟิลด์ left, right, father

Dynamic



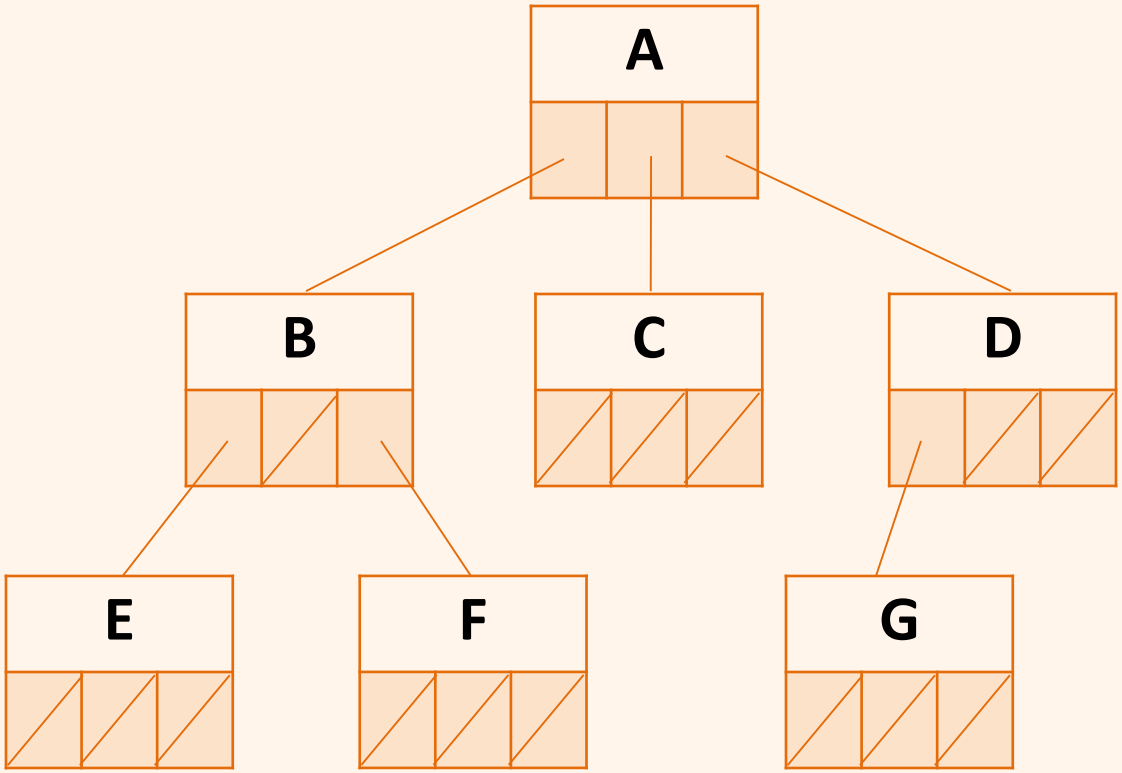
- จำนวน node ถูกจำกัดโดย memory

Tree 2

3. AVL Tree
   Height Balanced Tree
4. Which Representions ?
5. n-ary Tree
6. Generic Tree
7. Multiway Search Tree
8. Top Down Tree
9. B-Tree

1. Tree Definitions
2. Binary Tree
   - Traversals
   - Binary Search Tree
   - Representations
   - Application : Expression Tree

n-ary Tree แต่ละ node มีลูกได้อย่างมากที่สุด n ตัว



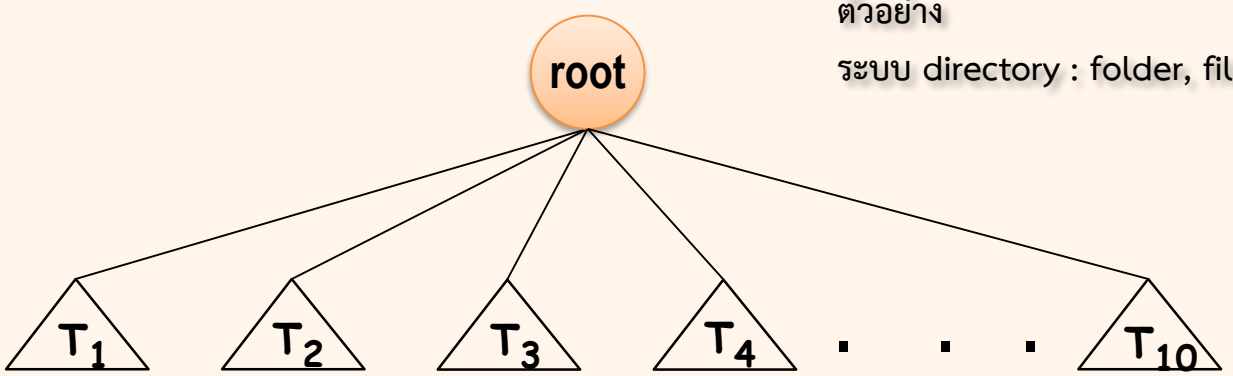
Ternary Tree n = 3

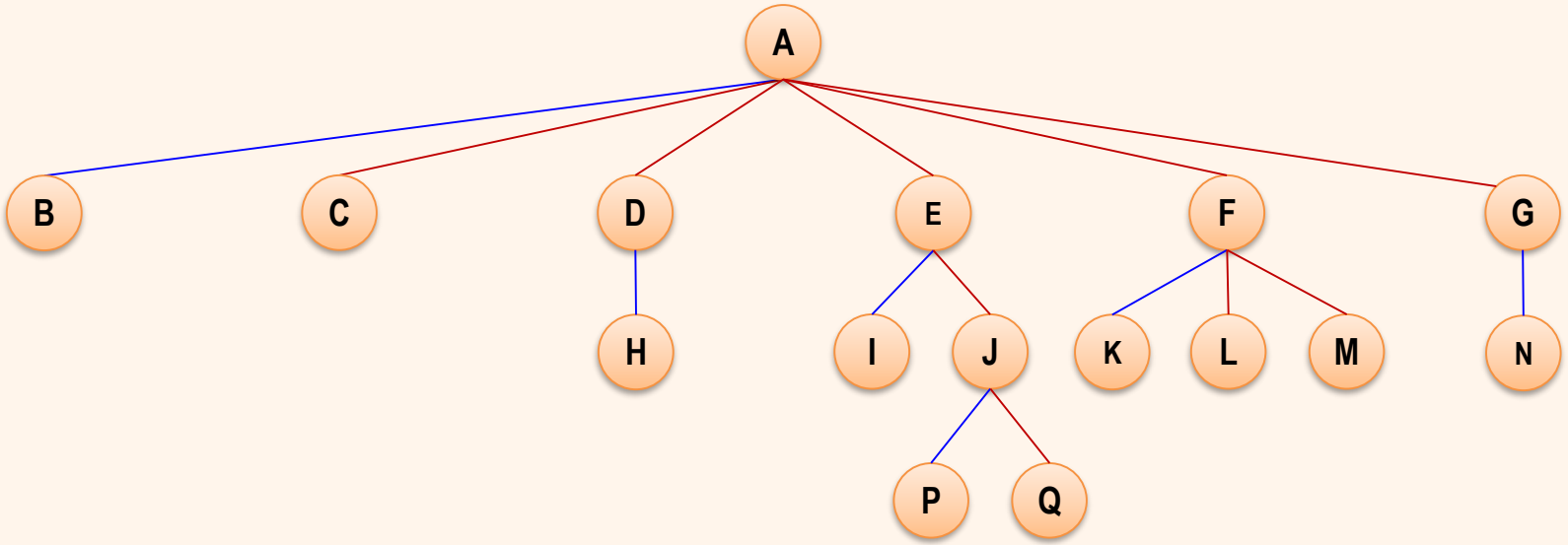
Generic Tree

Generic Tree : แต่ละ node มีลูกได้ไม่จำกัดจำนวน

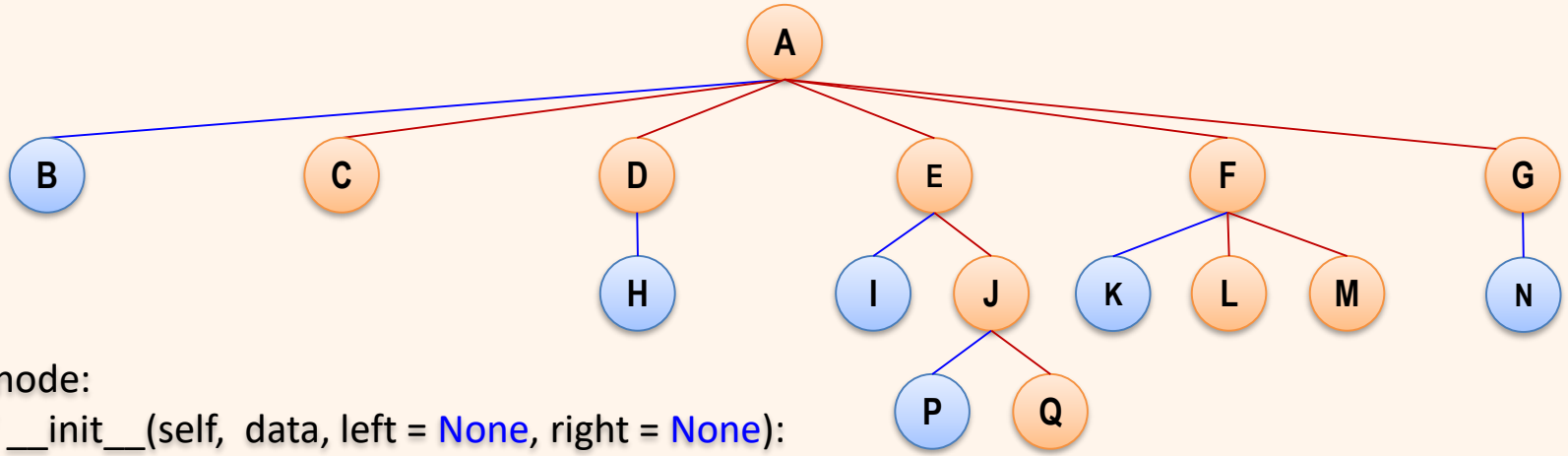
ตัวอย่าง

ระบบ directory : folder, file

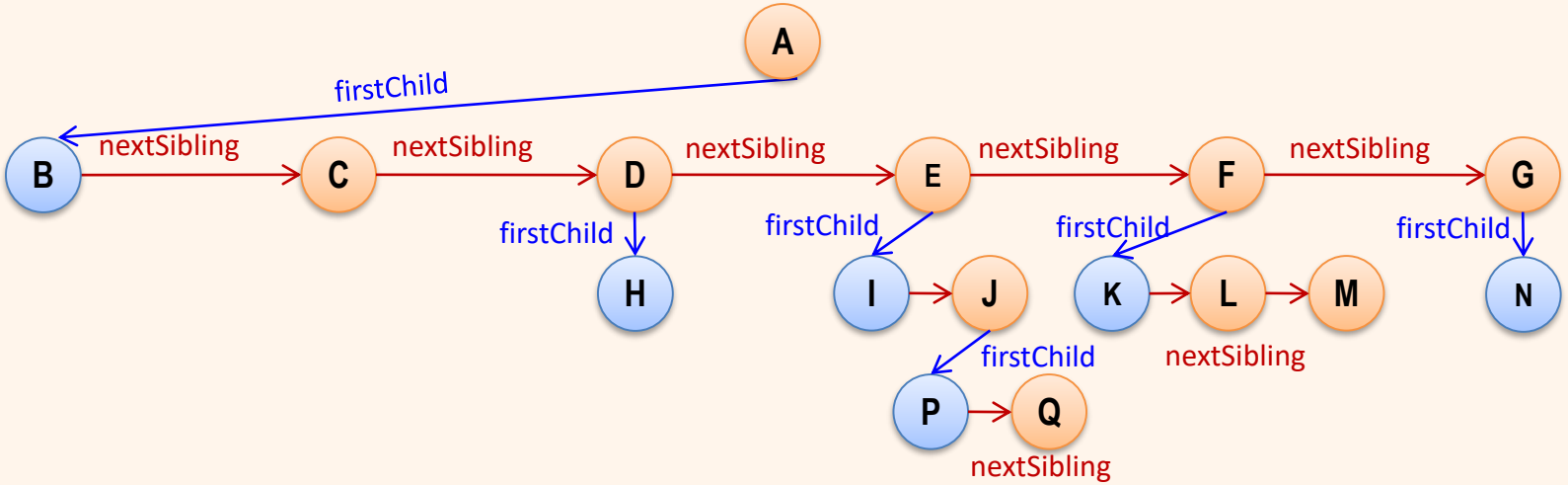




Implementation of Generic Tree




```
class node:
    def __init__(self,  data, left = None, right = None):
        self.data = data
        self.left = firstChild
        self.right = nextSibling
```



Tree 2

3. AVL Tree
   Height Balanced Tree
4. Which Representions ?
5. n-ary Tree
6. Generic Tree
7. Multiway Search Tree
   - ❖ Top Down Tree
   - ❖ B-Tree

1. Tree Definitions
2. Binary Tree
   - – Traversals
   - – Binary Search Tree
   - – Representations
   - – Application : Expression Tree

Multiway Search Tree

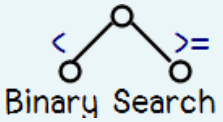
Multiway Tree order n : มีลูกได้มากที่สุด n ตัว (0, 1,... or n subtrees) (#max. sons = n)

For order n :
- #max. sons = n
- #max. keys = n-1

Multiway Search Tree : for every key K

left decendents ≤ K

right decendents > K

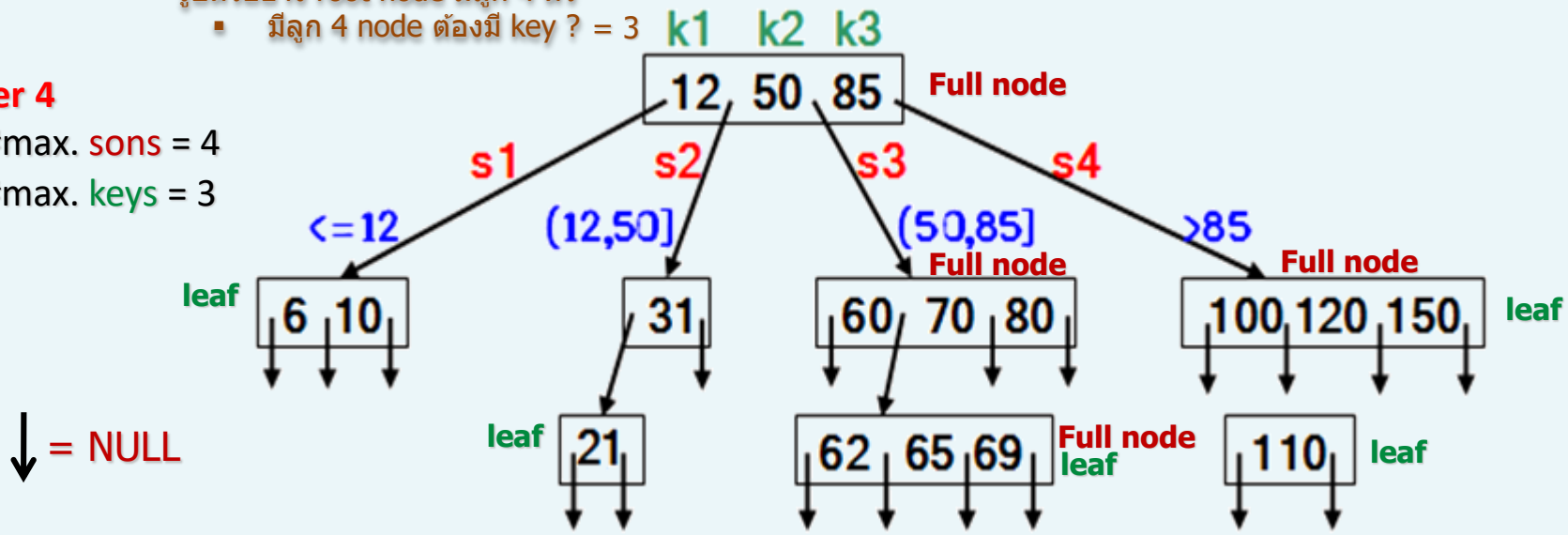


leaf : node ที่ไม่มีลูกเลย

Full node : node ที่มีจำนวน key เต็มที่

รูปตัวอย่าง root node มีลูก 4 ตัว
- มีลูก 4 node ต้องมี key ? = 3

Order 4
- #max. sons = 4
- #max. keys = 3



Tree 2

3. AVL Tree
   Height Balanced Tree
4. Which Representions ?
5. n-ary Tree
6. Generic Tree
7. Multiway Search Tree
   - ❖ Top Down Tree
   - ❖ B-Tree

1. Tree Definitions
2. Binary Tree
   - – Traversals
   - – Binary Search Tree
   - – Representations
   - – Application : Expression Tree

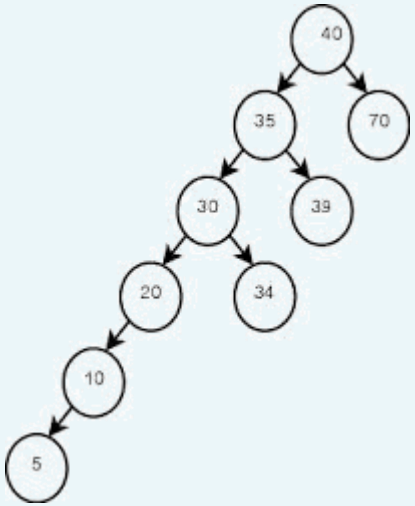
Top Down Tree

Top Down Tree : Non full node must be leaf.
Node ที่ไม่เต็มจะเป็น leaf ได้เท่านั้น เป็น internal node ไม่ได้

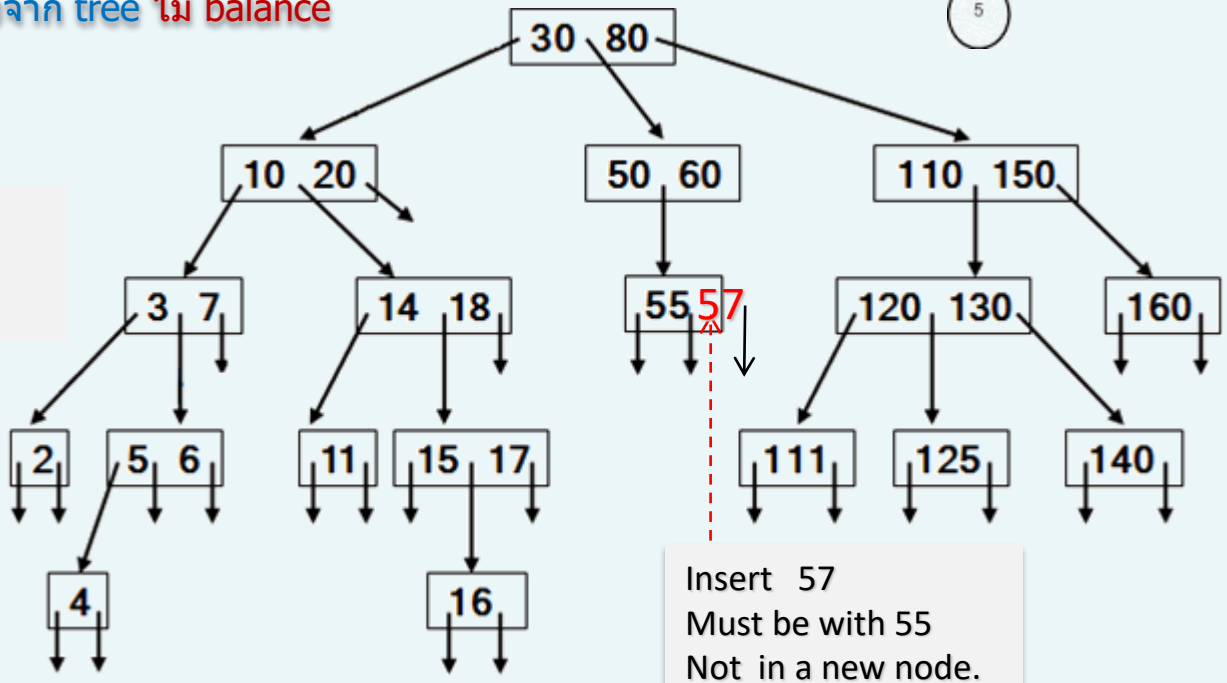
- Fill up the existing nodes before creating a new node !
  เติม node ให้เต็มก่อนค่อยแตก node ใหม่

แนวคิด : มี nodes ให้น้อยที่สุด เพราะ อยากให้ height สั้น

แม้ว่าพยามทำให้ node น้อย โดยเติมให้เต็มก็ตาม
height อาจยาวได้เนื่องจาก tree ไม่ balance



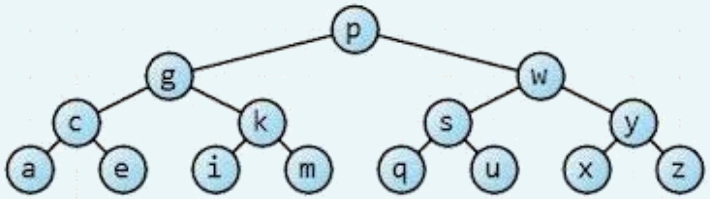
Order 3
- #max. sons = 3
- #max. keys = 2

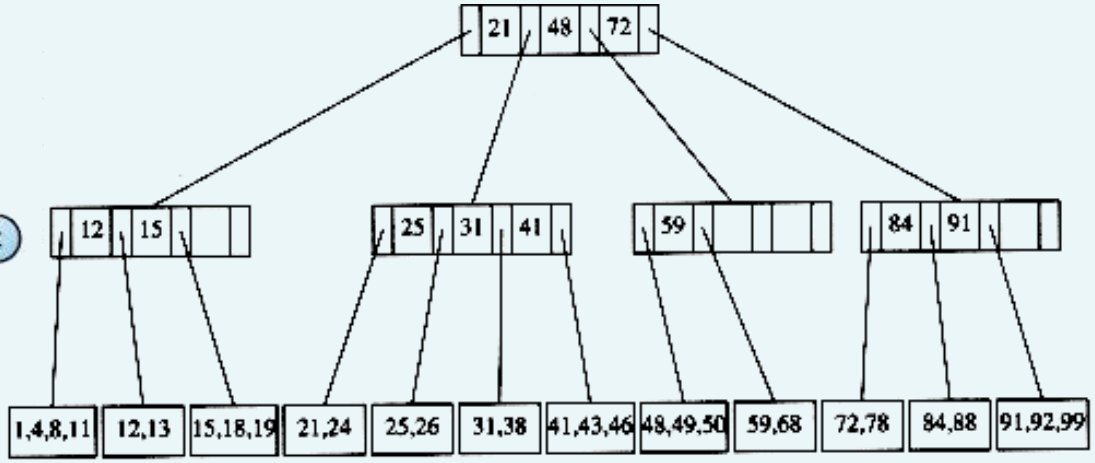


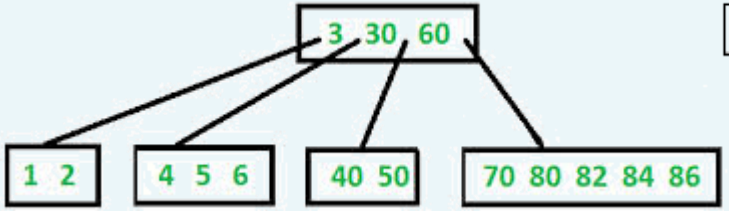
Insert 57
Must be with 55
Not in a new node.

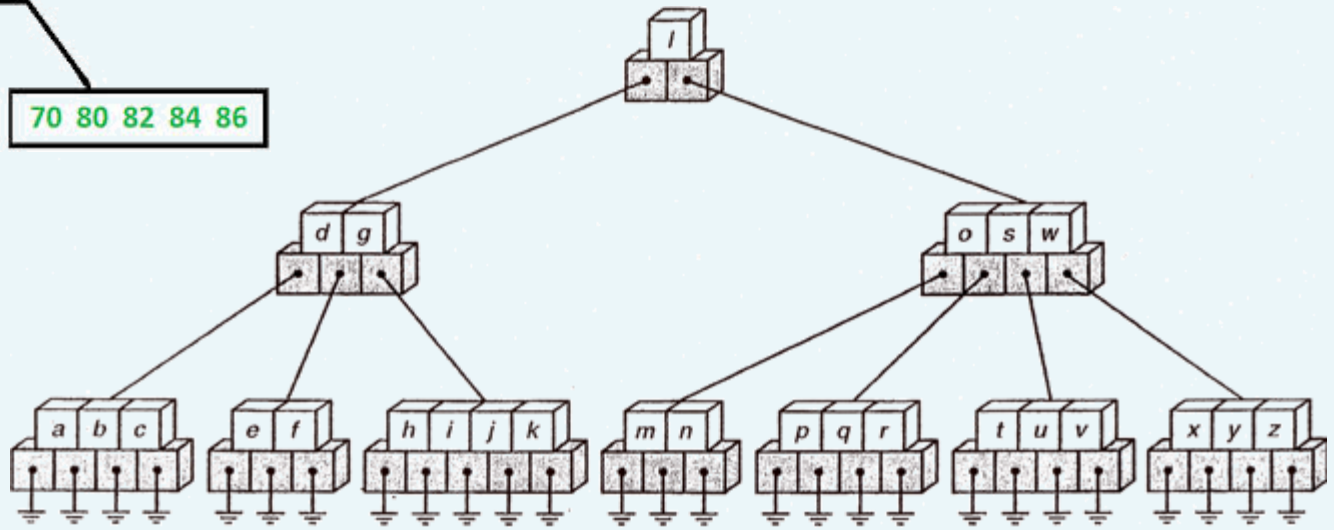
Balanced Tree

Balanced Tree : every leaf node is at the same level
(Perfect balance , perfectly height-balanced)









Tree 2

3. AVL Tree
   Height Balanced Tree
4. Which Representions ?
5. n-ary Tree
6. Generic Tree
7. Multiway Search Tree
   - ❖ Top Down Tree
   - ❖ B-Tree

1. Tree Definitions
2. Binary Tree
   - – Traversals
   - – Binary Search Tree
   - – Representations
   - – Application : Expression Tree

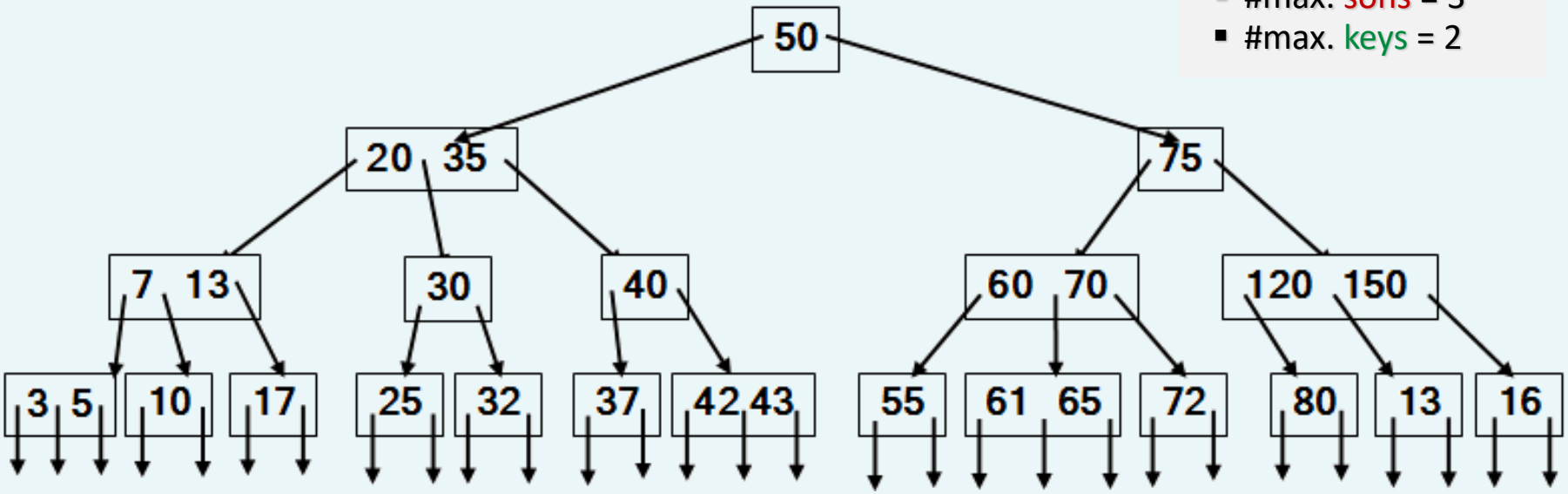
B-tree order n

1. Multi-way search tree order n : #max son = n, #max keys = n-1
2. Balanced tree : ทุก leaf อยู่ใน level เดียวกัน

   (Perfect balance , perfectly height-balanced)
3. => next page

Order 3

- #max. sons = 3
- #max. keys = 2



B-tree order n

1. Multi-way search tree order n

   #max son = n, #max keys = n-1

2. Balanced tree

3. ทุก node มีจำนวน non-empty subtree (มีลูก)อย่างน้อยที่สุด เท่ากับครึ่งหนึ่งของ n (มีน้อยกว่านี้ไม่ได้)

   (Half Full) ยกเว้น root node ไม่ต้อง half full

   #min sons = $\lceil n/2 \rceil$

EX.

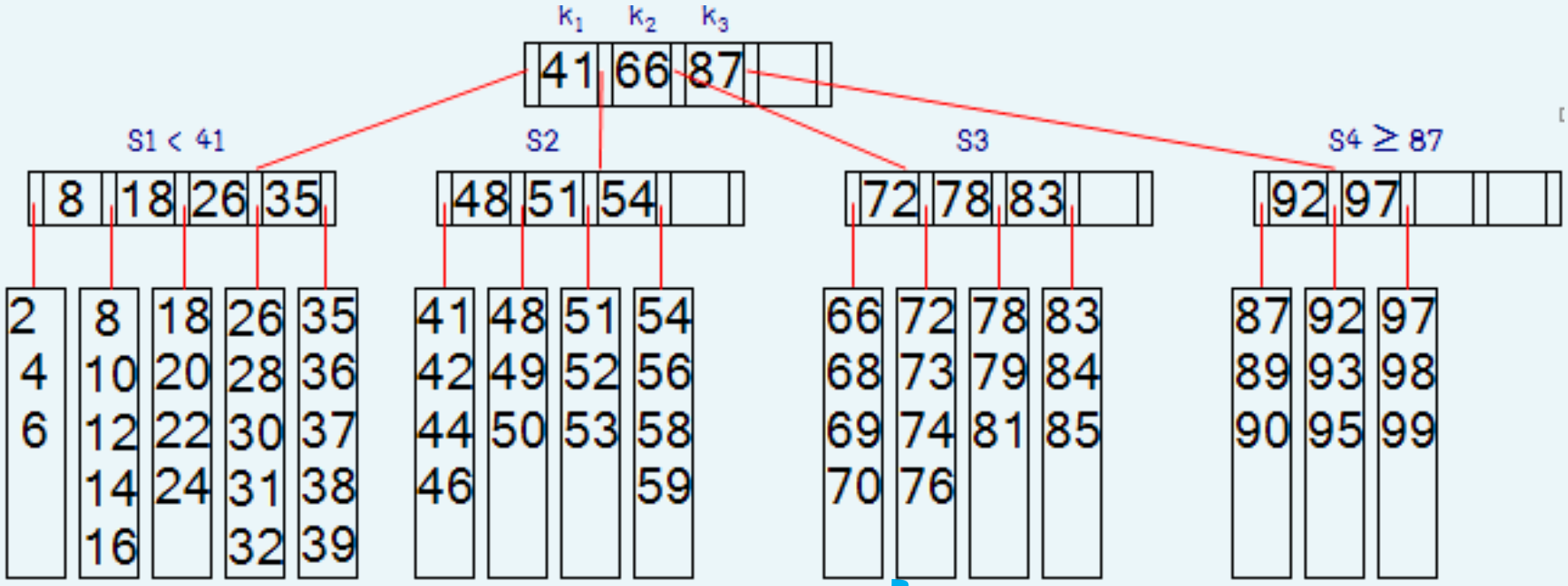
1. order 2 : #max sons =2 , #min sons = $\lceil 2/2 \rceil = 1$

   ∴ Binary ที่เป็น (almost) complete binary tree จะเป็น B-tree

2 order 5 : #max sons = 5 , #min sons = $\lceil 5/2 \rceil = \lceil 2.5 \rceil = 3$

3 order 201 : #max sons = 201 , #min sons = $\lceil 201/2 \rceil = 101$

B-Tree Variation



order 5 #max son = 5, #max keys = 4

- Half Full #min sons = $\lceil 5/2 \rceil = \lceil 2.5 \rceil = 3$ (ยกเว้น root node)

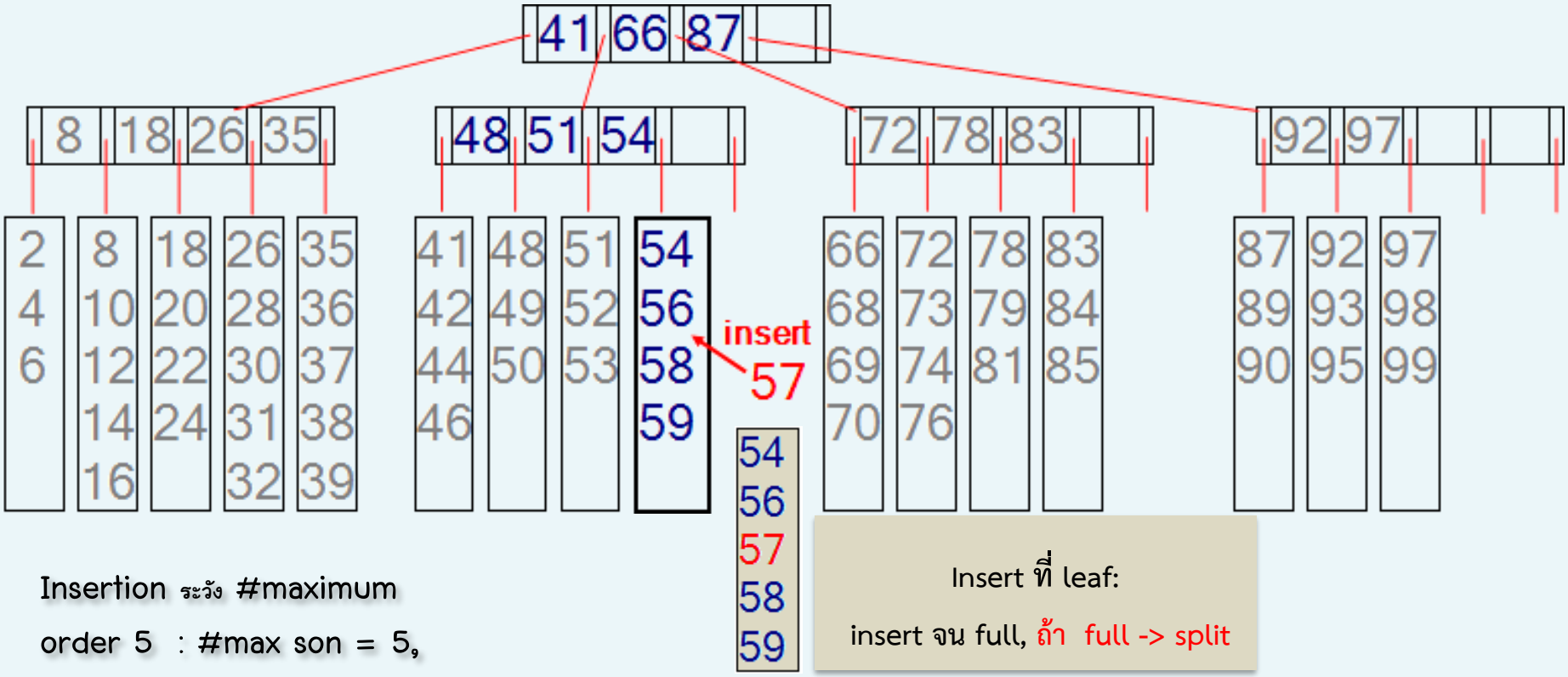
B-tree -> variations

- internal nodes : เก็บเฉพาะ key เพื่อประหยัดพื้นที่, order กำหนดจำนวนลูก
- external nodes : เก็บ data ทั้ง record

ค่า L กำหนดจำนวน record/node ของ leaf

ถ้า L = 5 : #max data = 5, #min data = $\lceil 5/2 \rceil = \lceil 2.5 \rceil = 3$

B-Tree (Variation) Insertion



Insertion ระวัง #maximum

order 5 : #max son = 5,

Internal : **#max keys = 4,**

#min sons = $\lceil 5/2 \rceil$ = 3

Leaf L = 5: **#max data = 5,**

#min data = $\lceil 5/2 \rceil$ = 3

Insert ที่ leaf:

insert จน full, ถ้า full -> split

จะ insert ต้องระวังไม่ให้เกินจำนวนที่กำหนด (max)

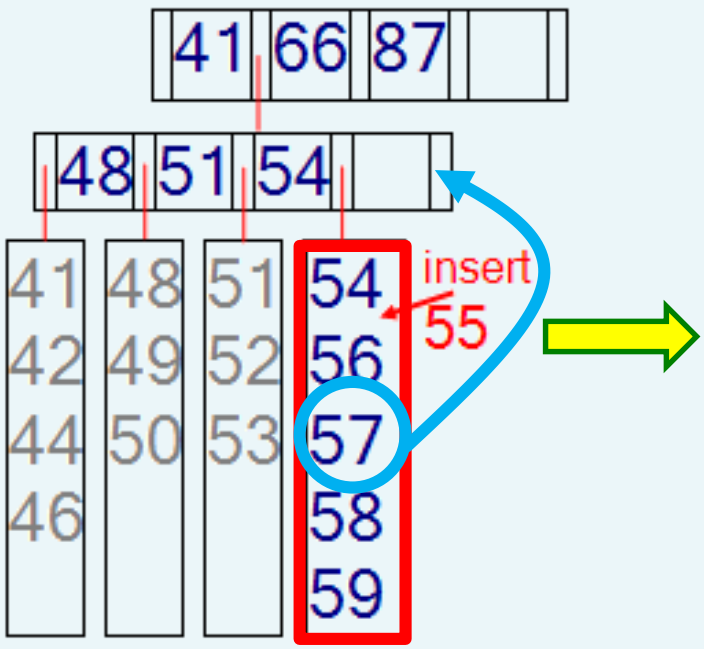
-> Insert ที่ leaf ดูค่า L

Leaf L = 5: #max data = 5

B-Tree (Variation) Insertion

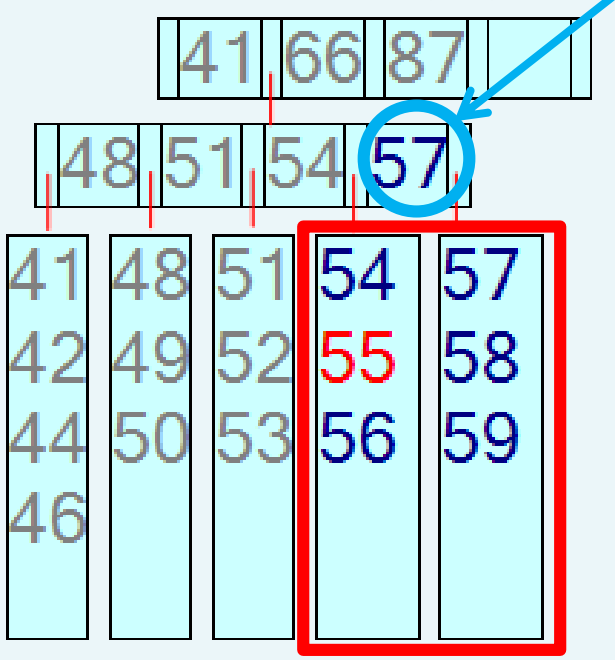
order 5, #max key = 5-1=4
L=5 #leaf max data = 5

order 5 : internal node มี #max son = 5
: **#max keys = 4,**
not full -> can insert



Insert 55
Insert ที่ leaf จน full :
L = 5: #max data = 5

Full

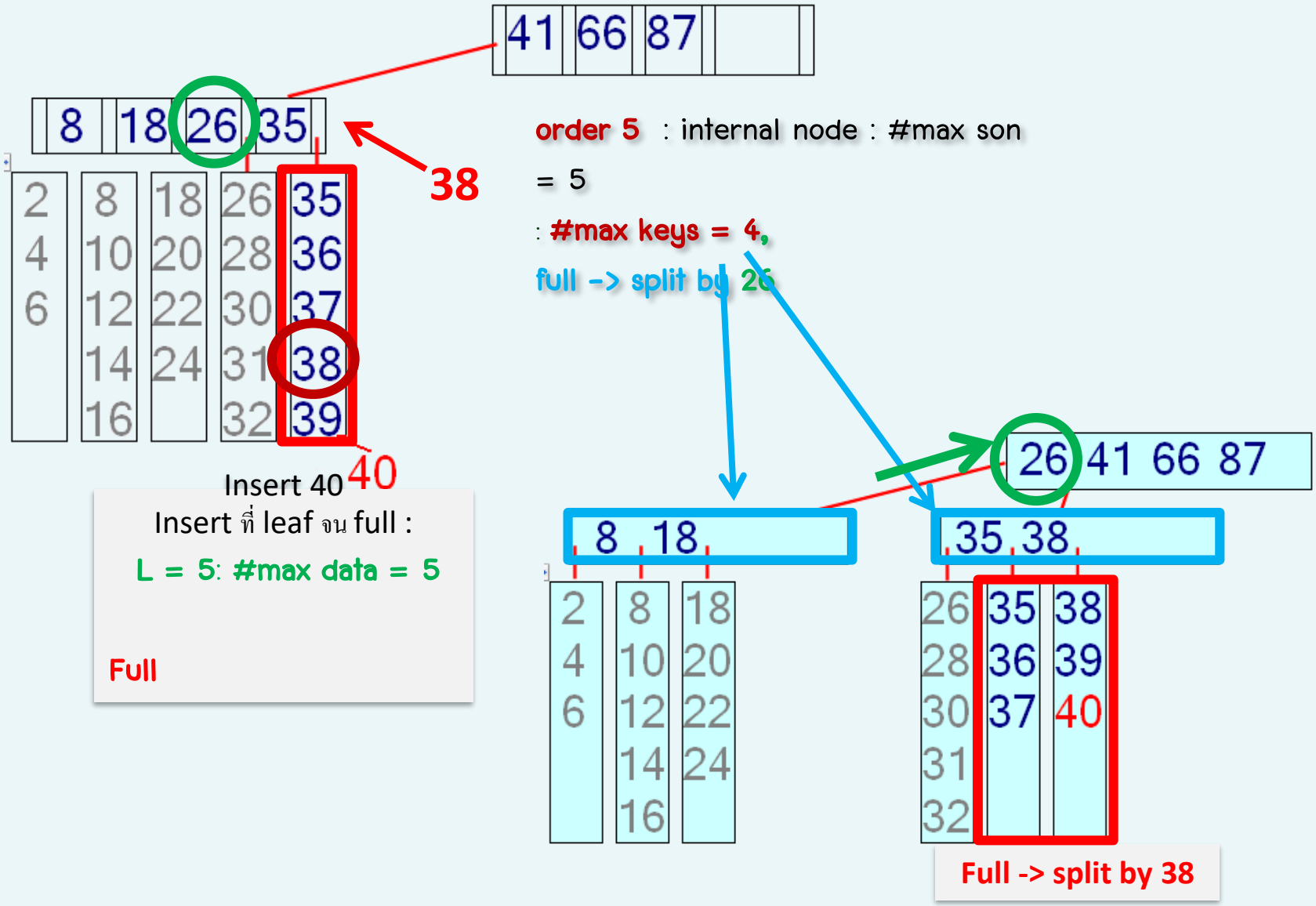


key 57 splits 2 nodes
key 57 ต้องถูก insert ใน father node

Full -> split half-half ที่ 57

B-Tree (Variation) Insertion

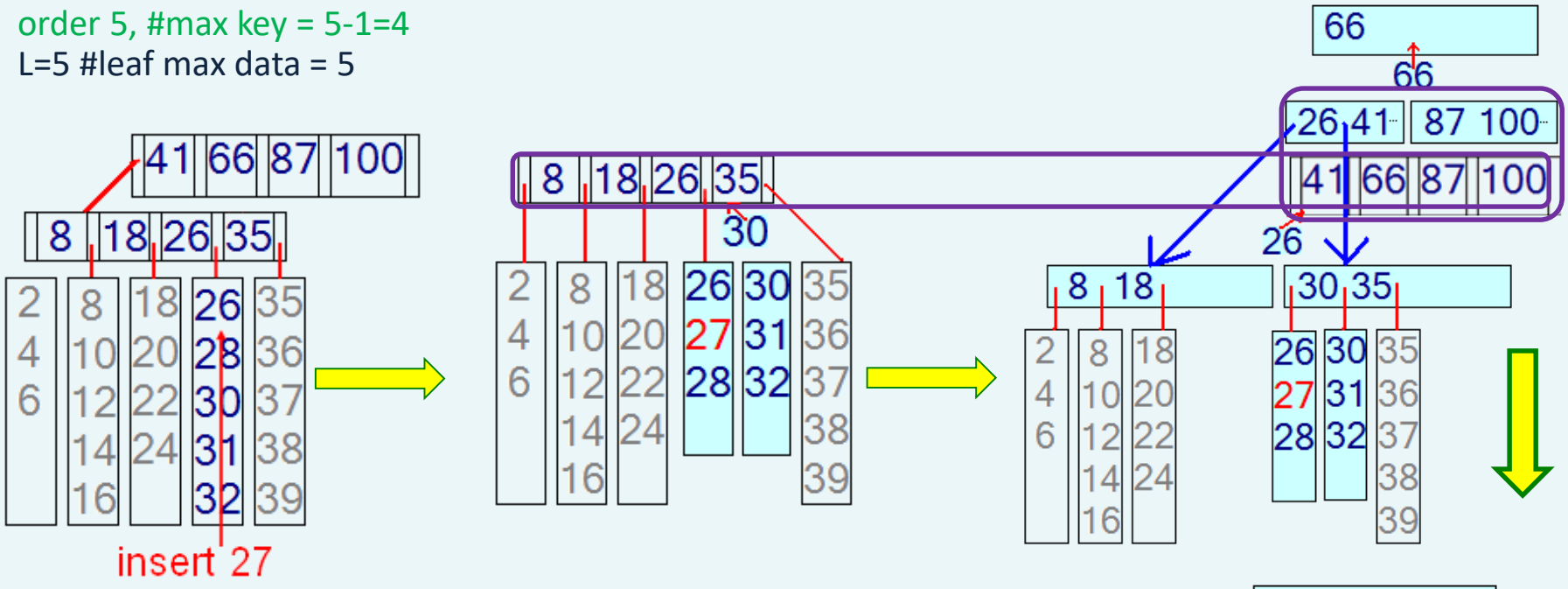
Insert at leaf:
until full, if full -> split

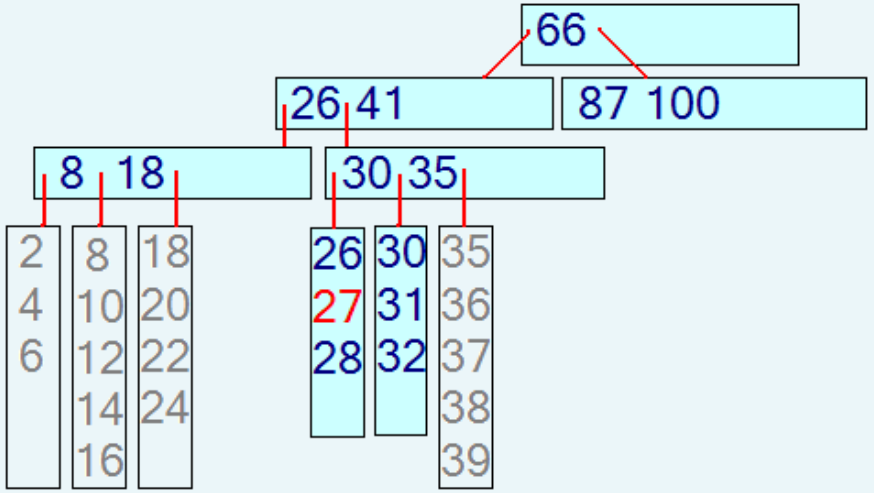


B-Tree (Variation) Insertion

Insert at leaf:
until full, if full -> split

order 5, #max key = 5-1=4
L=5 #leaf max data = 5





Insertion ระวัง #maximum

order 5 : #max son = 5,

Internal : **#max keys = 4,**

#min sons = $\lceil 5/2 \rceil = 3$

Leaf L = 5: **#max data = 5,**

#min data = $\lceil 5/2 \rceil = 3$

B-Tree (Variation) Deletion

Deletion consider

#minimum

Internal order 5 :

#min sons = $\lceil 5/2 \rceil = 3$

#min keys = 2

Leaf L = 5:

#max data = 5,

#min data = $\lceil 5/2 \rceil$ = 3

Delete at leaf:
until low, if low -> borrow

Borrowing

1. ยืมพี่น้อง(ผ่านพ่อ)
2. ยืมพ่อ ต้อง consolidate

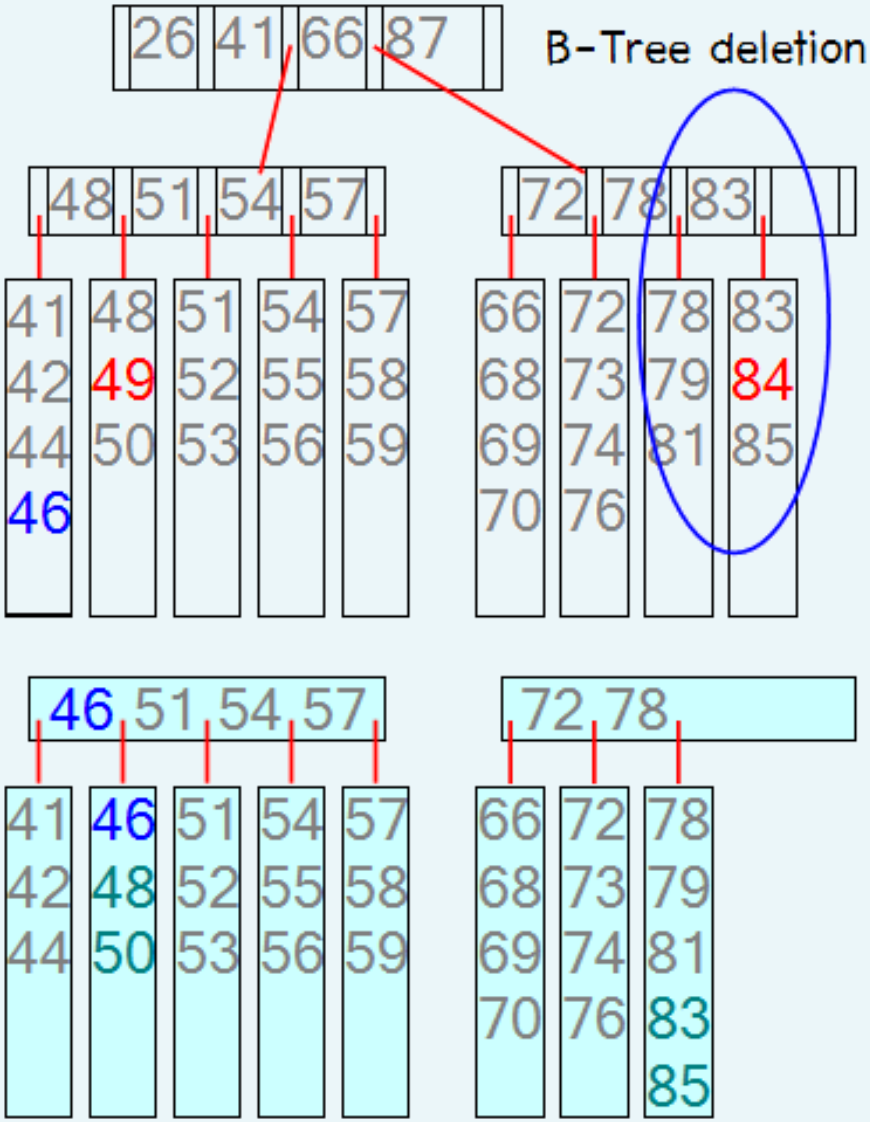
พ่อไม่มี พ่อยืม

-พี่น้องพ่อ(ผ่านปู่)

. . .

B-Tree (Variation) Deletion

Delete at leaf:
until low, if low -> borrow



Deletion consider #minimum

Internal order 5 : #min sons = $\lceil 5/2 \rceil = 3$

#min keys = 2

Leaf L = 5: #max data = 5,

#min data = $\lceil 5/2 \rceil = 3$

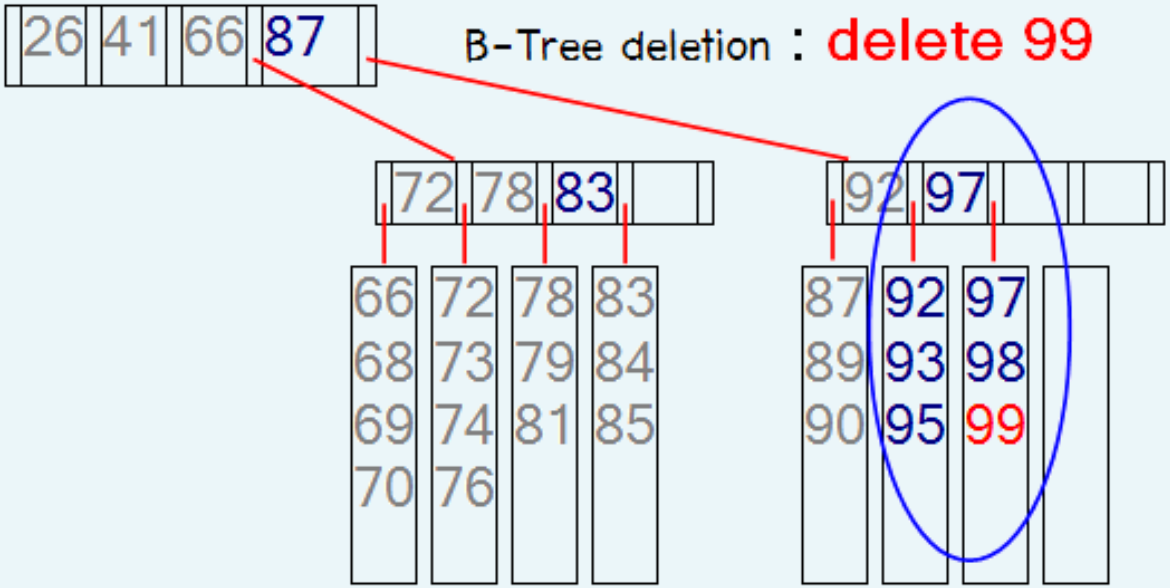
Borrowing

1. ยืมพี่น้อง(ผ่านพ่อ) เช่น del 49 เอา 46 ของพี่มา
2. ยืมพ่อ ต้อง consolidate เช่น del 84 พี่น้องไม่มีให้ยืม ยืมพ่อ โดยรวม consolidate กับพ่อ

ถ้าพ่อไม่มี พ่อยืม
-พี่น้องพ่อ(ผ่านปู่)
. . .

B-Tree (Variation) Deletion

Delete at leaf:
until low, if low -> borrow



Deletion consider #minimum

Internal order 5 :

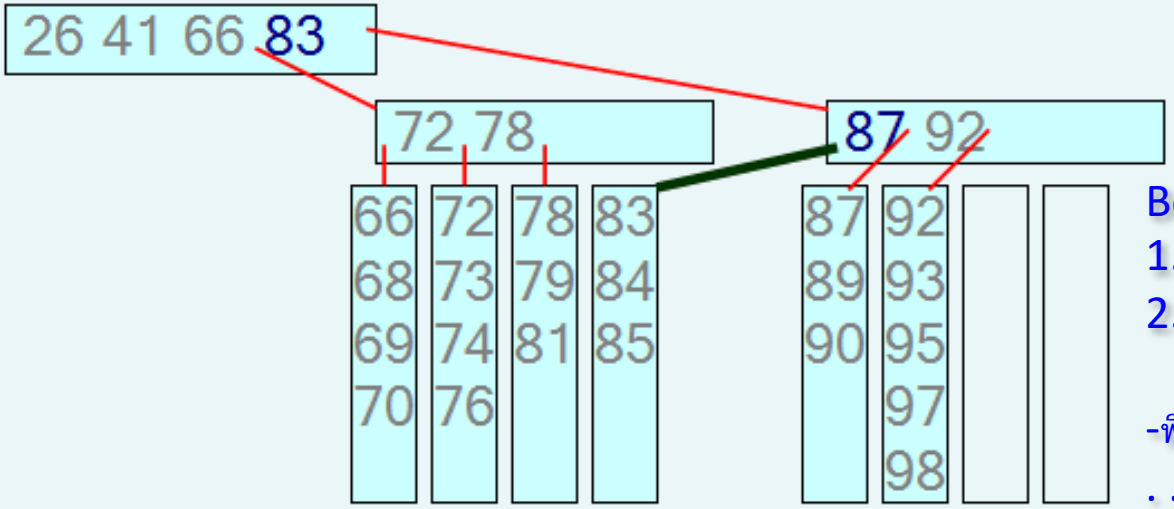
#min sons = $\lceil 5/2 \rceil = 3$

#min keys = 2

Leaf L = 5:

#max data = 5,

#min data = $\lceil 5/2 \rceil = 3$



Borrowing

1. ยืมพี่น้อง(ผ่านพ่อ)
2. ยืมพ่อ ต้อง consolidate
   พ่อไม่มี พ่อยืม

-พี่น้องพ่อ(ผ่านปู่)

. . .