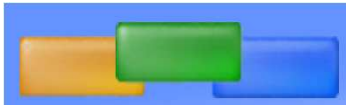
# Cây cân bằng Red Black và AA

- Review
- Giới thiệu
- Cây Đỏ – Đen (Red Black Tree)
- AA – Tree



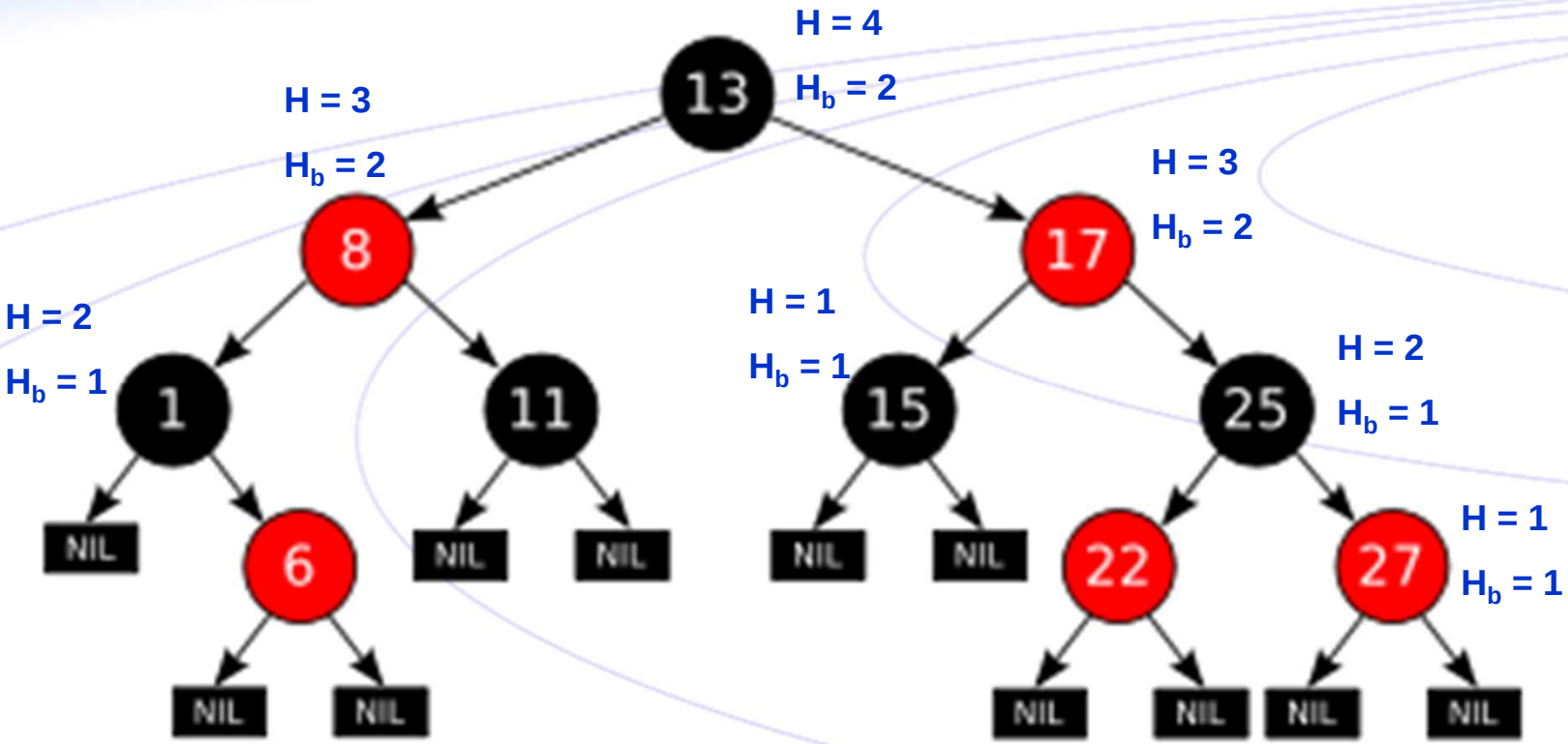
# Red Black Tree

- Định nghĩa
- Cấu trúc lưu trữ
- Các tính chất
- Các thao tác cơ bản
- Đánh giá

# Red Black Tree (tt)

- Định nghĩa: Red-Black tree là một cây nhị phân tìm kiếm (BST) tuân thủ các quy tắc sau:
  - [1] Mọi node phải là đỏ hoặc đen
  - [2] Node gốc là đen
  - [3] Các node ngoài (external node; NULL node) mặc định là những node đen
  - [4] Nếu một node là đỏ, những node con của nó phải là đen
  - [5] Mọi đường dẫn từ gốc đến node ngoài phải có cùng số lượng node đen

# Red Black Tree (tt)

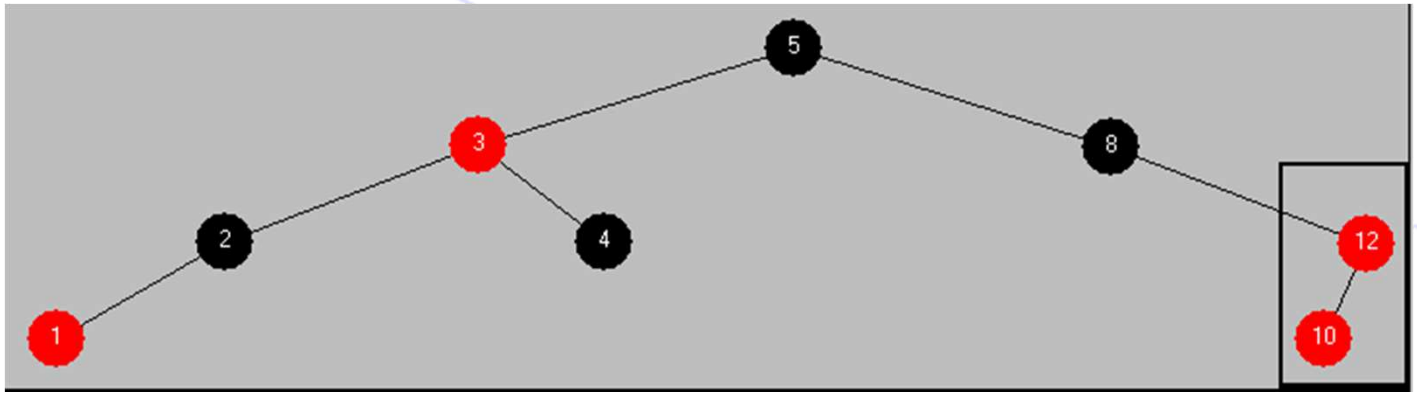



Minh họa Red-Black tree

# Red Black Tree (tt)

- Chiều cao đen (black height – $h_b(x)$): là số node đen trên đường đi từ node $x$ đến node ngoài (không bao gồm $x$)

- Từ quy tắc [4] →không thể tồn tại node cha và node con cùng đỏ. Khi cây đỏ đen vi phạm qui tắc này gọi là hiện tượng xung đột đỏ-đỏ



# Red Black Tree (tt)

- Cấu trúc lưu trữ:
  - Thông tin lưu trữ tại Node (key)
  - Địa chỉ node gốc của cây con bên trái (* pLeft)
  - Địa chỉ node gốc của cây con bên phải (* pRight)
  - Địa chỉ của node cha (* pParent)
  - Thuộc tính màu của node (color)

# Red Black Tree (tt)

```
typedef enum {BLACK, RED} NodeColor;

typedef int DataType;   // Kiểu dữ liệu

typedef struct NodeTag {
      DataType           key;         // Dữ liệu
      NodeColor          color;       // Màu của node

      struct NodeTag   *pLeft;
      struct NodeTag   *pRight;
      struct NodeTag   *pParent;      // Để dễ cài đặt
} RBNode;

typedef struct RBNode* RBTREE;
```

# Red Black Tree (tt)

- Các tính chất:
  - Tính chất 1:

    $h$: chiều cao của cây

    $h_b$: chiều cao đen

    **$h <= 2*h_b$**

  - Tính chất 2: Cây đỏ đen có N node thì

    **$h <= 2*\log_2(N+1)$**

  - Tính chất 3: thời gian tìm kiếm $O(\log_2 N)$

*(Chứng minh tính chất [1] và [2]: bài tập)*

# Red Black Tree (tt)

- Các thao tác cơ bản:
  - Tìm kiếm & duyệt cây: giống BST. Do cây Red-Black cân bằng nên chi phí duyệt cây tốt hơn BST
  - Thêm node mới (insert node)
  - Xóa node (delete node)

# Red Black Tree (tt)

- Insert node:
  - Thực hiện giống như cây BST
  - Node mới thêm luôn luôn có màu đỏ
  - Nếu xảy ra vi phạm qui tắc → điều chỉnh cây

  - *Demo chương trình*

# Red Black Tree (tt)

- Insert node: (tt) những qui tắc có thể bị vi phạm
  - Mọi node phải là đỏ hoặc đen → OK
  - Node gốc là đen → not OK ! Nếu node mới là root
  - Các node ngoài (NULL) phải luôn luôn đen → OK
  - Nếu một node là đỏ, những node con của nó phải là đen → not OK ! vì có thể parent[z] = RED → 2 node liên tiếp màu đỏ
  - Mọi đường dẫn từ gốc đến nút lá phải có cùng số lượng node đen → OK vì không làm thay đổi số node đen

# Red Black Tree (tt)

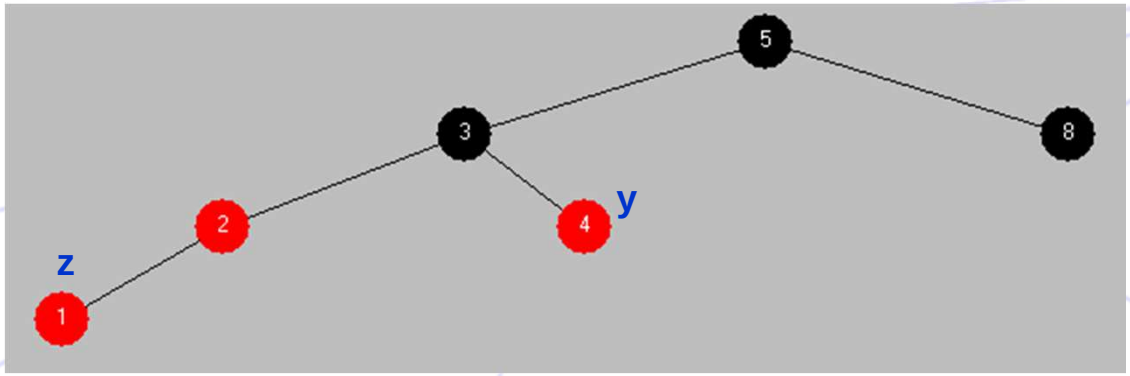
```
RB_Insert_Node(T, z)                    // T: cây; z: node mới
    y ← NULL; x ← root[T];
    while x ≠ NULL {                    // đi đến nút lá
       y ← x                            // y: node cha của x
       if (key[z] < key[x]) x ← left[x];
       else x ← right[x];
    }
    parent[z] ← y;                      // thêm node z vào cây
    if (y == NULL) root[T] ← z;         // là con của node y
    else if (key[z] < key[y]) left[y] ← z;
         else right[y] ← z;
    left[z] ← NULL
    right[z] ← NULL
    color[z] ← RED                      // node mới z có màu đỏ
    RB_Insert_FixUp(T, z)               // điều chỉnh cây
```

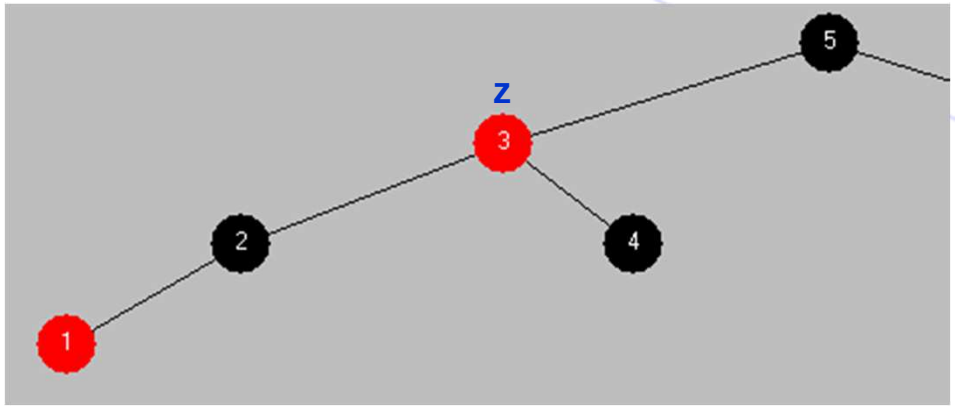
# Red Black Tree (tt)

- Cách thức điều chỉnh cây
  - Phép đảo màu
  - Phép xoay trái (Left-Rotation)
  - Phép xoay phải (Right-Rotation)

# Red Black Tree (tt)

- Phép đảo màu

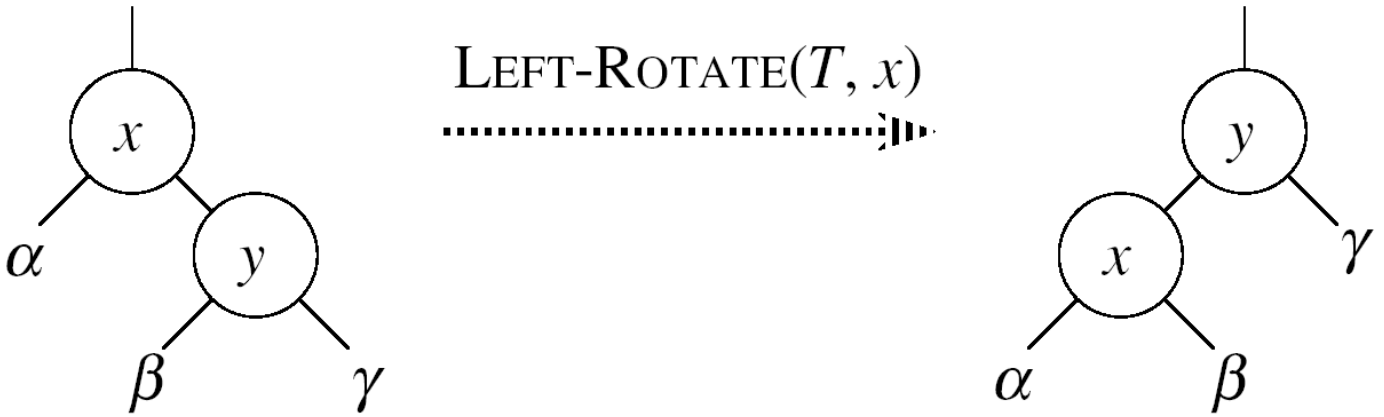


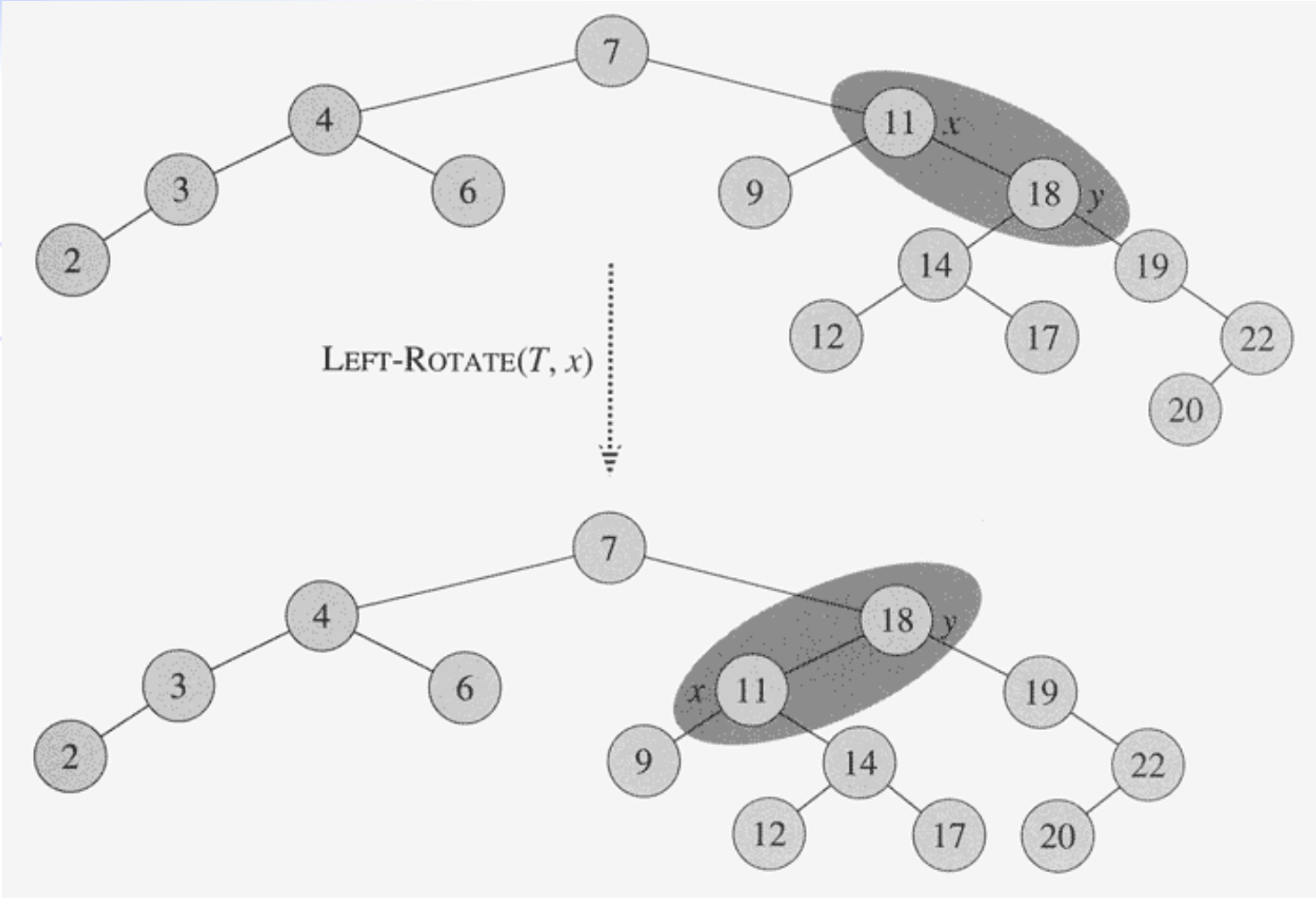

```
color[parent[z]] ← black
color[y] ← black
color[parent[parent[z]]] ← red
z = parent[parent[z]]
```

# Red Black Tree (tt)

- Phép xoay trái (Left-Rotation):



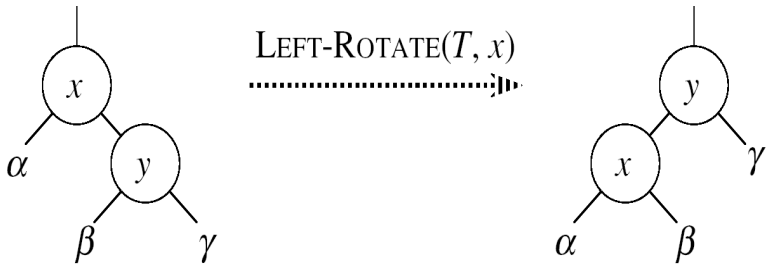
# Red Black Tree (tt)




Ví dụ phép xoay trái

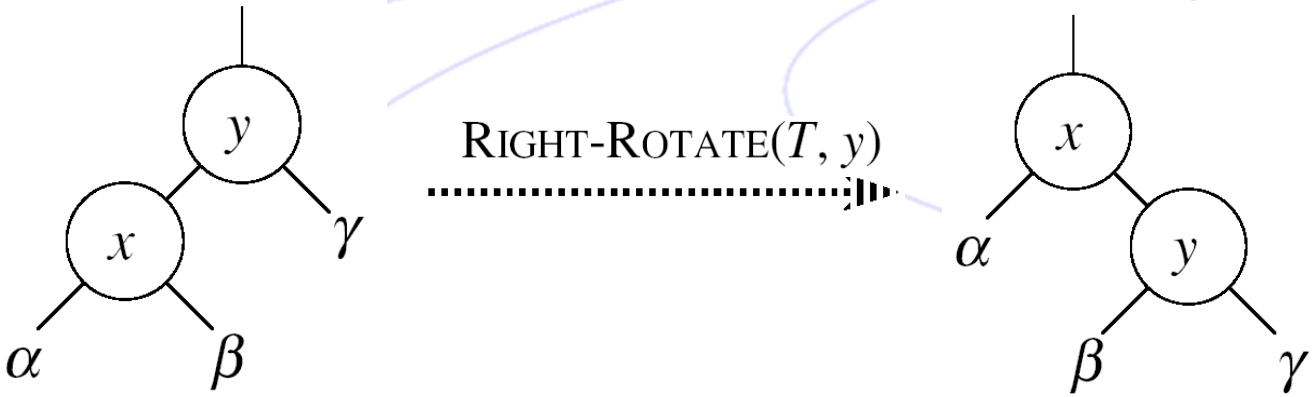
# Red Black Tree (tt)

```
RB_Left_Rotate(T, x)
    y ← right[x];
    right[x] ← left[y];
    if (left[y] ≠ NULL) parent[left[y]] ← x;
    parent[y] ← parent[x];
    if (parent[x] == NULL) root[T] ← y;
    else if (x == left[parent[x]])
                left[parent[x]] ← y;
         else right[parent[x]] ← y;
    left[y] ← x;
    parent[x] ← y;
```



# Red Black Tree (tt)

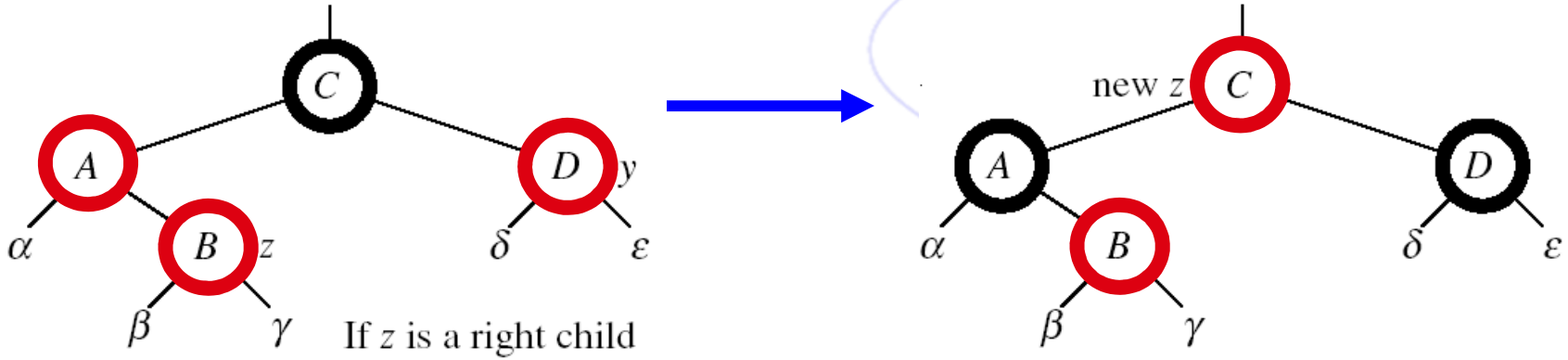
- Phép xoay phải (Right-Rotation):




- **`RB_Right_Rotate(T, x):`** tương tự hàm xoay trái (tự viết)

# Red Black Tree (tt)

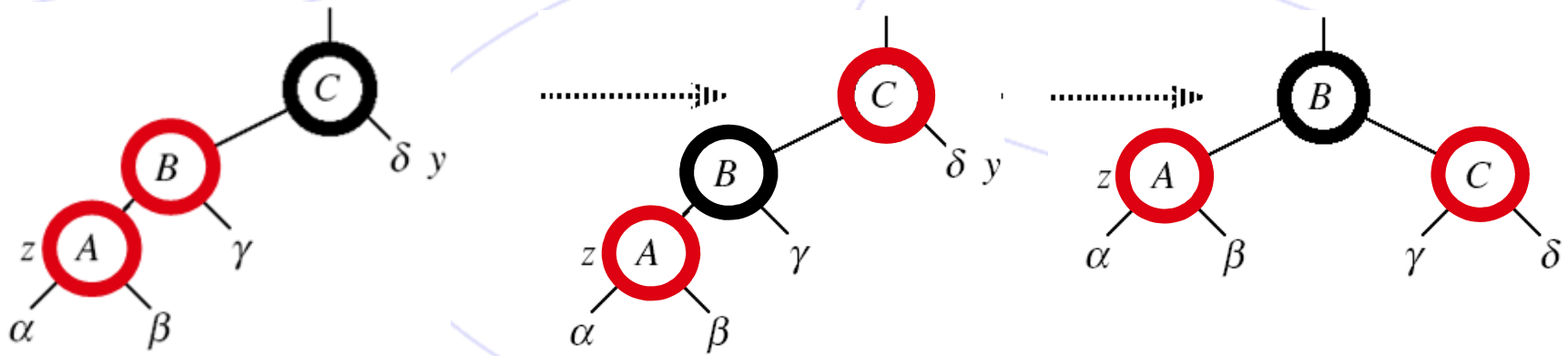
- Tổng kết: có 6 trường hợp xử lý chi tiết
  - Trường hợp 1: áp dụng phép đảo màu




```
color[parent[z]] ← black
color[y] ← black
color[parent[parent[z]]] ← red
z = parent[parent[z]]
```

# Red Black Tree (tt)

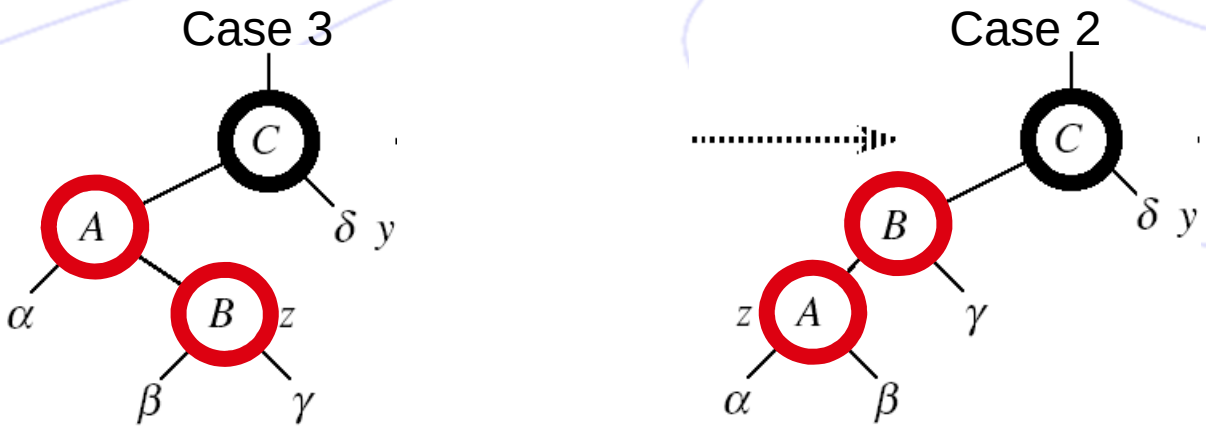
- Tổng kết: (tt)
  - Trường hợp 2: áp dụng phép đảo màu và xoay phải




```
color[parent[z]] ← black
color[parent[parent[z]]] ← red
RIGHT-ROTATE(T, parent[parent[z]])
```

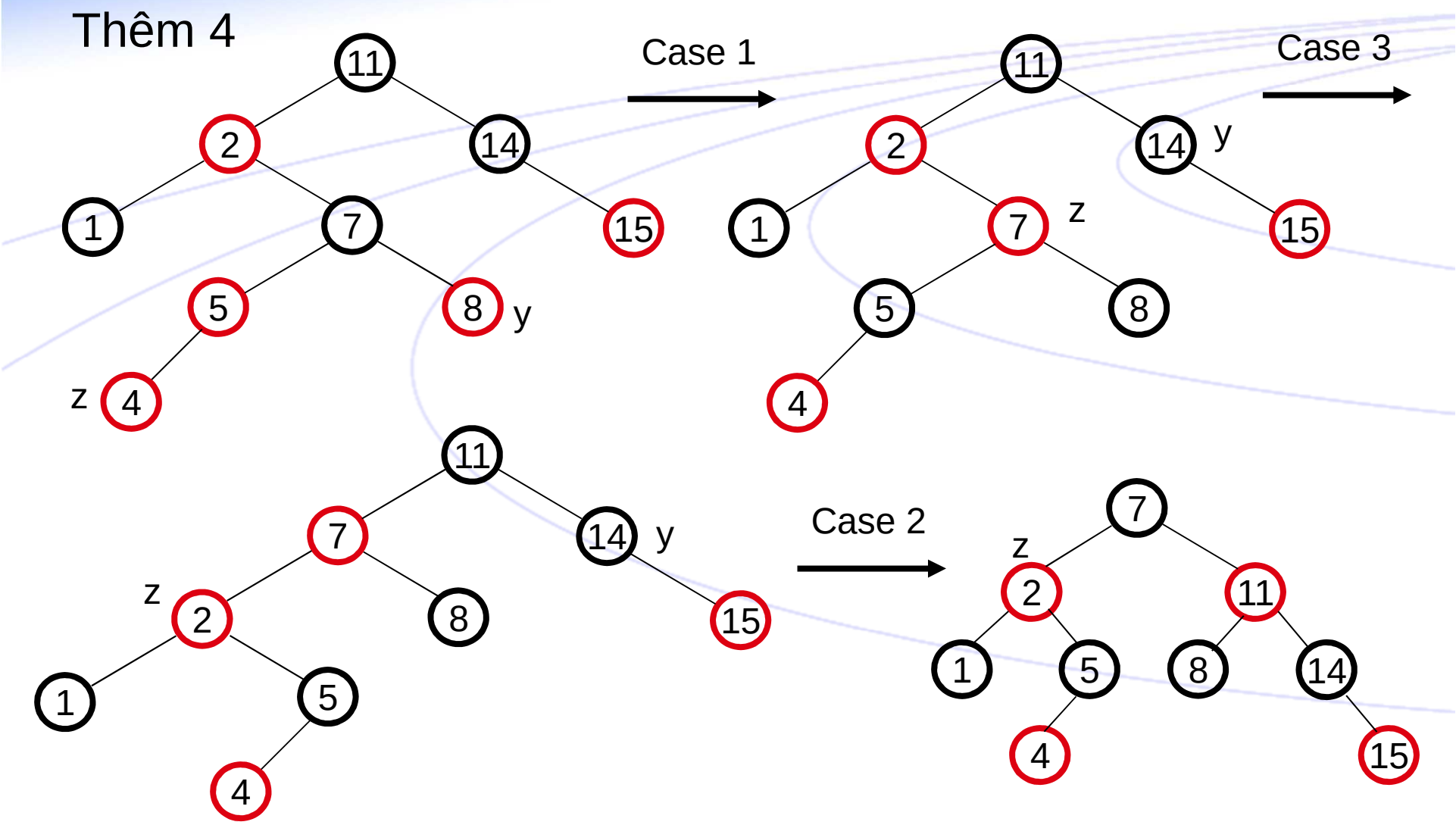
# Red Black Tree (tt)

- Tổng kết: (tt)
  - Trường hợp 3: áp dụng xoay trái để đưa về dạng 2



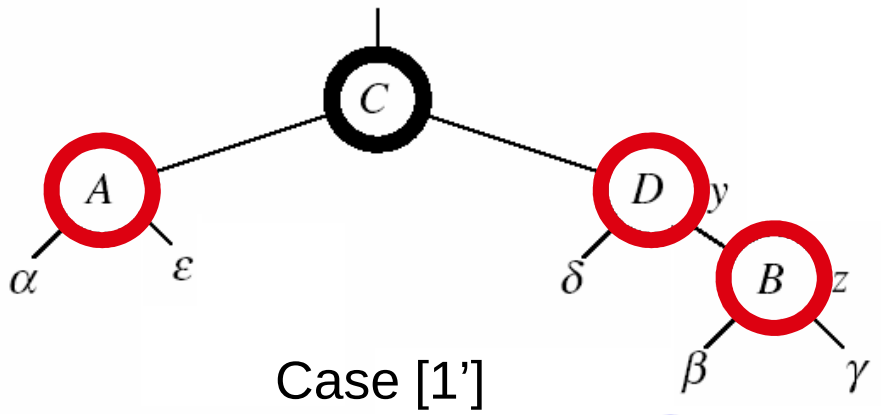

```
z ← parent[z]
LEFT-ROTATE(T, z)
"Xử lý như trường hợp 2"
```

# Red Black Tree (tt)



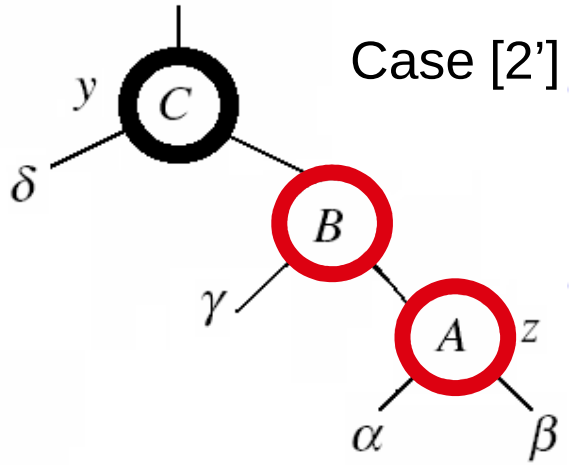
# Red Black Tree (tt)

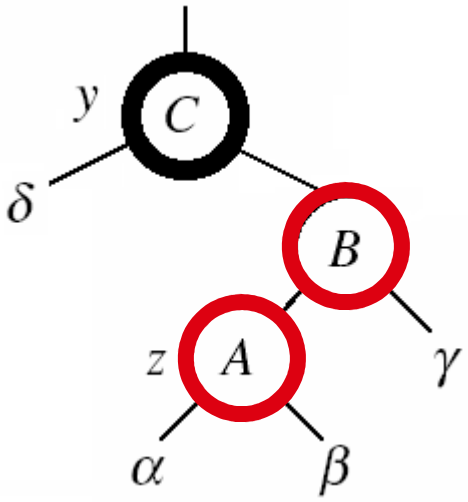
- Tổng kết: (tt)
  - 3 trường hợp tương tự [1'], [2'], [3'] đối xứng với [1], [2], [3] qua trục Y




Case [1']

Case [2']




Case [3']

# Red Black Tree (tt)

```
RB_Insert_FixUp(T, z)
    while (color[parent[z]] == RED) {
       // trường hợp [1], [2], [3]
       if (parent[z] == left[parent[parent[z]]]) {
             y ← right[parent[parent[z]]];
             if (color[y] == RED) “Case 1”;
             else {
                if (z == right[parent[z]]) “Case 3”;
                “Case 2”;
             }
       }
       else …// trường hợp [1’], [2’], [3’]
    color[root[T]] ← BLACK
```

# Red Black Tree (tt)

- Đánh giá thao tác Insert node:
  - Chi phí thêm phần tử mới ($z$): $\boldsymbol{O}(log_2 N)$
  - Chi phí của RB_Insert_FixUp: $\boldsymbol{O}(log_2 N)$
  - Chi phí tổng cộng: $\boldsymbol{O}(log_2 N)$

# Red Black Tree (tt)

- Delete node:
  - Cách thức xóa 1 node: giống như BST
  - *Demo chương trình*
  - Nếu node bị xoá có màu đỏ: không gây ra vi phạm
    - Mọi node phải là đỏ hoặc đen → OK
    - Node gốc là đen → OK
    - Các node lá (NULL) phải luôn luôn đen → OK
    - Nếu một node là đỏ, những node con của nó phải là đen → OK vì không tạo ra 2 node liên tiếp màu đỏ
    - Mọi đường dẫn từ gốc đến nút lá phải có cùng số lượng node đen → OK vì không làm thay đổi số node đen

# Red Black Tree (tt)

- Delete node: (tt)
  - Nếu node bị xoá có màu đen: có thể gây ra vi phạm
    - Mọi node phải là đỏ hoặc đen → OK
    - Node gốc là đen → not OK ! Vì có thể xóa root và thay bằng node đỏ
    - Các node lá (NULL) phải luôn luôn đen → OK
    - Nếu một node là đỏ, những node con của nó phải là đen → not OK ! Vì có thể tạo ra 2 node liên tiếp màu đỏ
    - Mọi đường dẫn từ gốc đến nút lá phải có cùng số lượng node đen → not OK ! Vì làm giảm đổi số node đen
  - Xem chi tiết *"Data structure & Analysis in C", p. 465*

# Red Black Tree (tt)

- Đánh giá:
  - Ưu điểm:
    - Chi phí tìm kiếm $O(log_2 N)$
    - Insert $O(log_2 N)$
    - Delete $O(log_2 N)$
    - Minimum $O(log_2 N)$
    - Maximum $O(log_2 N)$
  - Hạn chế:
    - Phải lưu trữ thuộc tính màu (color) và con trỏ đến nút cha (pParent)
    - Cài đặt phức tạp hơn cây AVL và AA

# Cây cân bằng Red Black và AA

Review

Giới thiệu

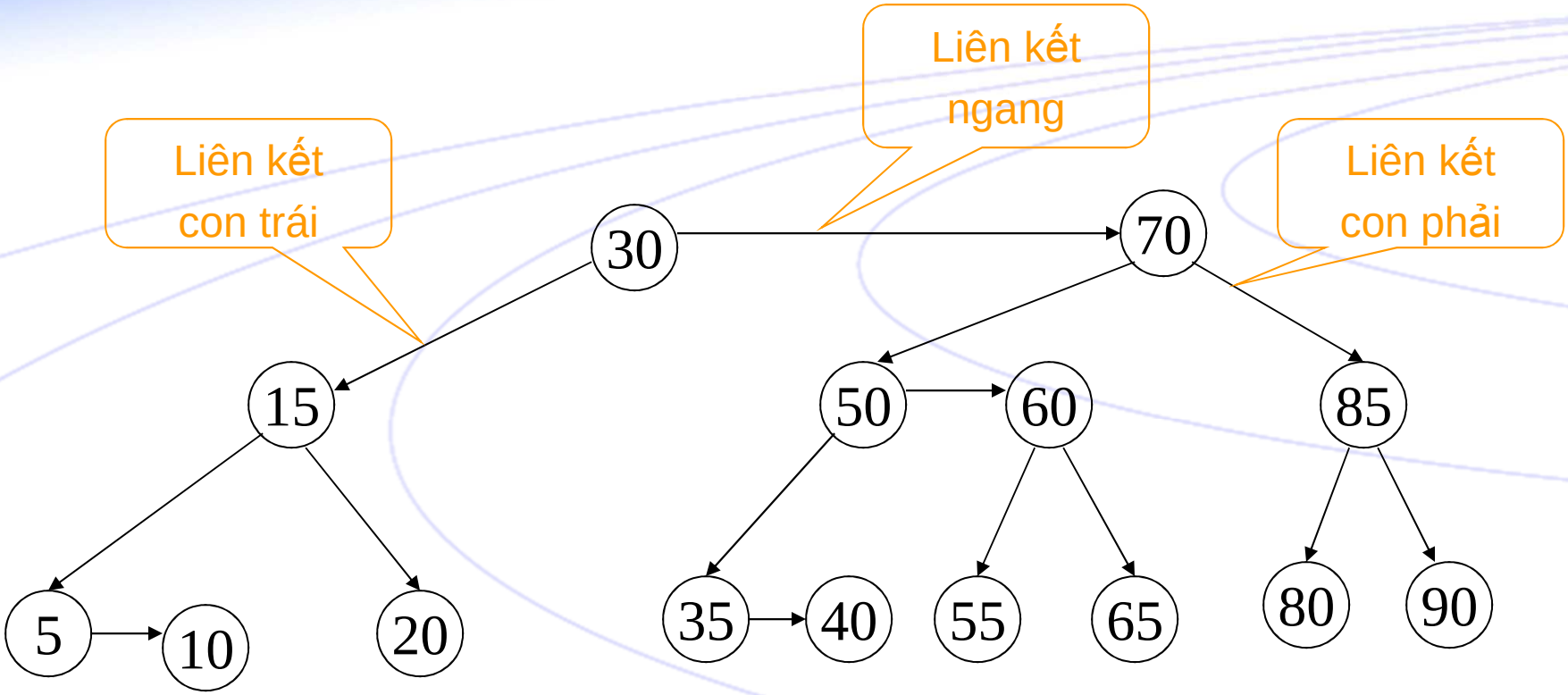
Cây Đỏ – Đen (Red Black Tree)

AA – Tree

# AA (Arne Andersson) – Tree

- Minh họa
- Các khái niệm
- Định nghĩa
- Cấu trúc lưu trữ
- Các thao tác cơ bản
- Đánh giá

# AA – Tree (tt)



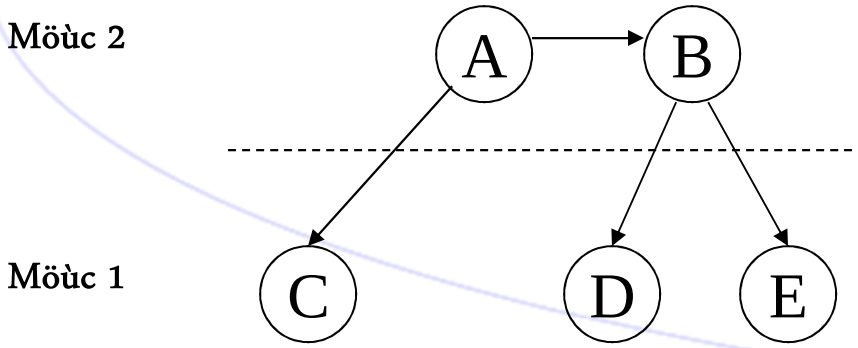

Minh họa cấu trúc cây AA

# AA – Tree (tt)

- Các khái niệm:
  - Mức (Level) của một node
  - Liên kết ngang (Horizontal link)
  - Xoay phải (Right rotation – Skew)
  - Xoay trái (Left rotation – Split)

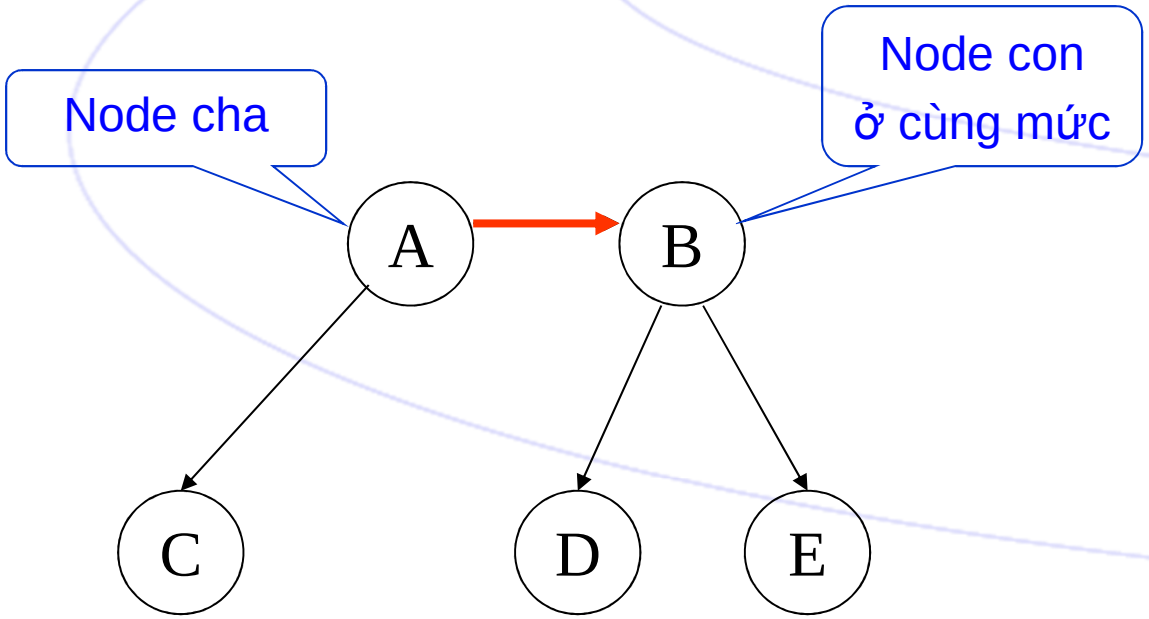
# AA – Tree (tt)

- Các khái niệm: (tt)
  - Mức (Level) của một node: là số liên kết trái từ node đó đến NULL
  - Mức của node NULL là 0
  - Mức của node lá là 1



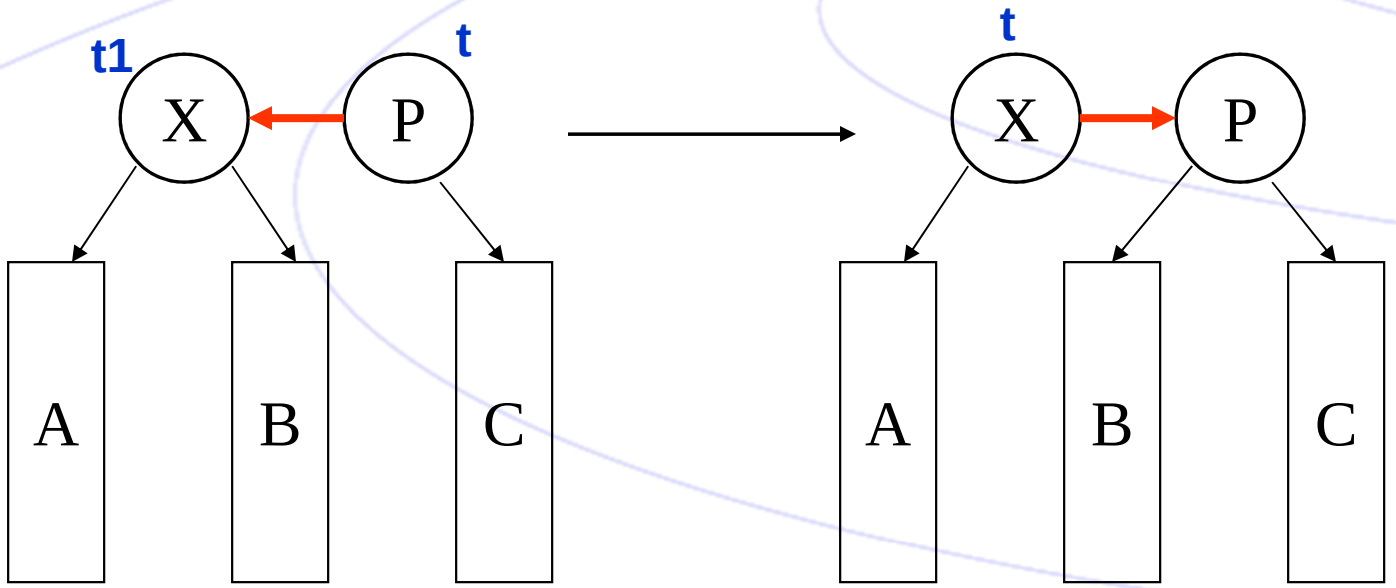
# AA – Tree (tt)

- Các khái niệm: (tt)
  - Liên kết ngang (Horizontal link): là liên kết giữa một node và node con của node đó ở cùng một mức



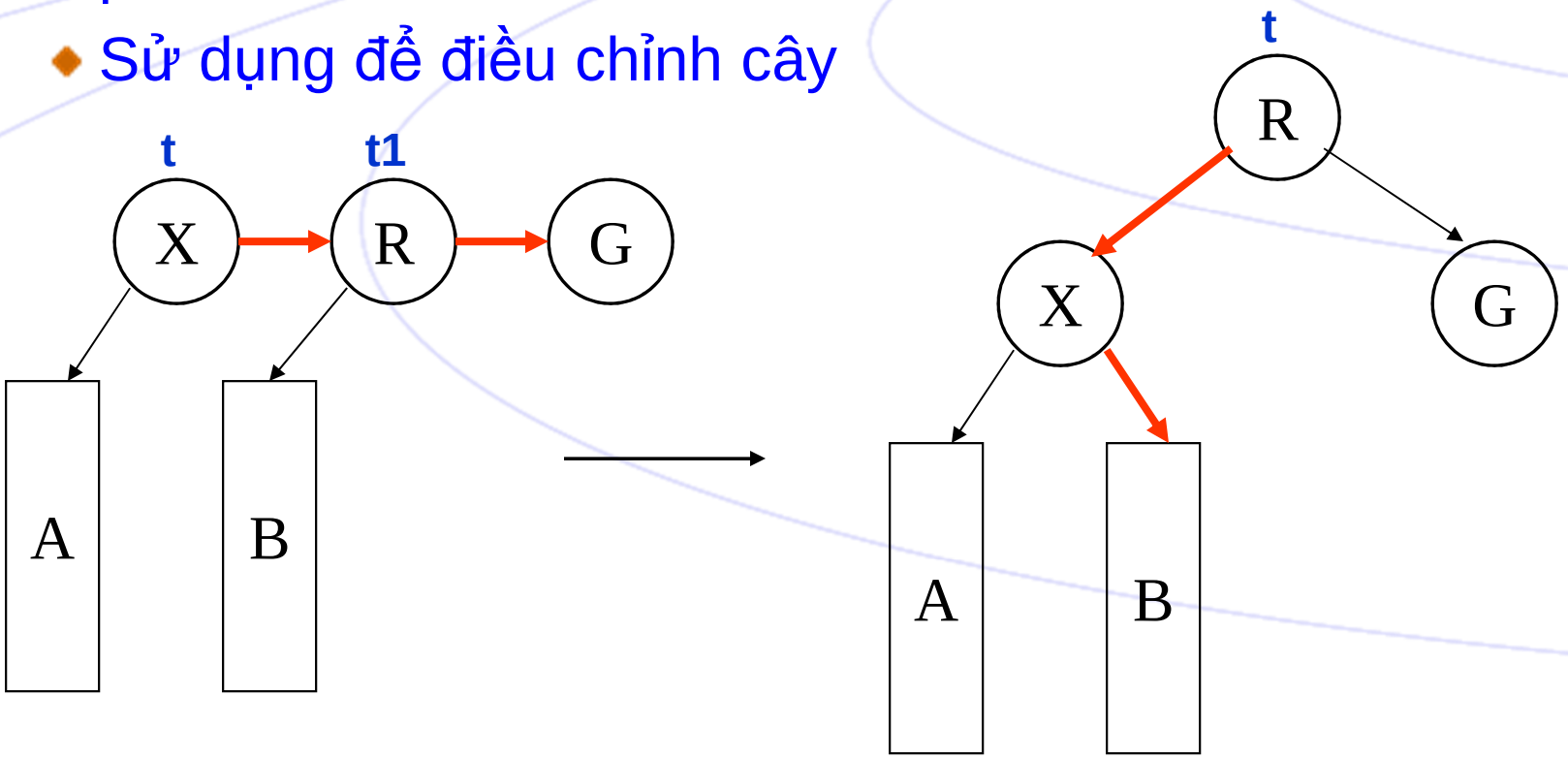
# AA – Tree (tt)

- Các khái niệm: (tt)
  - Xoay phải (Skew): xóa bỏ liên kết ngang bên trái
  - Sử dụng để điều chỉnh cây



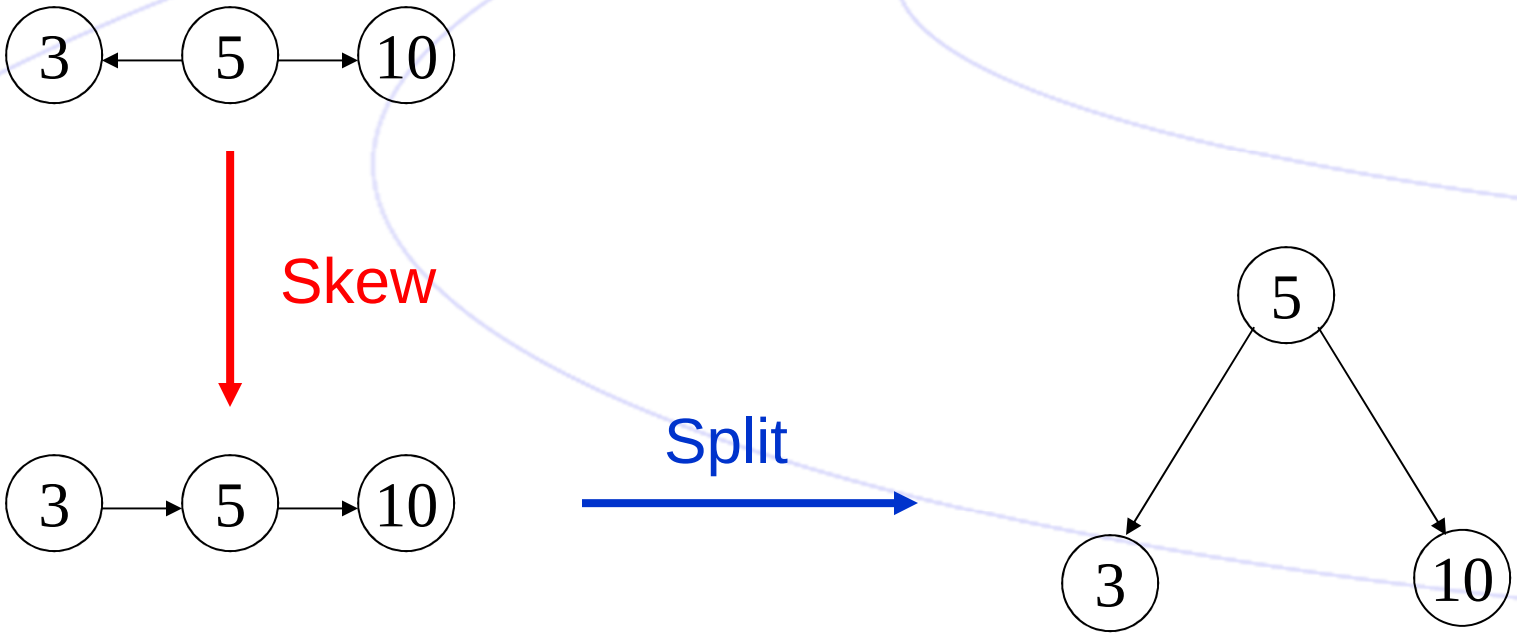
# AA – Tree (tt)

- Các khái niệm: (tt)
  - Xoay trái (Split): xoá bỏ 2 liên kết ngang liên tiếp bên phải
  - Sử dụng để điều chỉnh cây



# AA – Tree (tt)

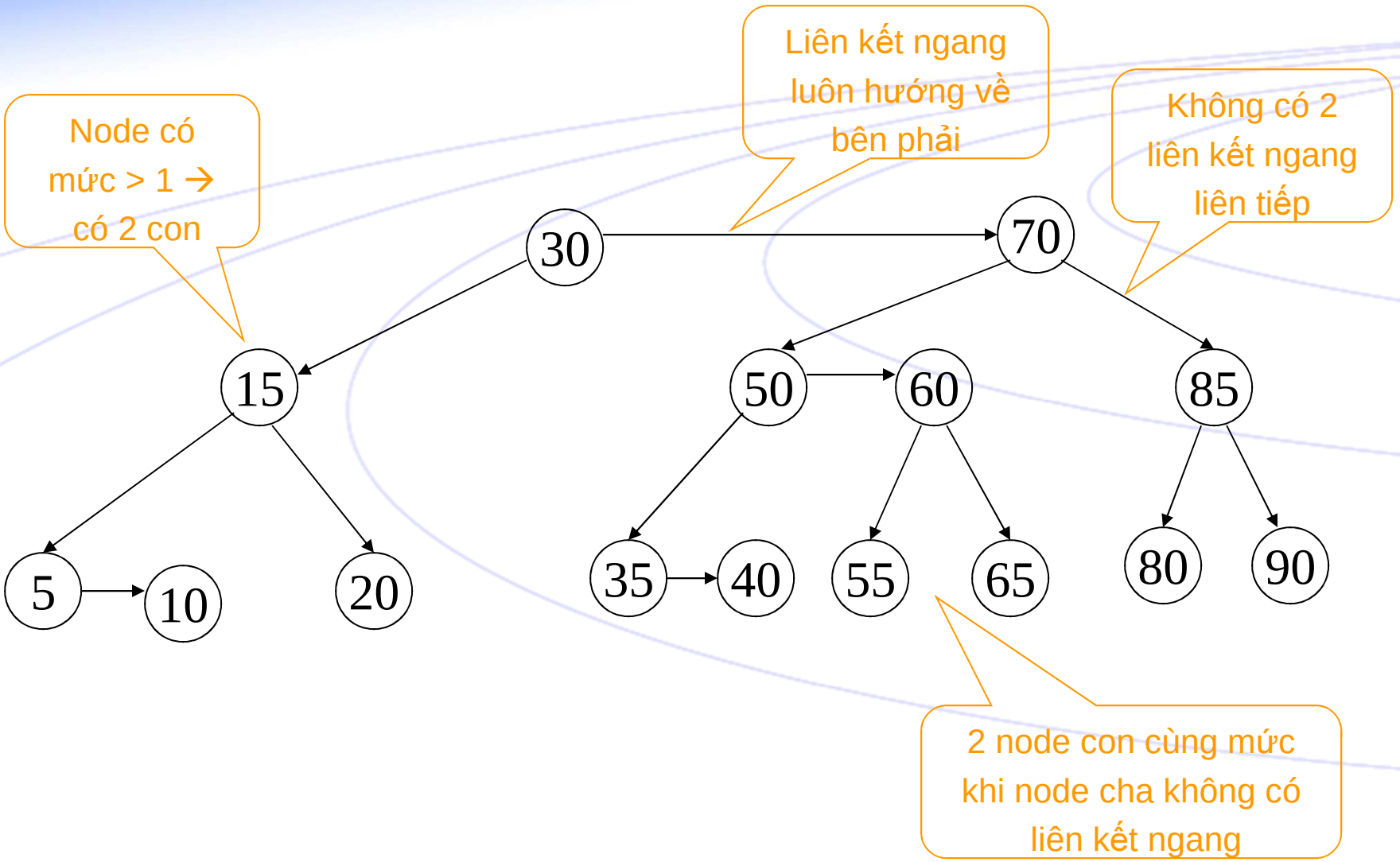
- Skew có thể tạo ra nhiều liên kết ngang phải liên tiếp → sử dụng Split để điều chỉnh



# AA – Tree (tt)

- Định nghĩa: AA tree là một cây nhị phân tìm kiếm (BST) tuân thủ các quy tắc sau:
  - Liên kết ngang luôn hướng về bên phải
  - Không có 2 liên kết ngang liên tiếp nhau
  - Mọi node có mức > 1 sẽ có 2 node con
  - Nếu một node không có liên kết ngang phải thì 2 node con của nó ở cùng mức

# AA – Tree (tt)


Liên kết ngang luôn hướng về bên phải
Node có mức > 1 → có 2 con
Không có 2 liên kết ngang liên tiếp
30
70
15
50
60
85
5
10
20
35
40
55
65
80
90
2 node con cùng mức khi node cha không có liên kết ngang

# AA – Tree (tt)

- Tính chất:
  - Level của node con trái luôn nhỏ hơn level của node cha 1 đơn vị
  - Level của node con phải nhỏ hơn hay bằng level của node cha (nhưng không quá 1 đơn vị)
  - Thêm một node luôn thực hiện ở node có mức = 1

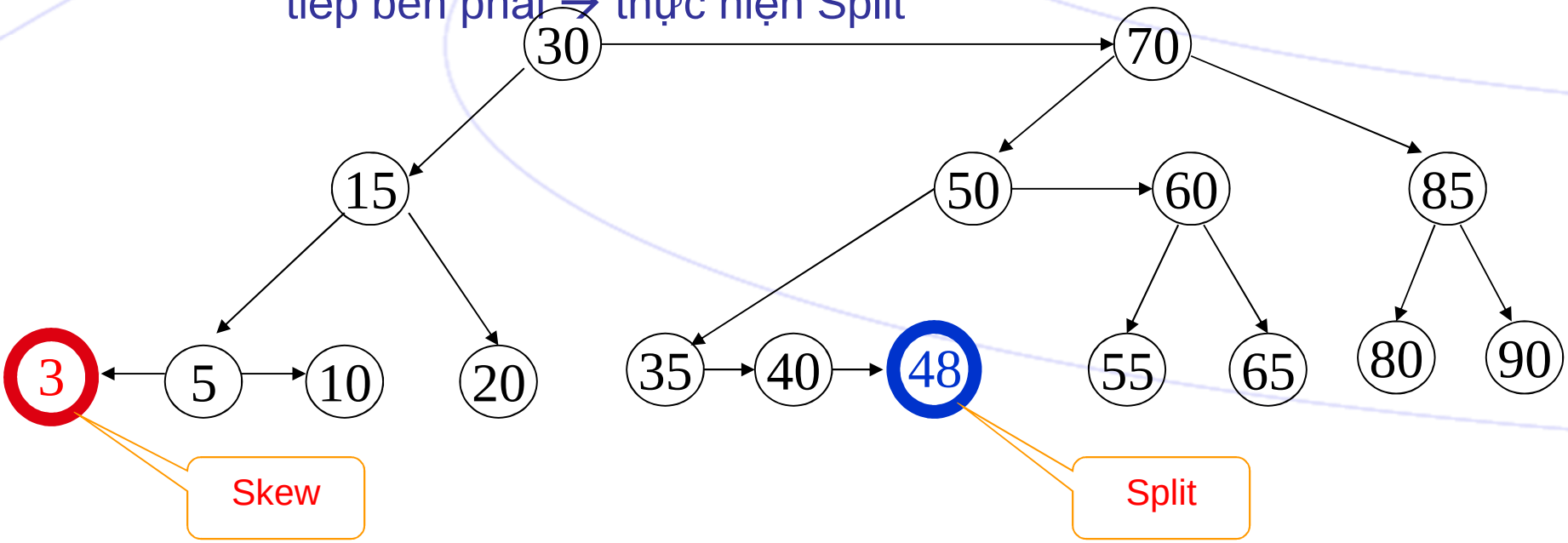
# AA – Tree (tt)

- Cấu trúc lưu trữ:

```
typedef int DataType;   // Kiểu dữ liệu

typedef struct NodeTag {
      DataType           key;        // Dữ liệu
      struct NodeTag   *pLeft;
      struct NodeTag   *pRight;
      int                level;      // mức của node
} AANode;

typedef struct AANode* AATREE;
```

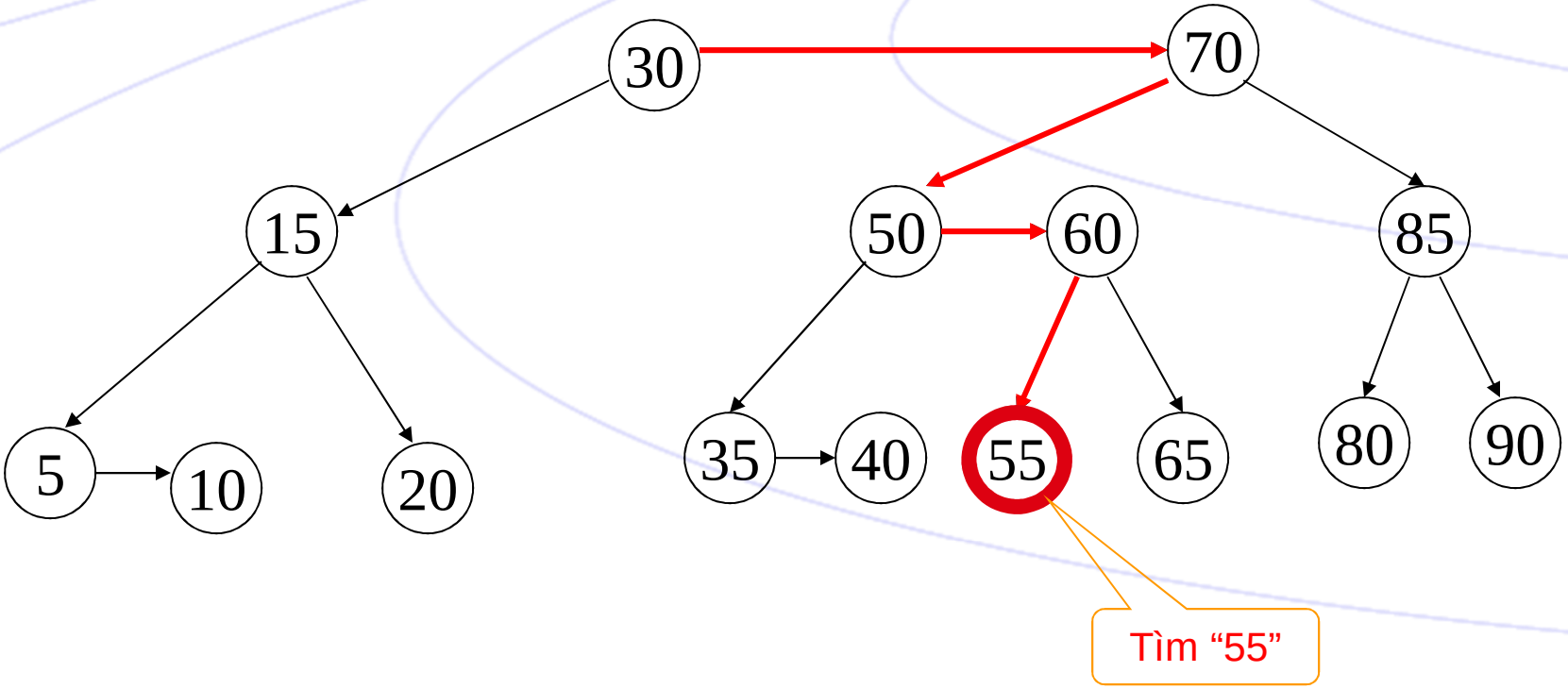
# AA – Tree (tt)

- Các thao tác cơ bản:
  - Khi thêm 1 node
    - Node thêm vào bên trái → tạo ra một liên kết ngang bên trái → thực hiện Skew
    - Node thêm vào bên phải → nếu tạo ra 2 liên kết ngang liên tiếp bên phải → thực hiện Split



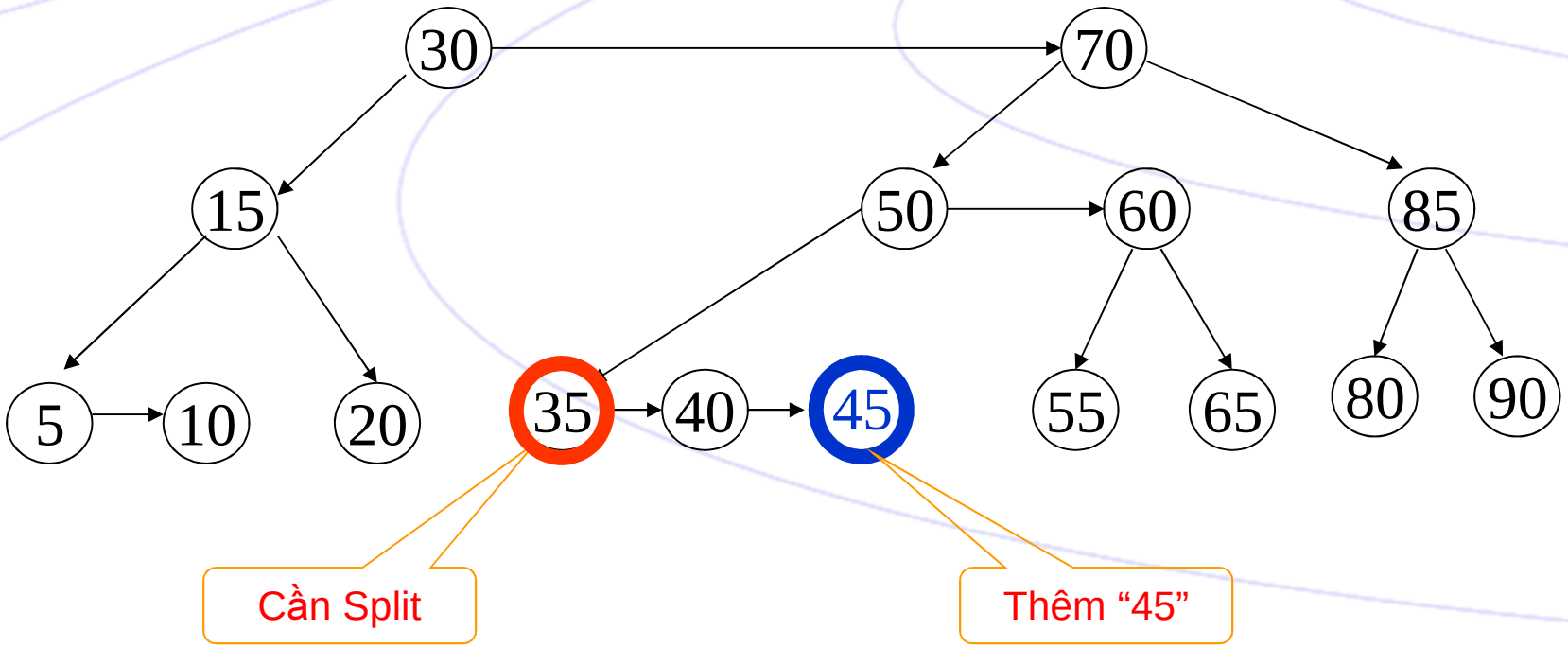
# AA – Tree (tt)

- Các thao tác cơ bản: (tt)
  - Tìm một phần tử: hoàn toàn giống cây BST



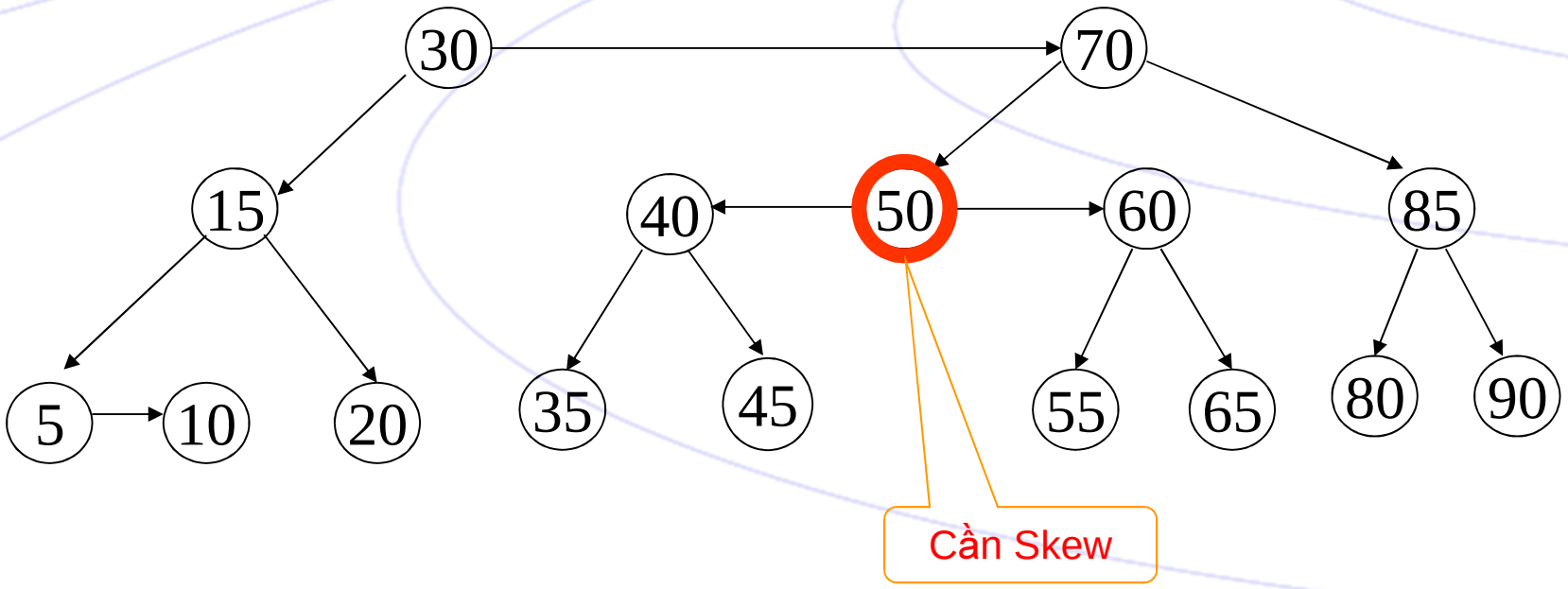
# AA – Tree (tt)

- Các thao tác cơ bản: (tt)
  - Thêm một node:



# AA – Tree (tt)

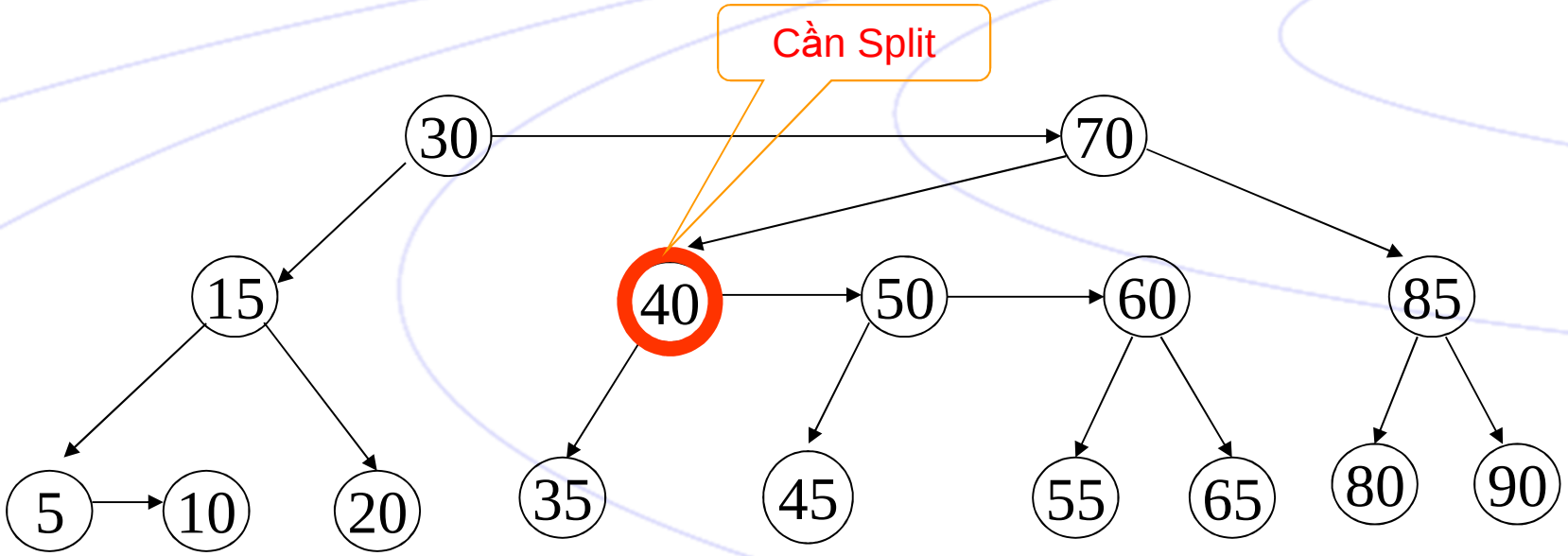
- Các thao tác cơ bản: (tt)
  - Thêm một node:




Sau khi Split tại “35”

# AA – Tree (tt)

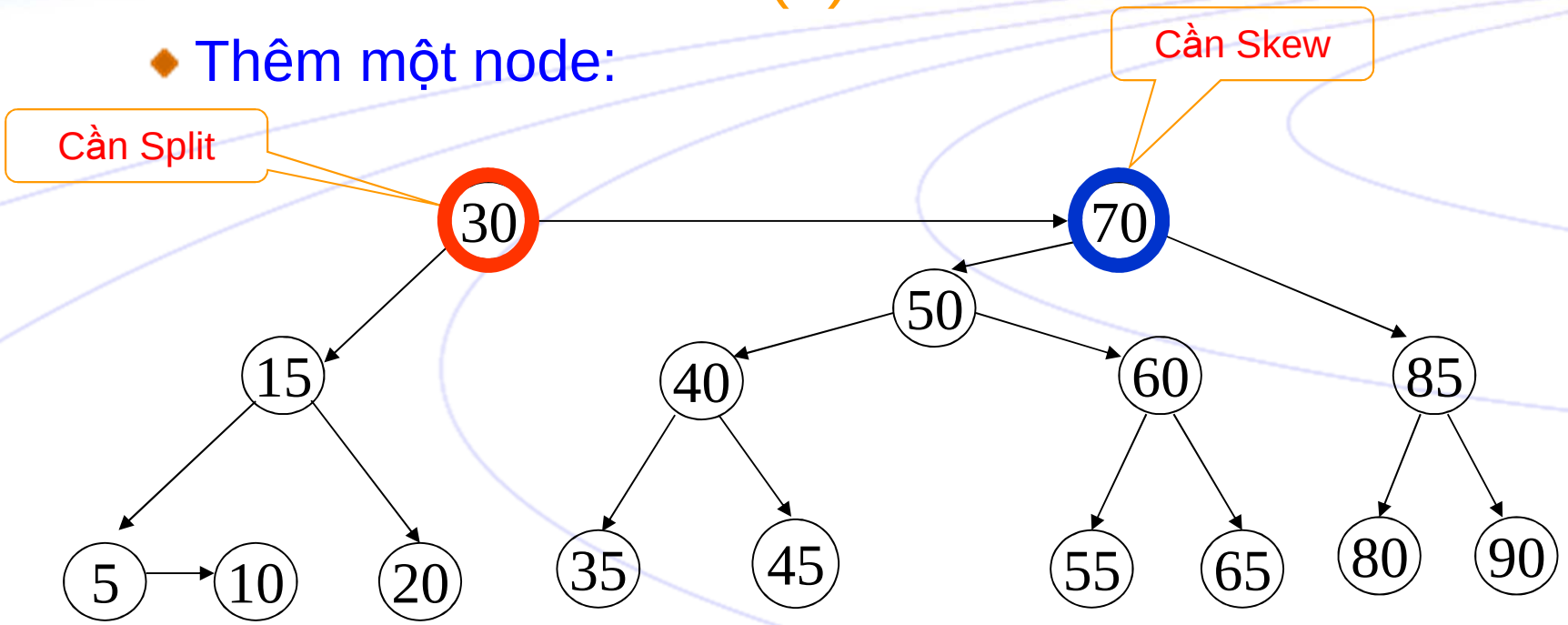
- Các thao tác cơ bản: (tt)
  - Thêm một node:




Sau khi Skew tại “50”

# AA – Tree (tt)

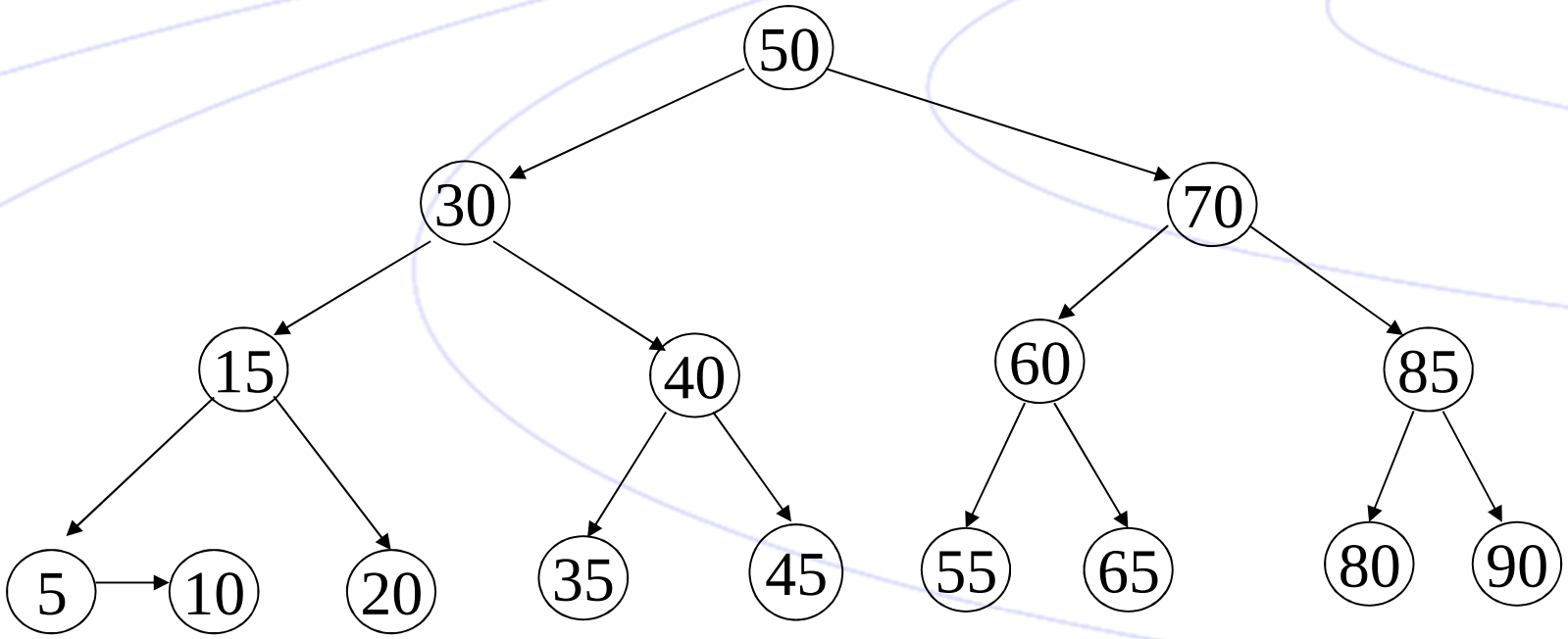
- Các thao tác cơ bản: (tt)
  - Thêm một node:




Sau khi Split tại “40”

# AA – Tree (tt)

- Các thao tác cơ bản: (tt)
  - Thêm một node:




Sau khi Skew tại “70”, và Split tại “30”

→ STOP !

# AA – Tree (tt)

```
AATREE AA_Insert_Node(DataType x, AATREE t)
{
   if(t == NULL)     {      // tạo node mới và thêm vào cây
        t = new AANode;
        t->key = x;
        t->pLeft = t->pRight = NULL;
        t->level = 1;
   }
   else if(x < t->key)
              t->pLeft = AA_Insert_Node(x, t->pLeft);

         else if(x > t->key)
                  t->pRight = AA_Insert_Node(x, t->pRight);

              else return t;  // trùng khóa
   t = Skew(t);
   t = Split(t);
   return t;
}
```

# AA – Tree (tt)

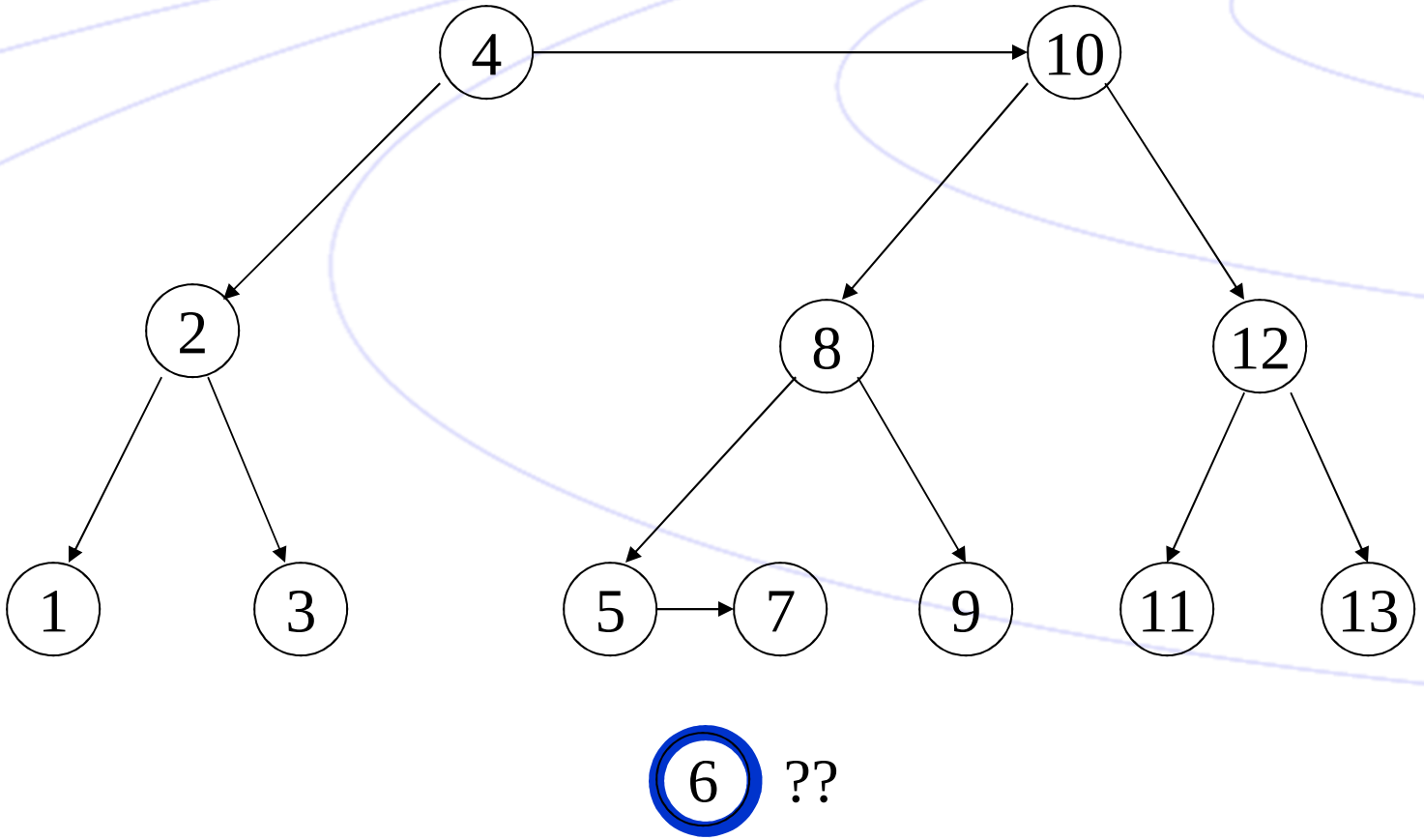
```
AATREE right_rotate(AATREE &t)
{
    AATREE t1;
    t1 = t->pLeft;
    t->pLeft = t1->pRight;
    t1->pRight = t;

    return t1;
}

AATREE Skew(AATREE &t)
{
    if (t->pLeft != NULL)
        if (t->pLeft->level == t->level)
            t = right_rotate(t);
    return t;
}
```

# AA – Tree (tt)

```
AATREE left_rotate(AATREE &t)
{
   AATREE t1;
   t1 = t->pRight;
   t->pRight = t1->pLeft;
   t1->pLeft = t;
   t1->level++;
   return t1;
}
AATREE Split(AATREE &t)
{
    if (t->pRight !=NULL)
       if (t->pRight->pRight != NULL)
        if (t->pRight->pRight->level == t->level)
            t = left_rotate(t);
    return t;
}
```

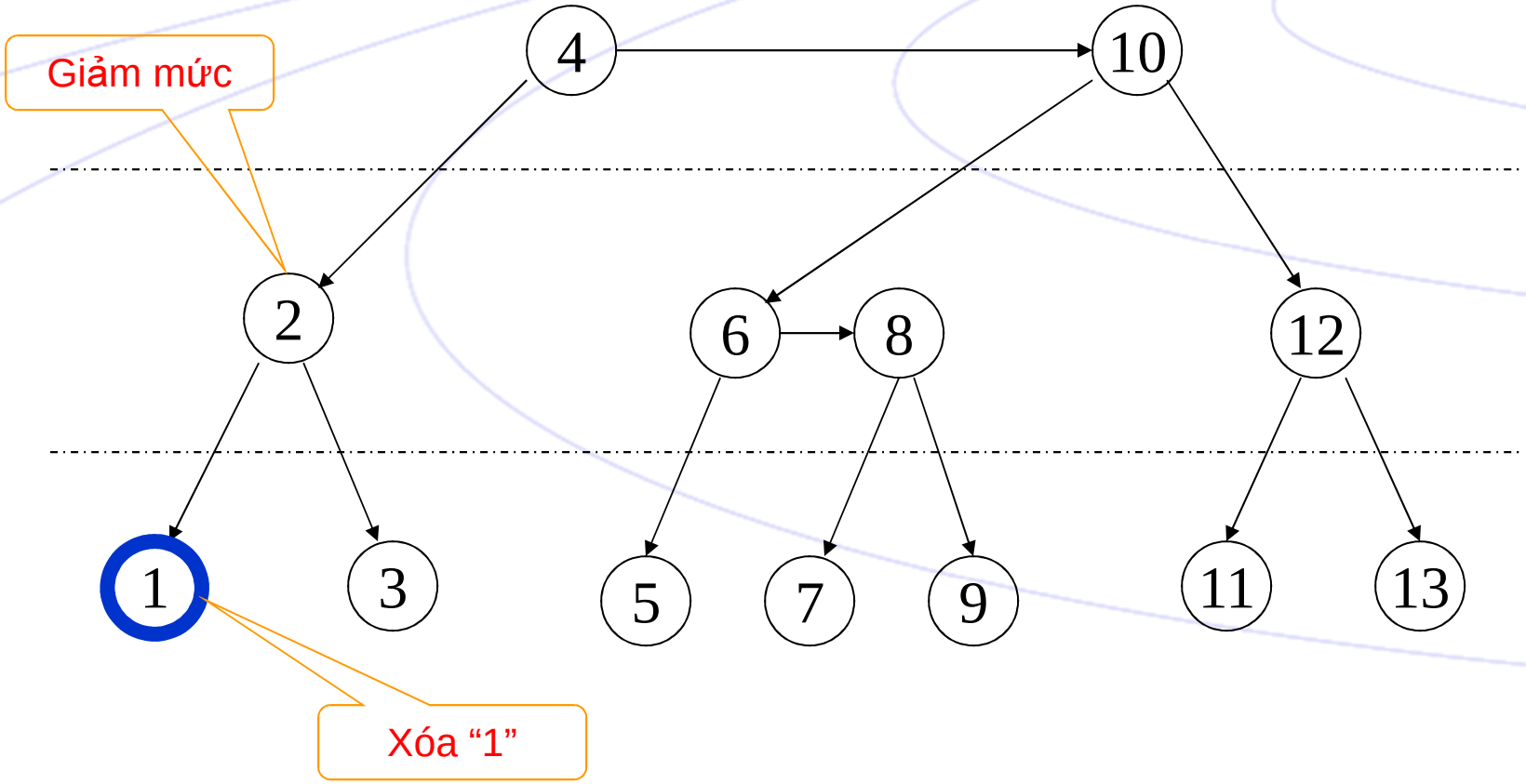
# AA – Tree (tt)

- Các thao tác cơ bản: (tt)
  - Thêm một node: VD. thêm “6”



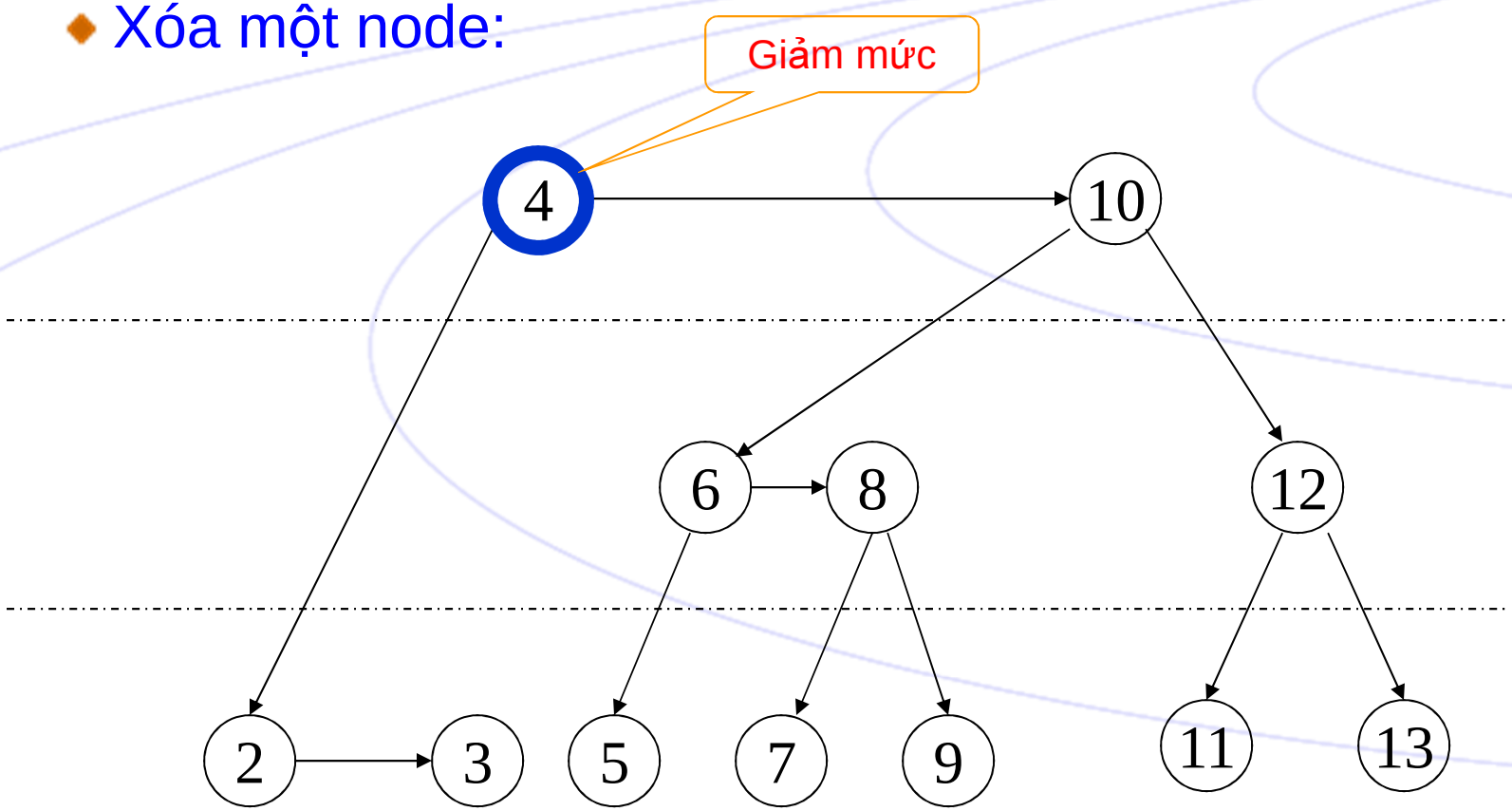
# AA – Tree (tt)

- Các thao tác cơ bản: (tt)
  - Xóa một node:



# AA – Tree (tt)

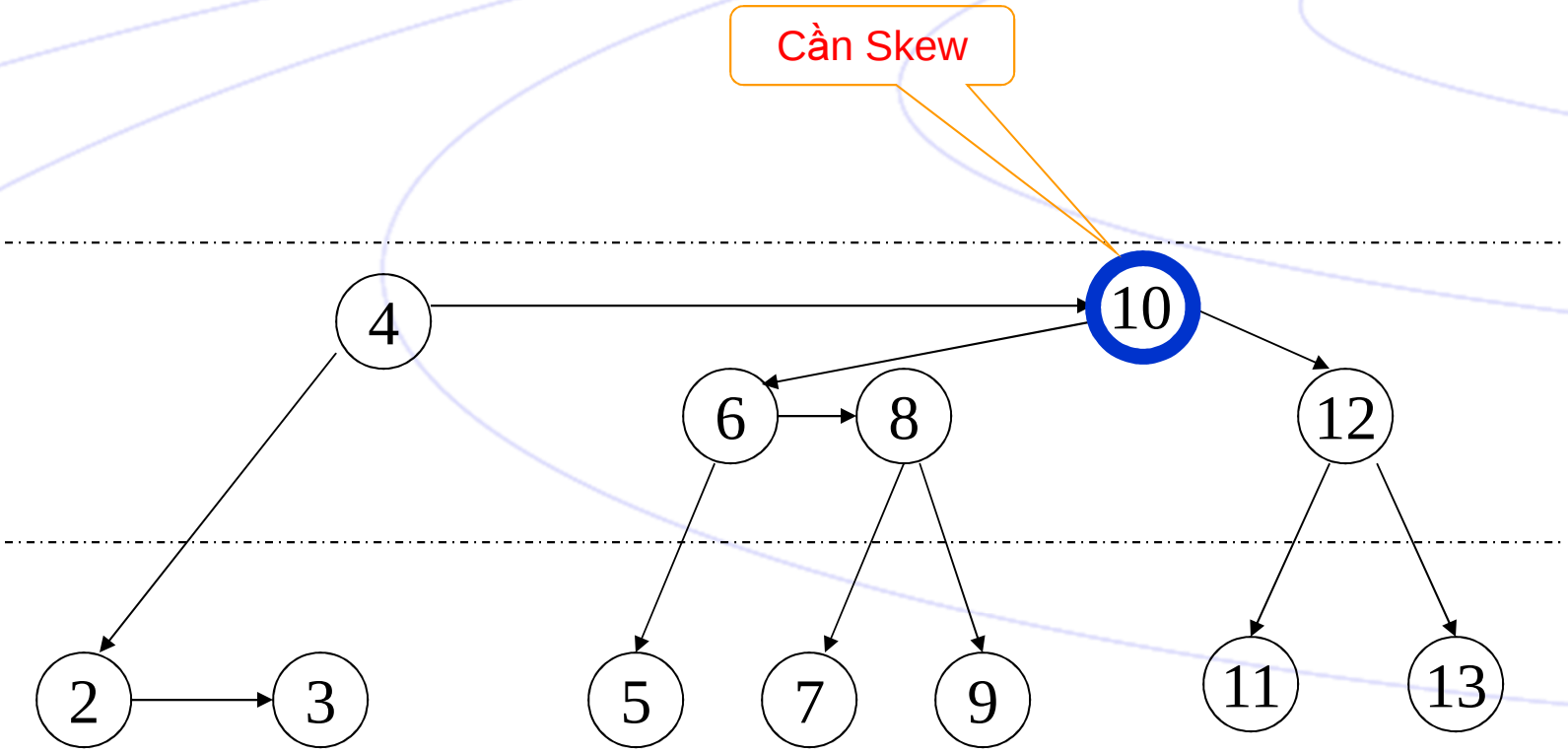
- Các thao tác cơ bản: (tt)
  - Xóa một node:




Sau khi giảm mức tại “2”

# AA – Tree (tt)

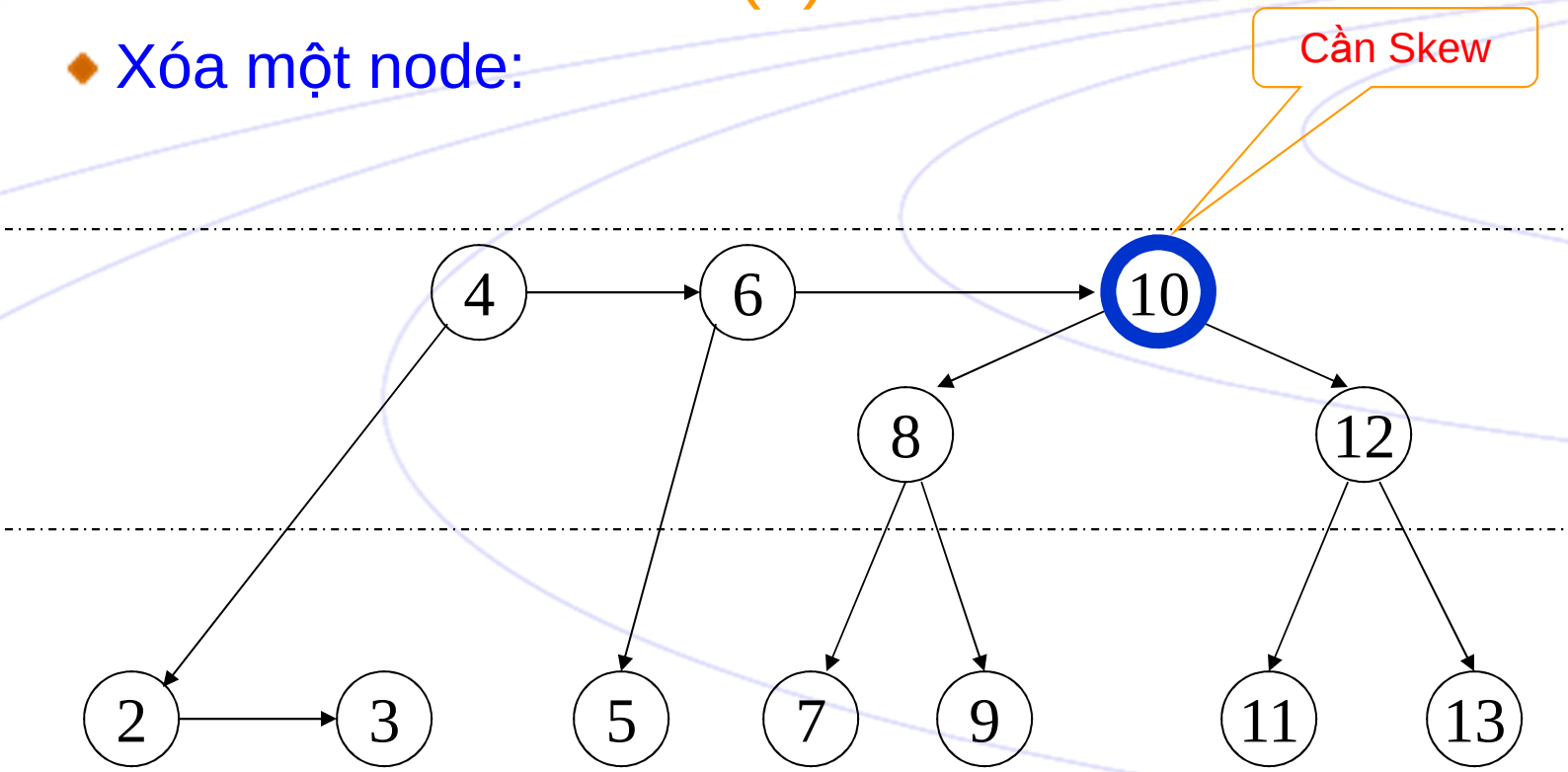
- Các thao tác cơ bản: (tt)
  - Xóa một node:




Sau khi giảm mức tại “4” và “10”

# AA – Tree (tt)

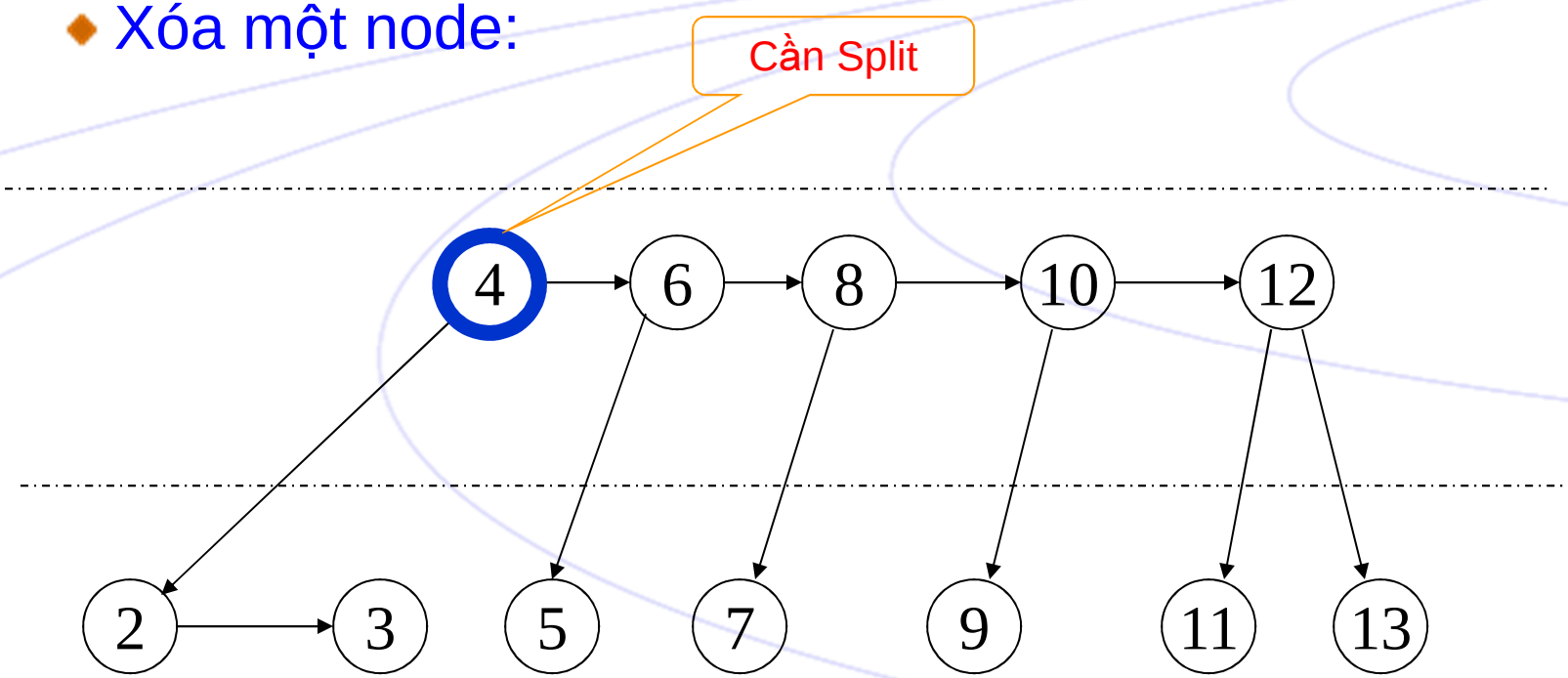
- Các thao tác cơ bản: (tt)
  - Xóa một node:




Sau khi Skew tại “10”, lần 1

# AA – Tree (tt)

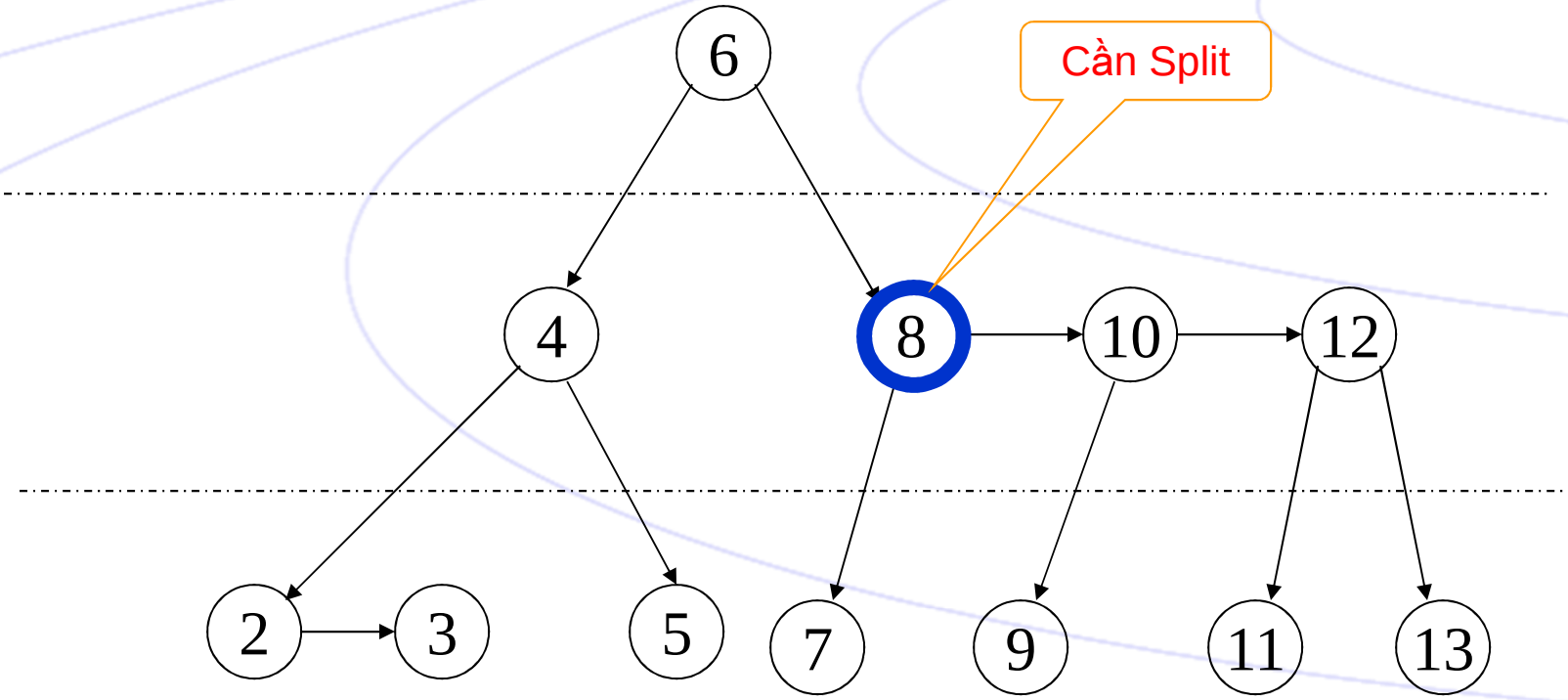
- Các thao tác cơ bản: (tt)
  - Xóa một node:




Sau khi Skew tại “10”, lần 2

# AA – Tree (tt)

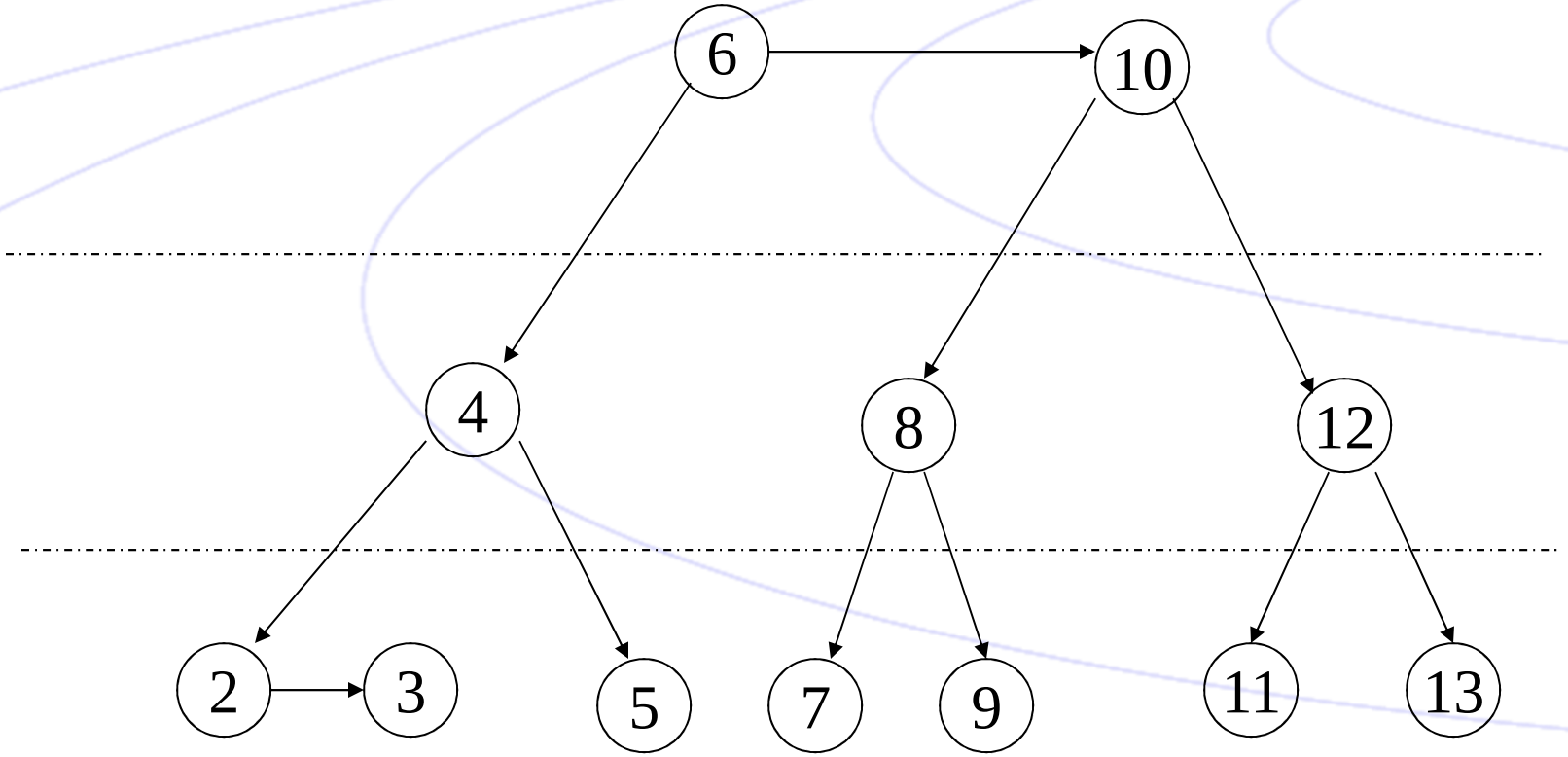
- Các thao tác cơ bản: (tt)
  - Xóa một node:




Sau khi Split tại “4”

# AA – Tree (tt)

- Các thao tác cơ bản: (tt)
  - Xóa một node:




Sau khi Split tại “8” → STOP !

# AA – Tree (tt)

- Đánh giá:
  - Độ phức tạp $O(log_2 N)$
  - Không cần lưu con trỏ đến node cha (pParent)
  - Cài đặt đơn giản hơn cây Red-Black