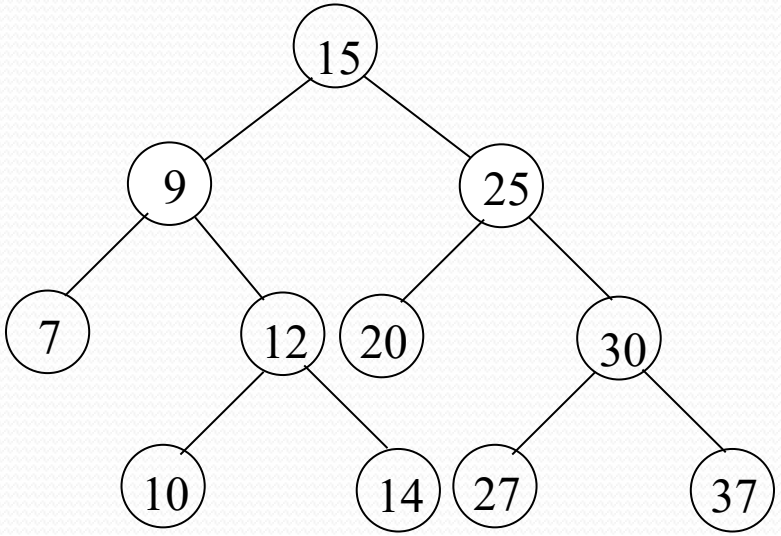
Cây tìm kiếm nhị phân

Nội dung

1. Khái niệm cây tìm kiếm nhị phân
2. Các thao tác
    1. Thêm
    2. Tìm
    3. Xóa
3. Cây cân bằng
4. Bài tập

1. Khái niệm cây tìm kiếm nhị phân

- Cây tìm kiếm nhị phân:
  - Là 1 cây nhị phân
  - ***mọi cây con*** khóa của nút gốc lớn hơn khóa của các nút ở cây con trái và nhỏ hơn khóa của các nút ở cây con phải.



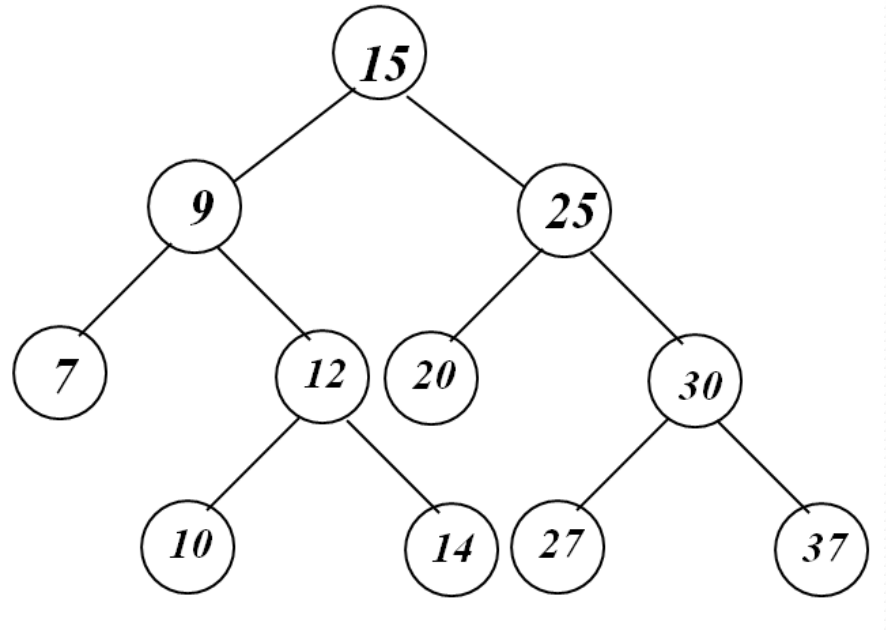
1. Cây tìm kiếm nhị phân

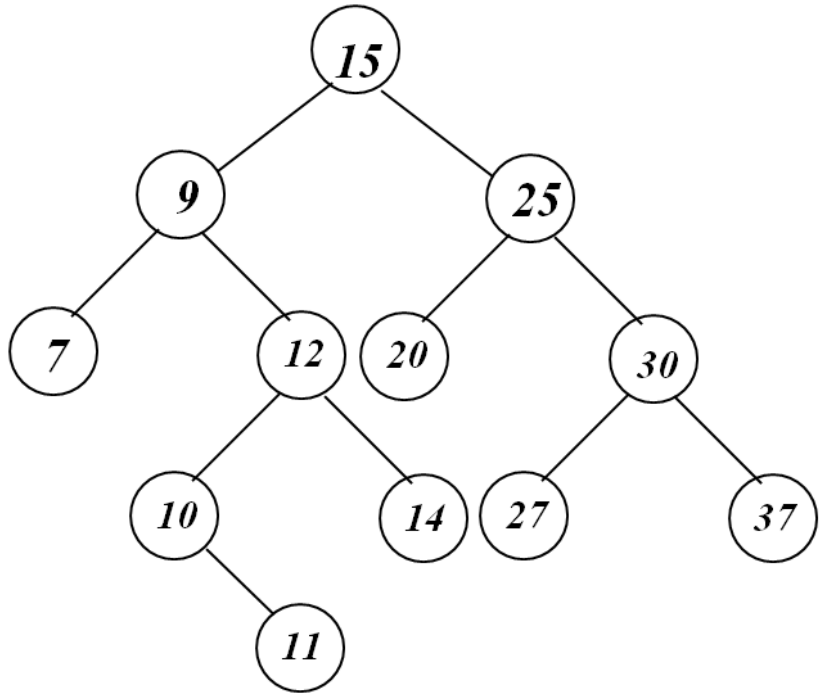
- Ý nghĩa của cây tìm kiếm nhị phân dùng để tìm kiếm nhanh theo khóa.
- Khai báo như khai báo cây nhị phân:

```
struct BinaryTreeNode{
  ElementType data;
  BinaryTreeNode *left, *right;
};
```

2. Thao tác

- Thêm: thêm một phần tử vào cây TKNP
- Ví dụ: thêm phần tử 11 vào cây TKNP





Thuật toán đệ quy

Input: Cây tìm kiếm nhị phân có nút gốc là root, phần tử cần thêm x

Output: Cây tìm kiếm nhị phân sau khi thêm

Action:

+ Nếu cây rỗng thì nút cần thêm là nút gốc.

+ Ngược lại:

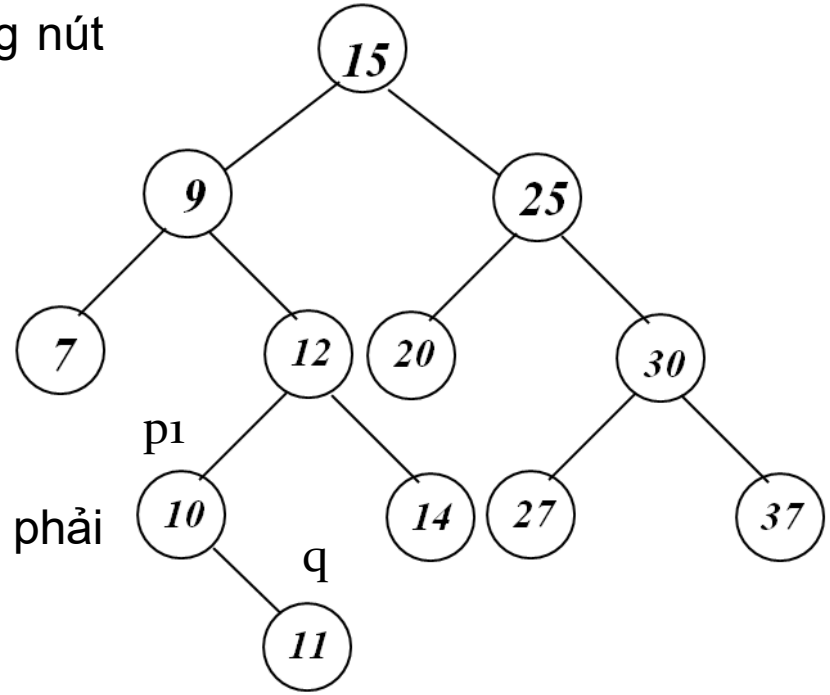
Nếu khóa nút gốc lớn hơn khóa nút cần thêm thì thêm vào cây con trái.

Ngược lại, nếu khóa nút gốc nhỏ hơn khoá của nút cần thêm thì thêm vào cây con phải.

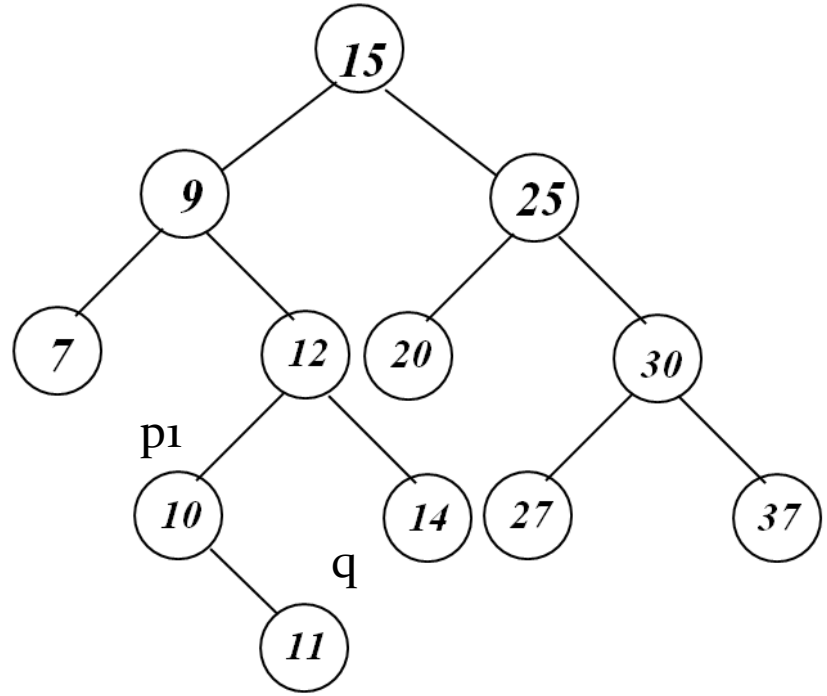
```
void insert(BinaryTreeNode* &root, ElementType x)
 {
   if (root == nullptr) {
      root = new BinaryTreeNode;
      root->data = x;
      root->left = nullptr;
      root->right = nullptr;
 }
 else
 {
    if(root->data.key > x.key)
      insert(root->left, x);
    else
       if(root->data.key < x.key)
          insert(root->right, x);
   }
}
```

Thuật toán lặp

Xuất phát p tại nút gốc của cây, p1 rỗng
Lặp khi p khác rỗng:
+ p1 = p
+ Nếu khóa của dữ liệu tại nút p nhỏ hơn khóa của x thì chuyển p sang nút con trái
+ Ngược lại
Nếu khóa của dữ liệu tại nút p nhỏ hơn khóa của x thì chuyển p sang nút con phải
Ngược lại thì dừng
Cấp phát ô nhớ q kiểu nút của cây nhị phân
Đưa x vào dữ liệu của q
Cho con trái và phải của q là rỗng
Nếu cây rỗng thì gán q cho nút gốc
Ngược lại:
Nếu khóa của dữ liệu tại nút p1 nhỏ hơn khóa của x thì cho p1 liên kết phải đến q
Ngược lại thì cho p1 liên kết trái đến q



```
void insert(BinaryTreeNode* &root, ElementType x)
  {
    BinaryTreeNode *p, *p1, *q;
    p = root; p1 = nullptr;
    while (p != nullptr)
    {
      p1 = p;
      if (p->data.key > x.key)
         p = p->left;
       else
         if (p->data.key < x.key)
         p = p->right;
         else return; //khóa đã có
      }
    q = new BinaryTreeNode;
    q->data = x;
    q->left = nullptr;
    q->right = nullptr;
```




```
    if (root == nullptr)
      root = q;
    else
      if(x.key > p1->data.key)
        p1->right = q;
      else
        p1->left = q;
  }
}
```

2. Thao tác Tìm

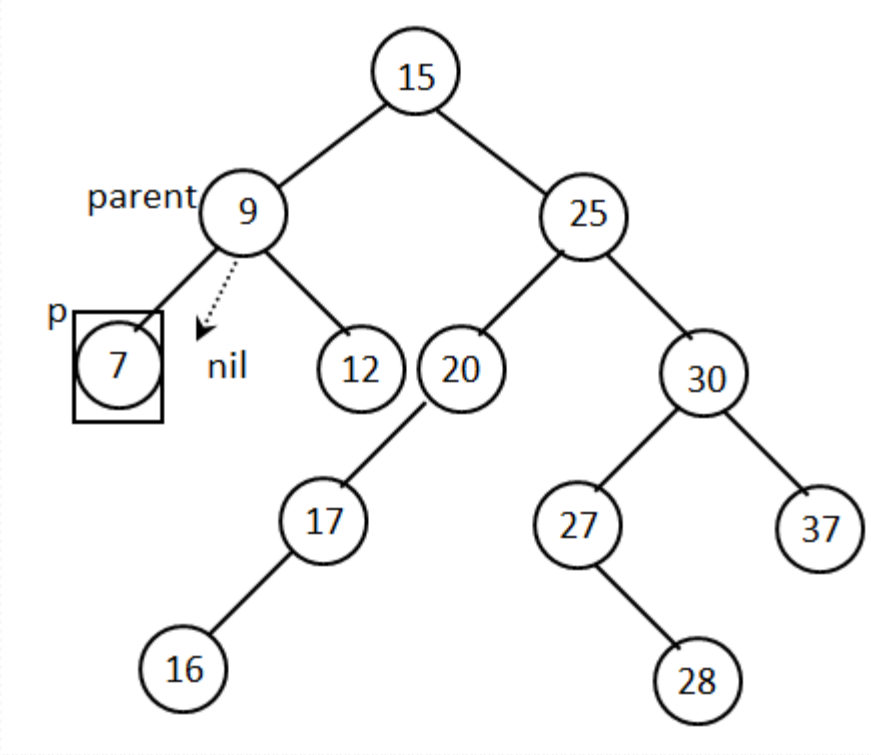
Input: Cây tìm kiếm nhị phân có nút gốc là root, khóa cần tìm là x
Output: True nếu khóa x có trong cây, False trong trường hợp ngược lại
Action:
+ Nếu cây rỗng thì trả về False
+ Ngược lại:
    Nếu dữ liệu tại nút gốc có khóa bằng x thì trả về True
    Ngược lại
        Nếu dữ liệu tại nút gốc có khóa lớn hơn x thì trả về kết quả tìm x ở cây con trái của nút gốc
        Ngược lại thì trả về kết quả tìm x ở cây con phải của nút gốc

```
bool search(BinaryTreeNode* root, KeyType x)
{
    if(root == nullptr) return false;
    else
     if (root->data.key == x) return true;
     else
        if(root->data.key > x)
       return search(root->left, x);
      else return search(root->right, x);
}
```

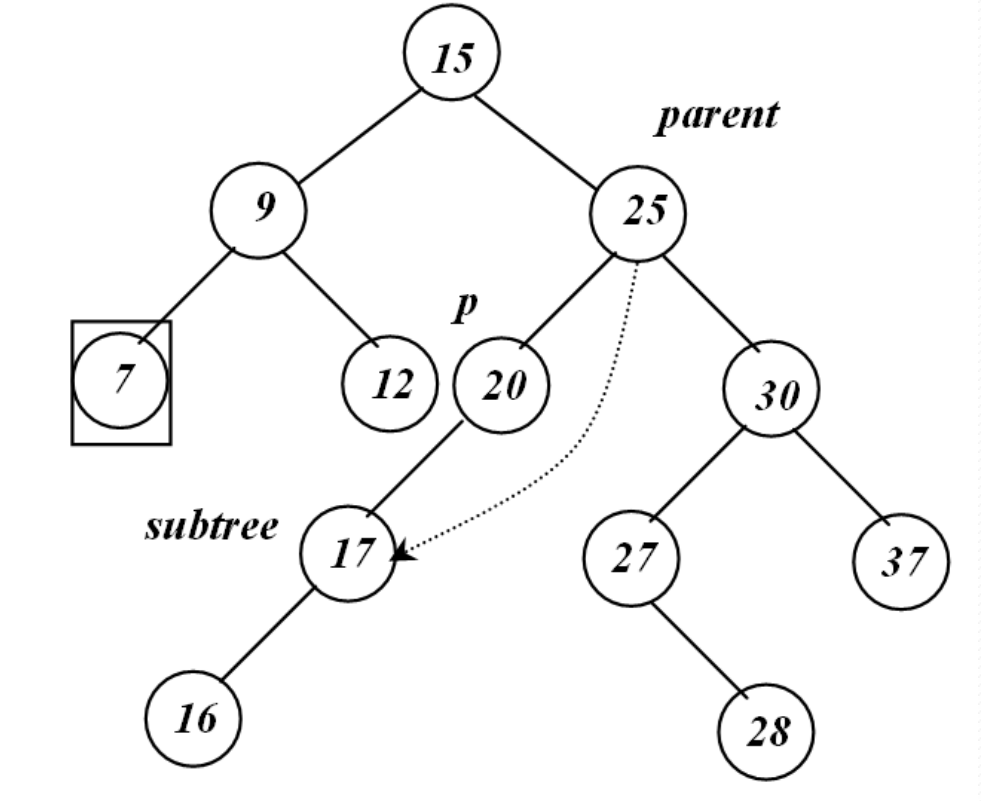
```
bool search(BinaryTreeNode* root, KeyType x)
{
   BinaryTree *p = root;
   while (p != nullptr)
     if(p->data.key == x) return true;
     else
       if(p->data.key > x) p = p->left;
       else p = p->right;
   return false;
}
```

2. Thao tác Xóa

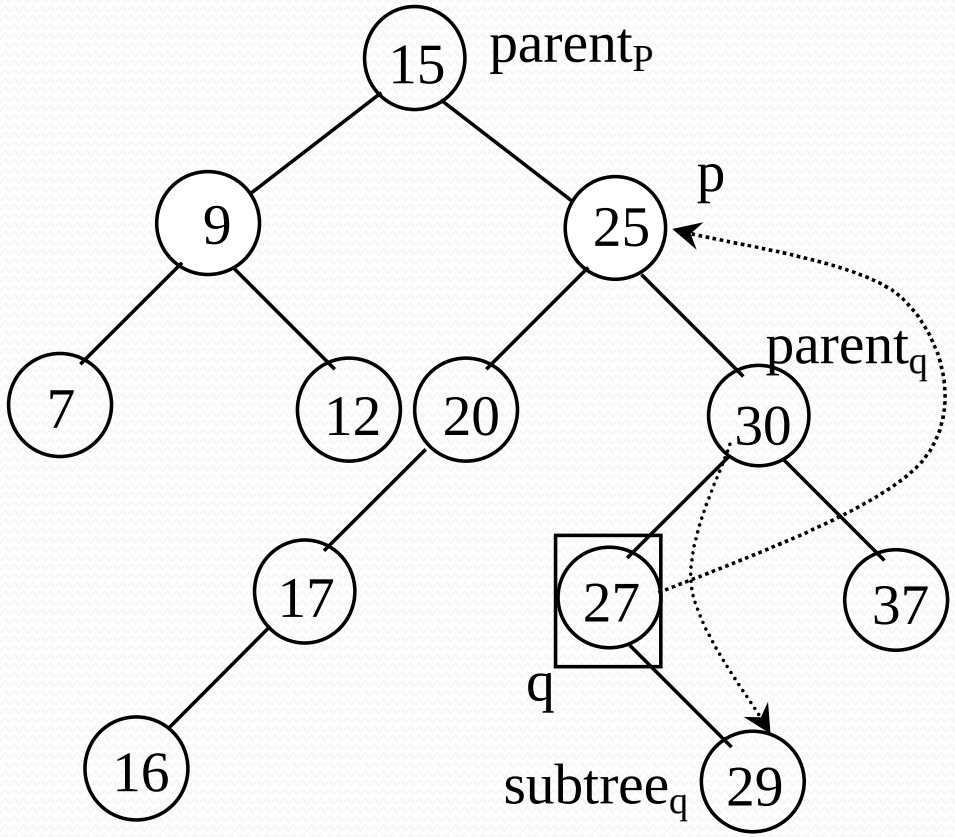
- Xóa nút lá



- Xóa nút có 1 nút con

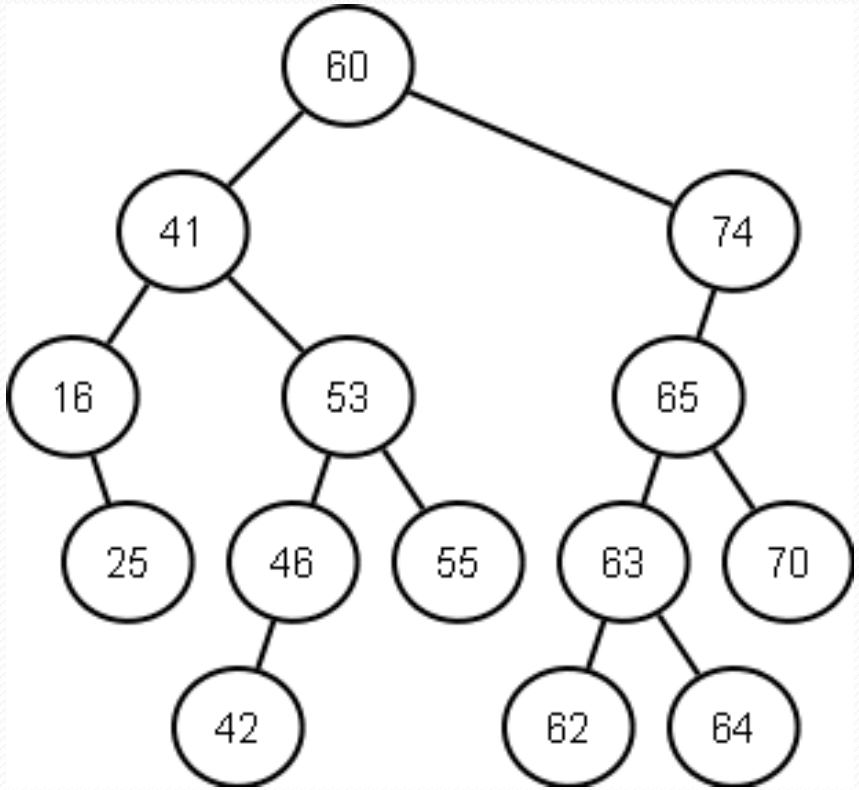


- Xóa nút có 2 nút con



Ví dụ

- Xóa số 60 trong cây



Thuật toán xóa

Input: Cây tìm kiếm nhị phân có nút gốc là root, khóa cần xóa x
Output: Cây tìm kiếm nhị phân sau khi xóa nút có khóa x
Action:
Tìm nút p có khóa x và nút cha của p là parent
Nếu tìm thấy:
+ Nếu p có một cây con là subtree thì
- Cho parent liên kết trái/phải đến subtree
- Thu hồi ô nhớ p
+ Nếu p có hai cây con khác rỗng thì:
- Tìm q là nút có khóa nhỏ nhất ở cây con phải của q với nút cha là $parent_q$
- Đưa dữ liệu từ nút q lên nút p
- Cho $parent_q$ liên kết trái đến nút con phải của q
- Thu hồi ô nhớ q

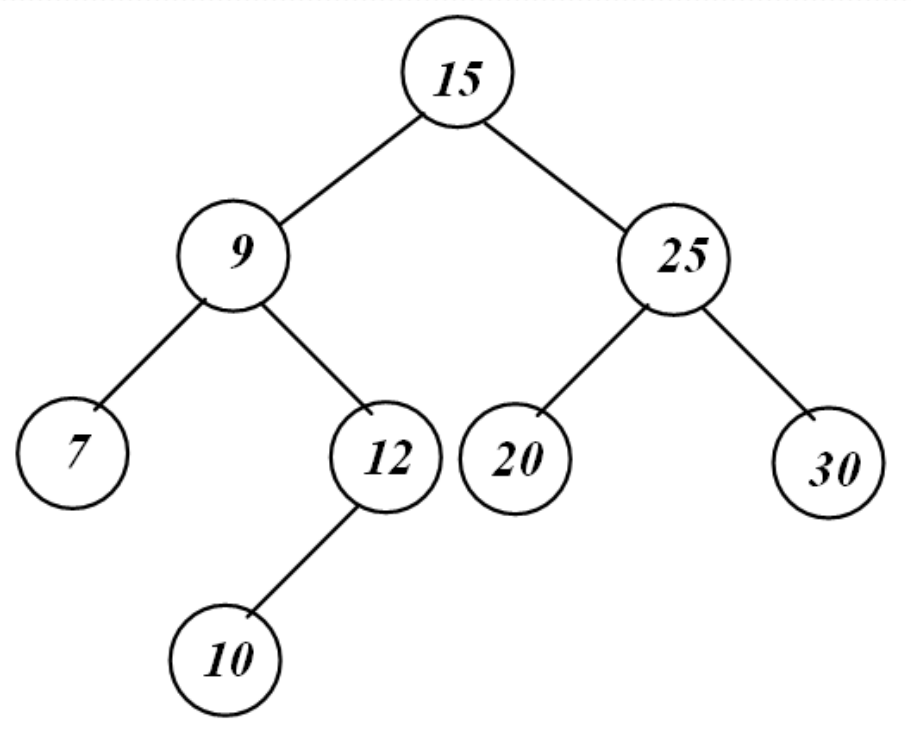
```
void deleteNode(BinaryTreeNode* &root, KeyType x)
{
 BinaryTreeNode *p, *q, *parent, *subtree;
 // Tim nut can xoa p, parent la nut cha cua p
 p = root;
 parent = nullptr;
 while (p != nullptr && p->data.key != x)
 {
  parent = p;
  if (p->data.key < x)
   p = p->right;
  else
   if (p->data.key > x)
    p = p->left;
 }
```

```
if (p !=null) //Co phan tu can xoa
 {
     /* Truong hop nut can xoa co 2 nut con :
      - Tim nut q la nut trai nhat cua cay con ben phai.
      - parent la nut cha cua nut q. */

   if (p->left != nullptr && p->right != nullptr)
   {
     q = p->right;
    parent = p;
    while (q->left != nullptr)
      {
       parent = q;
         q = q->left;
      }

    // Dua du lieu cua nut q vao nut p
      p->data = q->data;
    p = q;
   }
```

```
// Xoa nut p trong truong hop khong qua mot nut con
// Tim cay con cua p
   subtree = p->left;
   if (subtree == nullptr)
    subtree = p->right;
   if (parent == nullptr) // nut can xoa la nut goc
      root = subtree;
   else
   {
   //p la nut trai cua parent
    if (parent->data.key > p->data.key)
       parent->left = subtree;
    else
      parent->right = subtree;
   }
   delete p;
 }
}
```

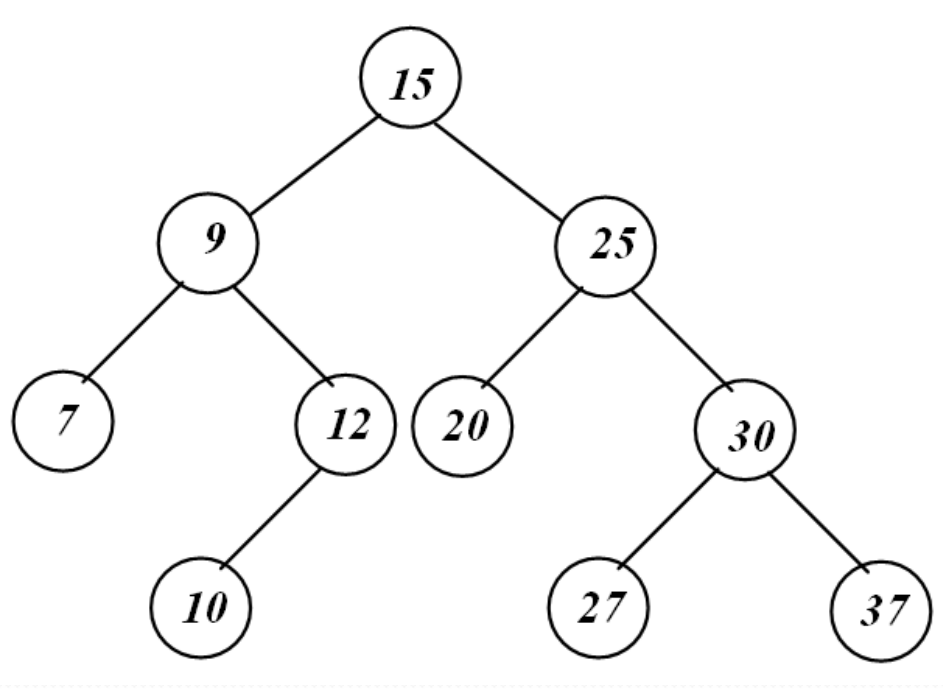
3. Cây cân bằng

- *Cây cân bằng hoàn toàn:*
  - Là cây tìm kiếm nhị phân
  - Mọi cây con, số nút của cây con trái và số nút của cây con phải lệch nhau không quá 1.



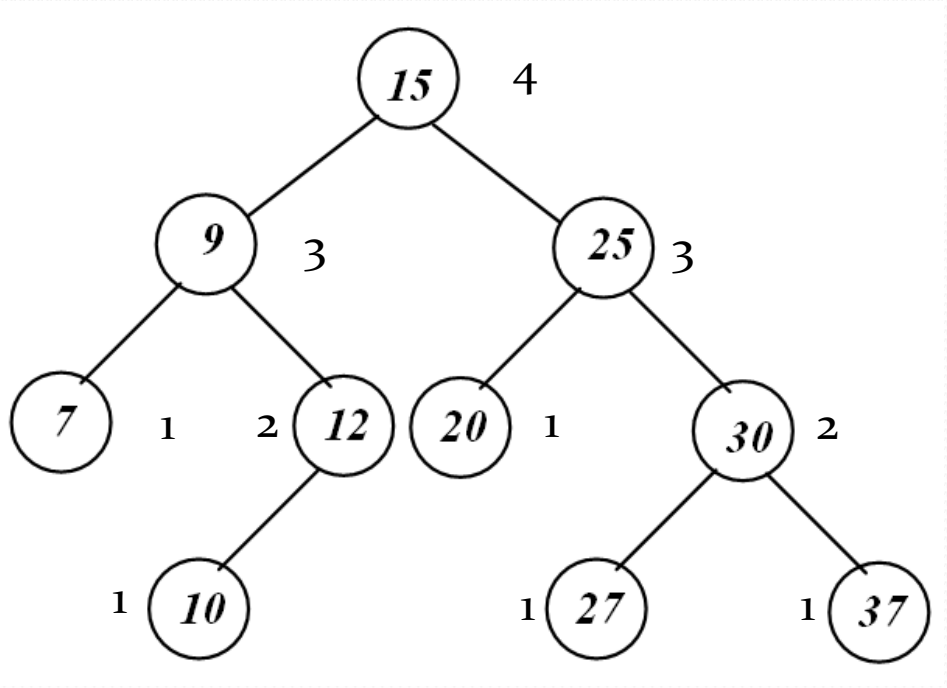
3. Cây cân bằng

- *Cây cân bằng:*
  - Là cây tìm kiếm nhị phân
  - Mọi cây con, chiều cao của cây con trái và chiều cao của cây con phải lệch nhau không quá 1.



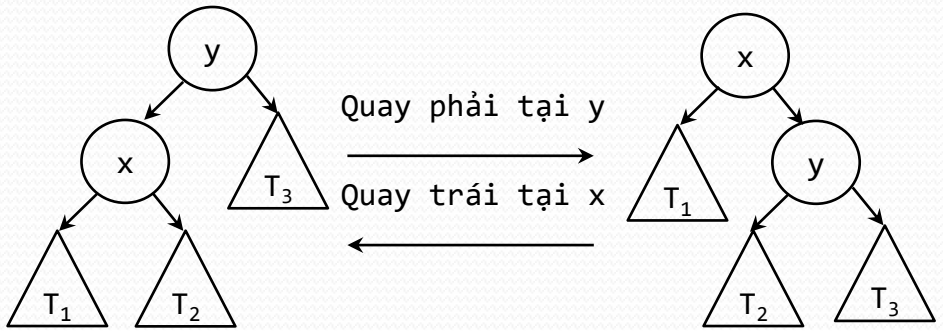
- Tổ chức dữ liệu: bổ sung trường lưu thông tin chiều cao của cây con tại nút gốc của cây con

```
struct BalanceTreeNode
{
    ElementType data;
    BalanceTreeNode *left;
    BalanceTreeNode *right;
    int height;
};
```



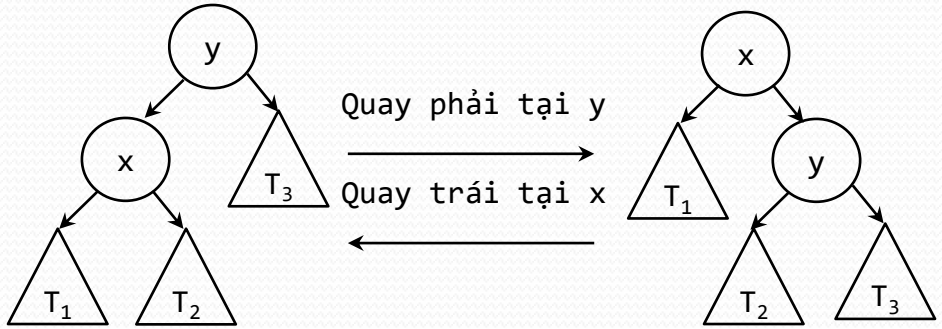
Cân bằng cây

```
BalanceTreeNode* rightRotate(BalanceTreeNode *y)
{
    BalanceTreeNode *x = y->left;
    BalanceTreeNode *T2 = x->right;

    // Thực hiện quay
    x->right = y;
    y->left = T2;

    // Cập nhật chiều cao
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Trả về nút gốc mới
    return x;
}
```



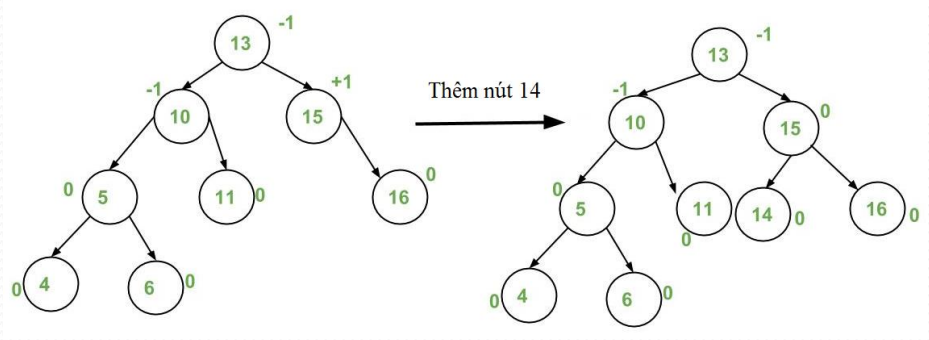

```
int height(Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}
```

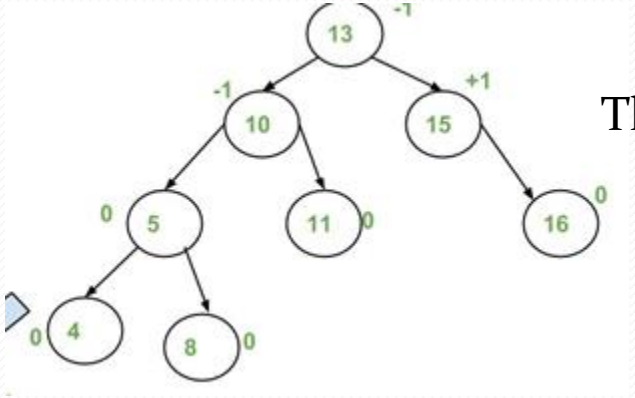
```
BalanceTreeNode* leftRotate(BalanceTreeNode *x)
{
    BalanceTreeNode *y = x->right;
    BalanceTreeNode *T2 = y->left;

    // Thực hiện quay
    y->left = x;
    x->right = T2;

    // Cập nhật chiều cao
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Trả về nút gốc mới
    return y;
}
```



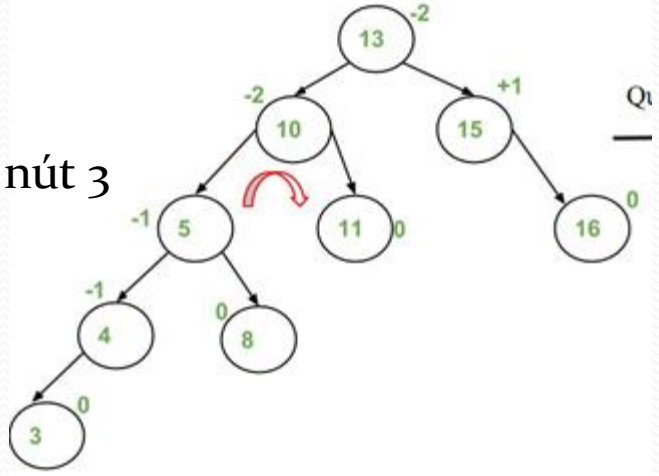
Thêm vào cây cân bằng

- Khi thêm một nút vào cây cân bằng có thể làm cây không còn cân bằng





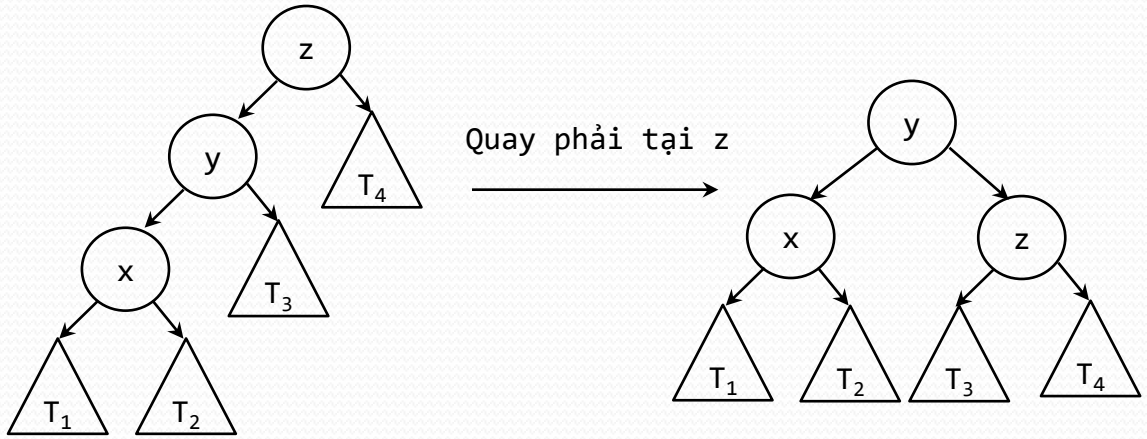
Thêm nút 3

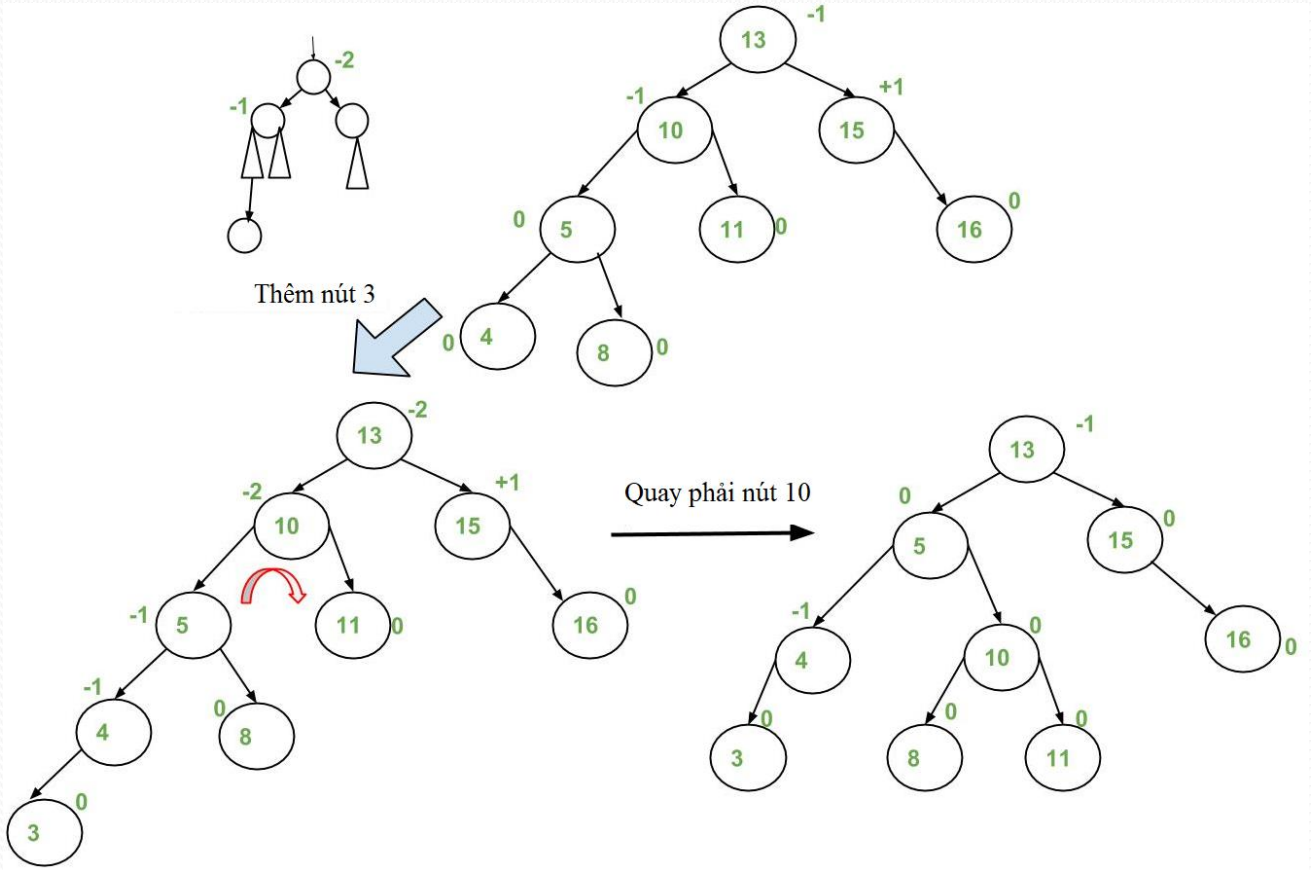


Thêm vào cây cân bằng

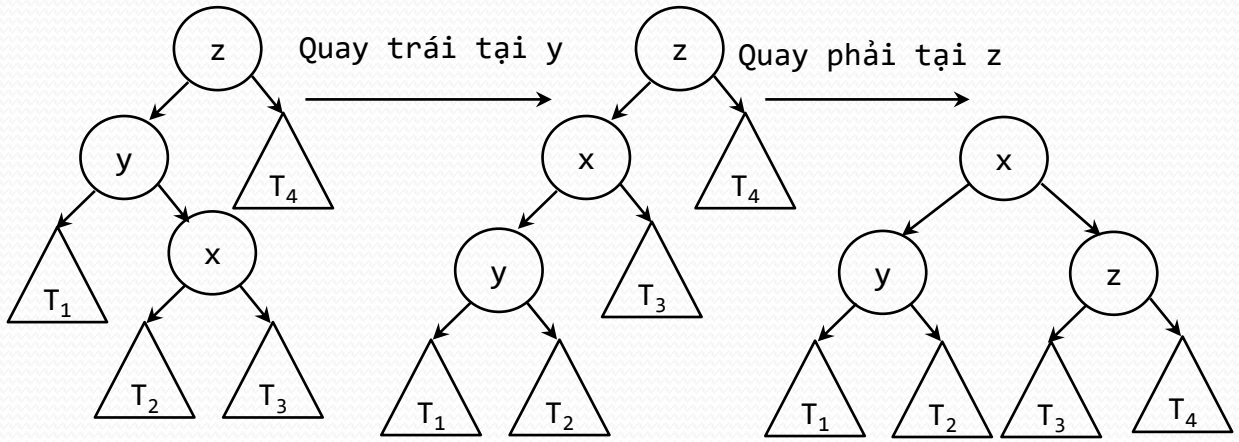
- Thực hiện thêm một nút w vào cây cân bằng như thêm vào cây tìm kiếm nhị phân
- Nếu sau khi thêm cây vẫn cân bằng thì dừng
- Nếu cây không cân bằng thì tìm nút z gần với w mà tại z cây không cân bằng
- Xét 4 trường hợp:

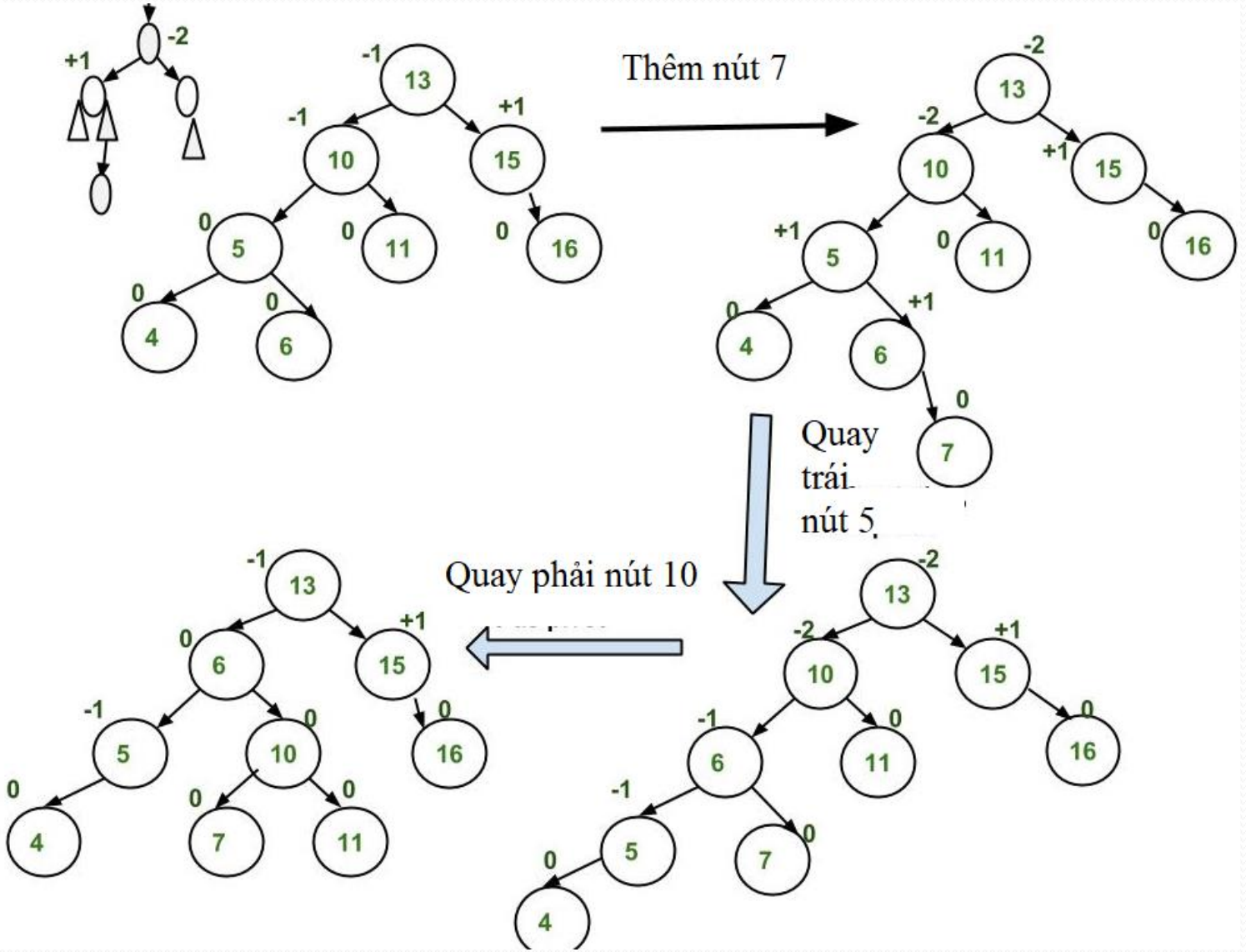
- a) Trường hợp Trái-Trái: y là nút con trái của z và y có nút con trái khác rỗng là x



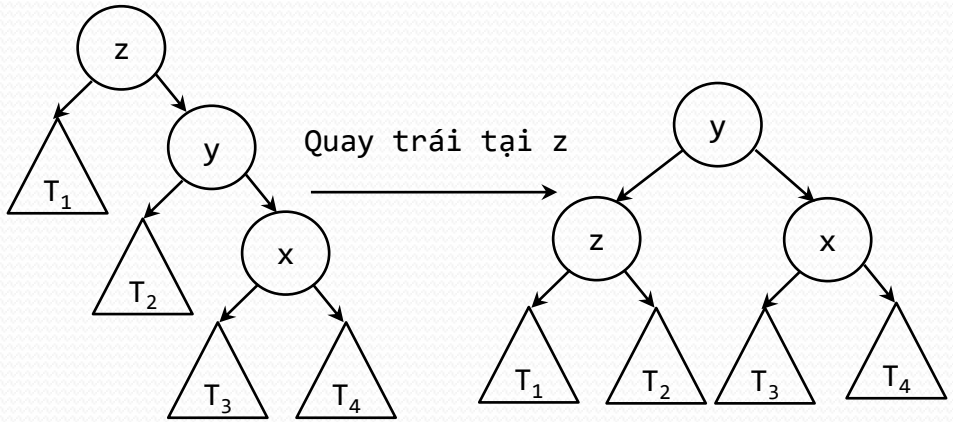

Thêm nút 3
Quay phải nút 10

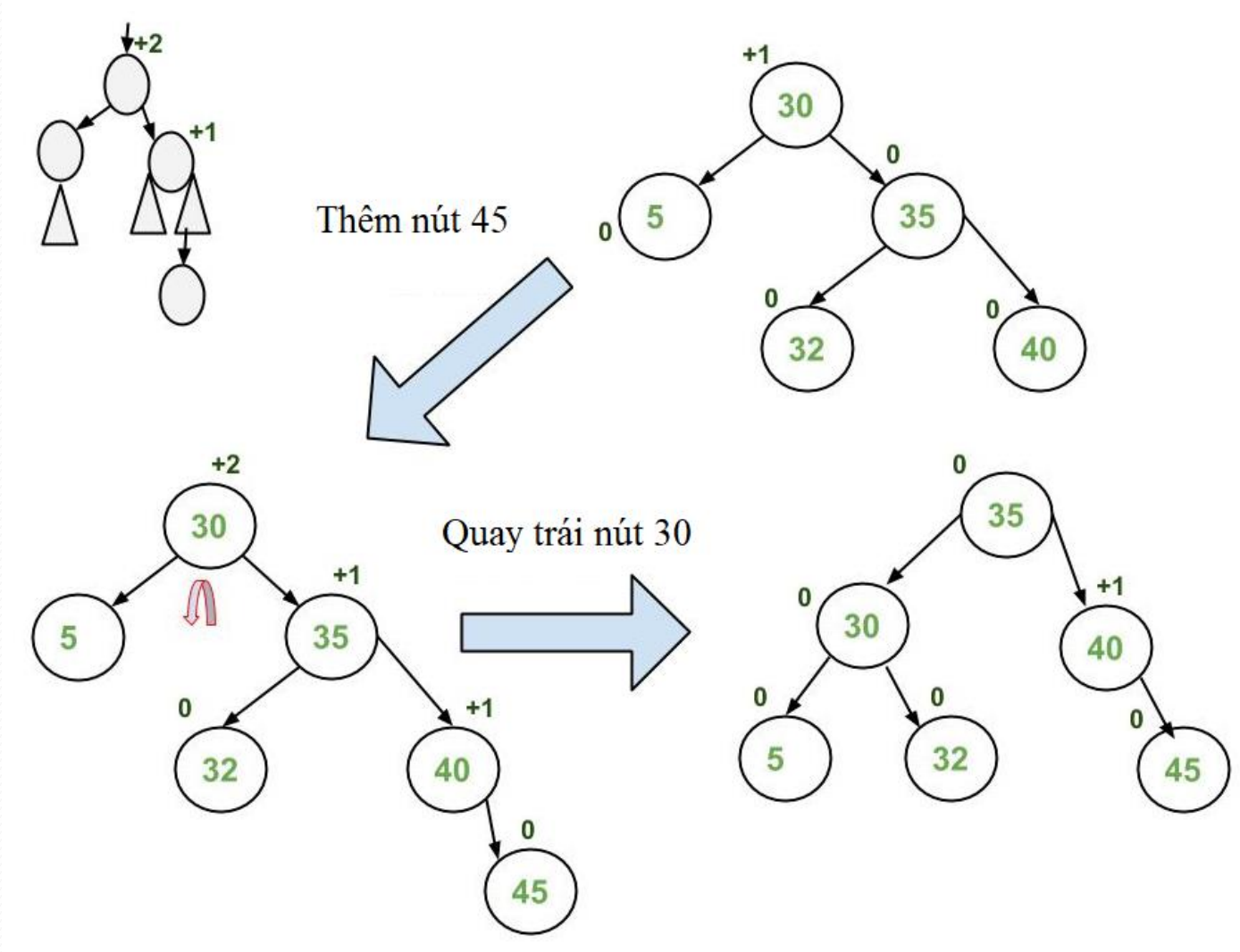
b) Trường hợp Trái-Phải: y là con trái của z và x là con phải của y



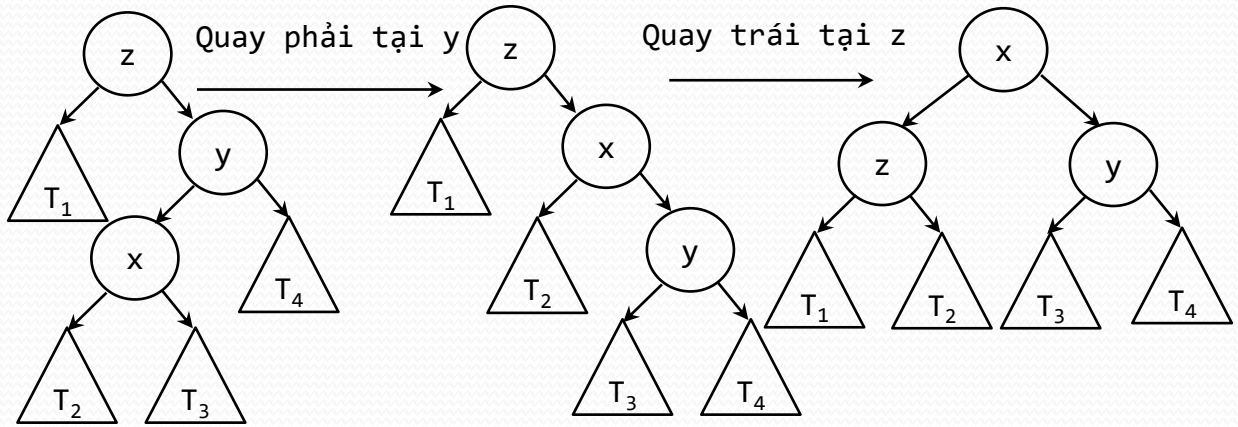

Thêm nút 7
Quay trái nút 5
Quay phải nút 10

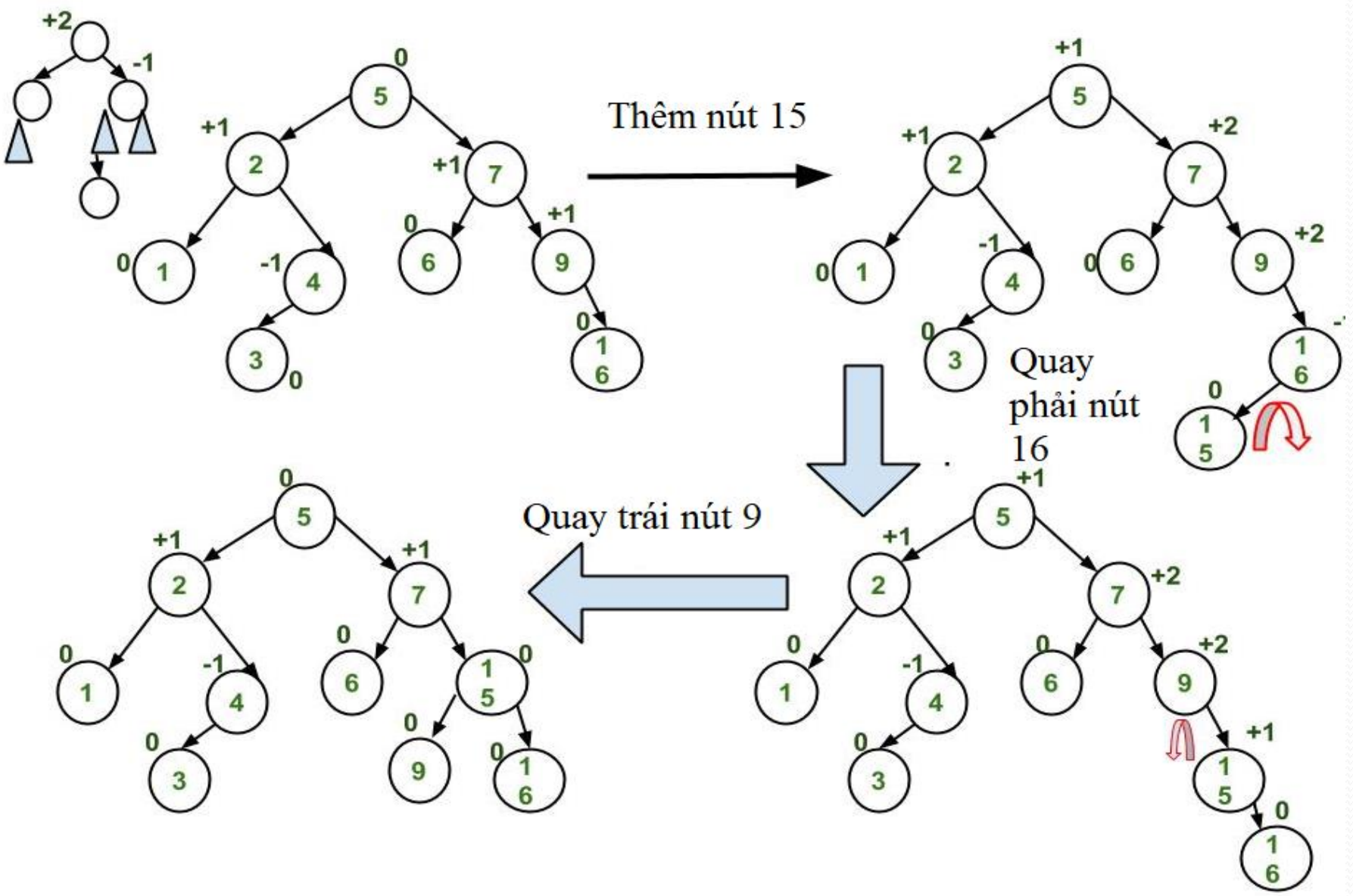
c) Trường hợp Phải-Phải: y là con phải của z và x là con phải của y.




+2
+1
Thêm nút 45
+1
30
0
5
35
0
32
0
40
+2
30
+1
5
35
0
32
+1
40
0
45
Quay trái nút 30
0
35
0
30
+1
40
0
5
0
32
0
45

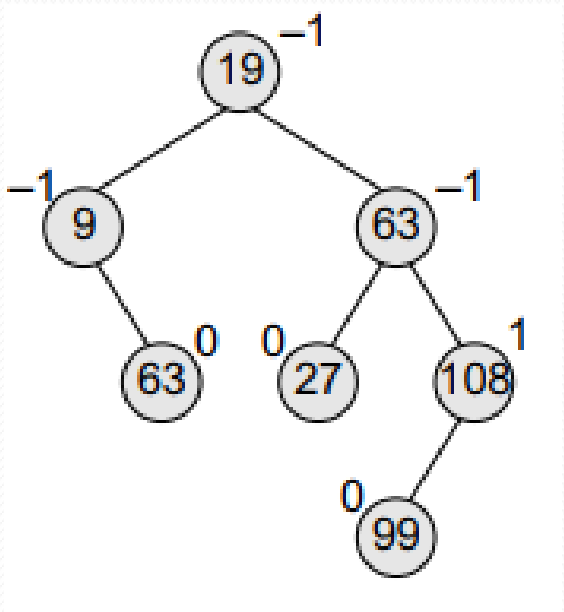
d) Trường hợp Phải-Trái: y là con phải của z và x là con trái của y.

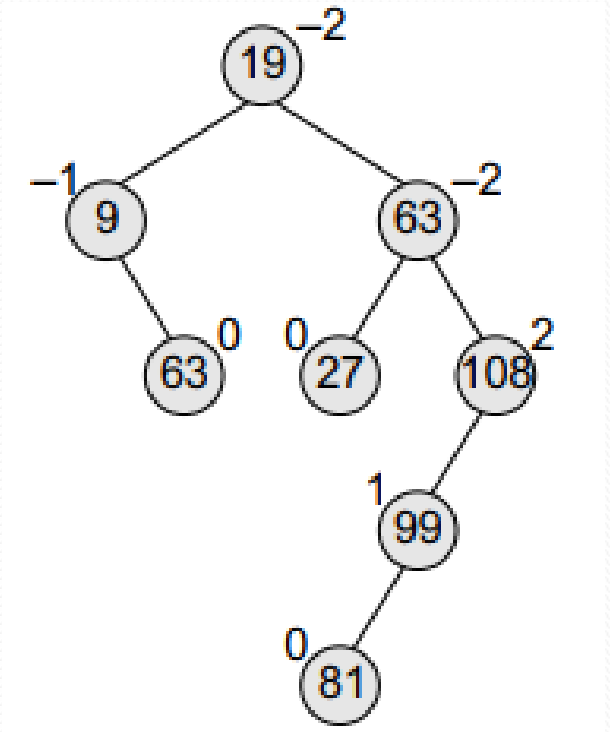


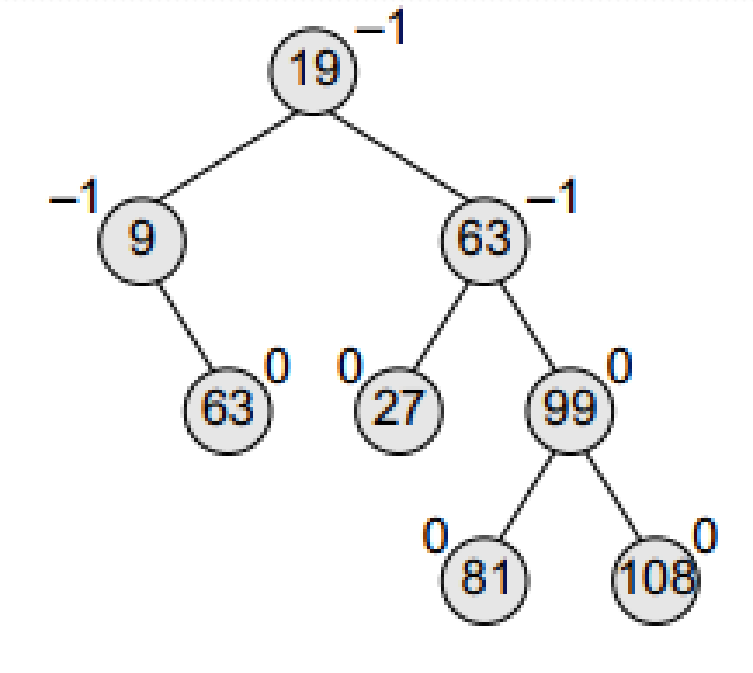

Thêm nút 15
Quay phải nút 16
Quay trái nút 9

Ví dụ

- Thêm số 81 vào cây

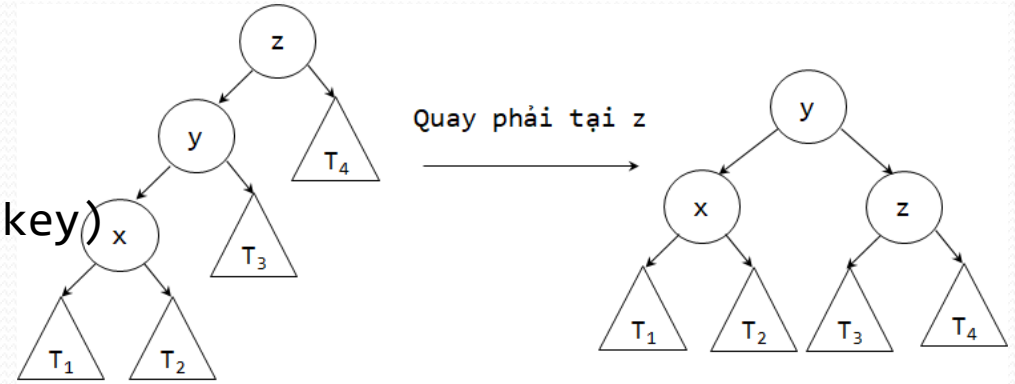


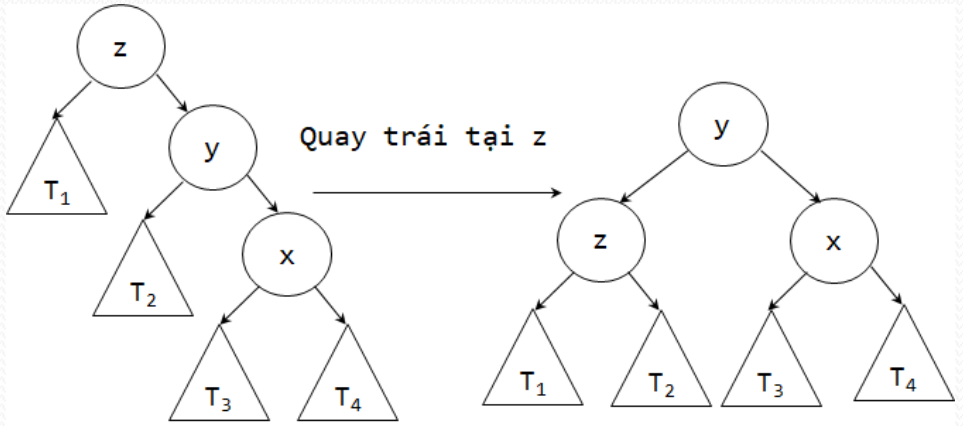




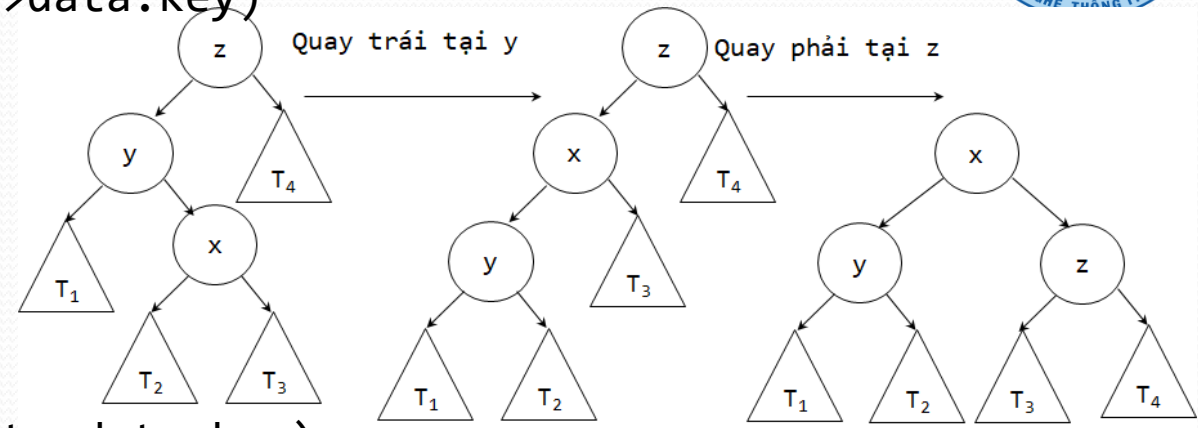
```
BalanceTreeNode* insert(BalanceTreeNode* node, ElementType element)
{
    /* 1. Thực hiện thêm như cây tìm kiếm nhị phân */
    if (node == nullptr){
     BalanceTreeNode* newNode = new BalanceTreeNode;
     newNode->data;
     newNode->height = 1;
     newNode->left = nullptr;
     newNode->right = nullptr;
       return newNode;
   }

    if (element.key < node->data.key)
        node->left = insert(node->left, element);
    else if (element.key > node->data.key)
        node->right = insert(node->right, element);
    else // Khóa đã có thì không thêm
        return node;
```

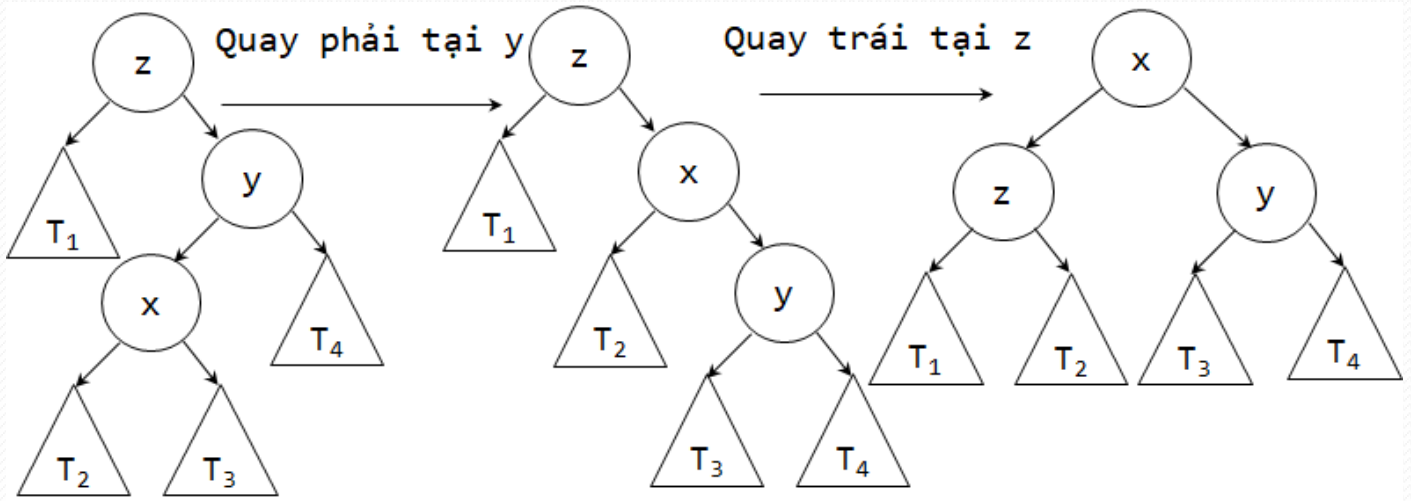
```
/* 3. Lấy hệ số cân bằng của node để kiểm tra tính cân bằng */
    int balance = getBalance(node);

    // Xét 4 trường hợp cây không cân bằng

    // Trường hợp Trái-Trái
    if (balance > 1 && element.key < node->left->data.key)
        return rightRotate(node);

    // Trường hợp Phải-Phải
    if (balance < -1 && element.key > node->right->data.key)
        return leftRotate(node);


// Hàm lấy hệ số cân bằng tại nút N
int getBalance(Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
```





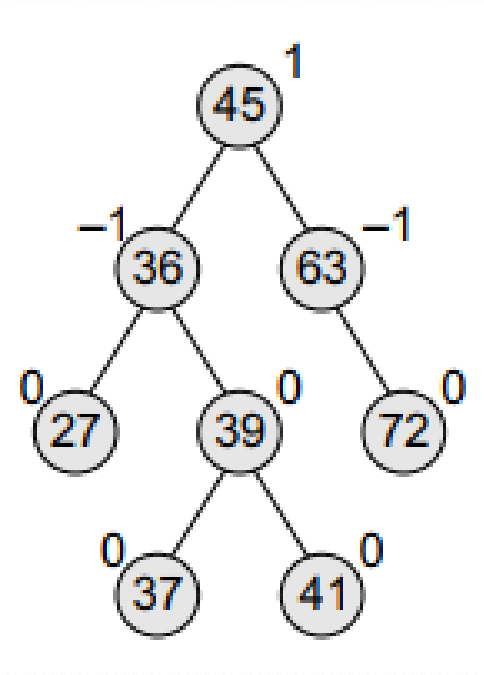
```
// Trường hợp Trái-Phải
    if (balance > 1 && element.key > node->left->data.key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Trường hợp Phải-Trái
    if (balance < -1 && element.key < node->right->data.key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* Trả về node nếu không thay đổi */
    return node;
}
```

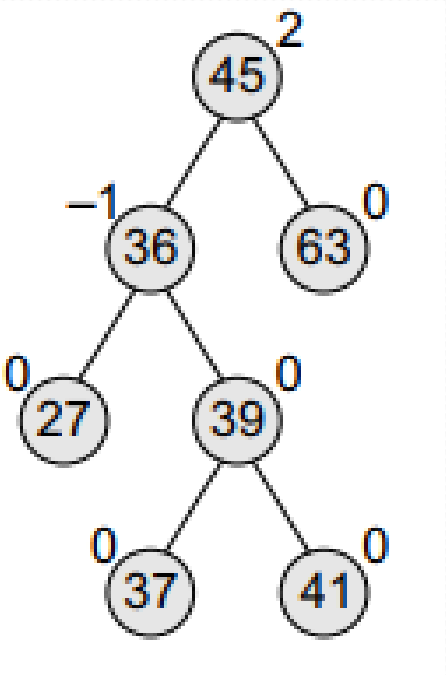


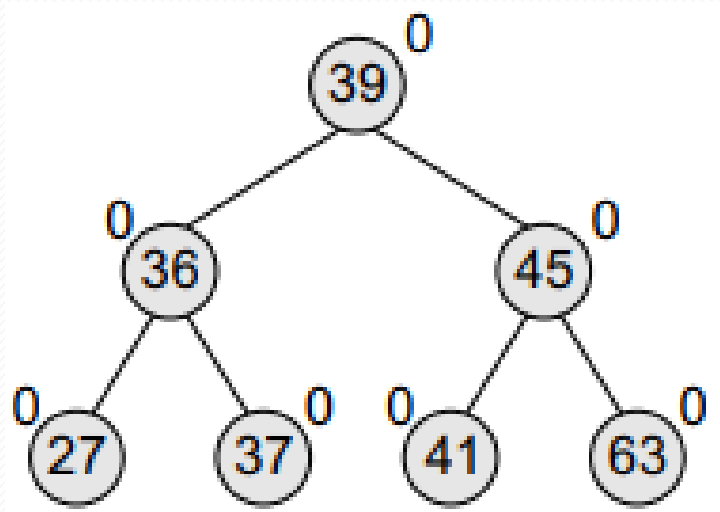


Ví dụ

- Xóa số 72 trong cây



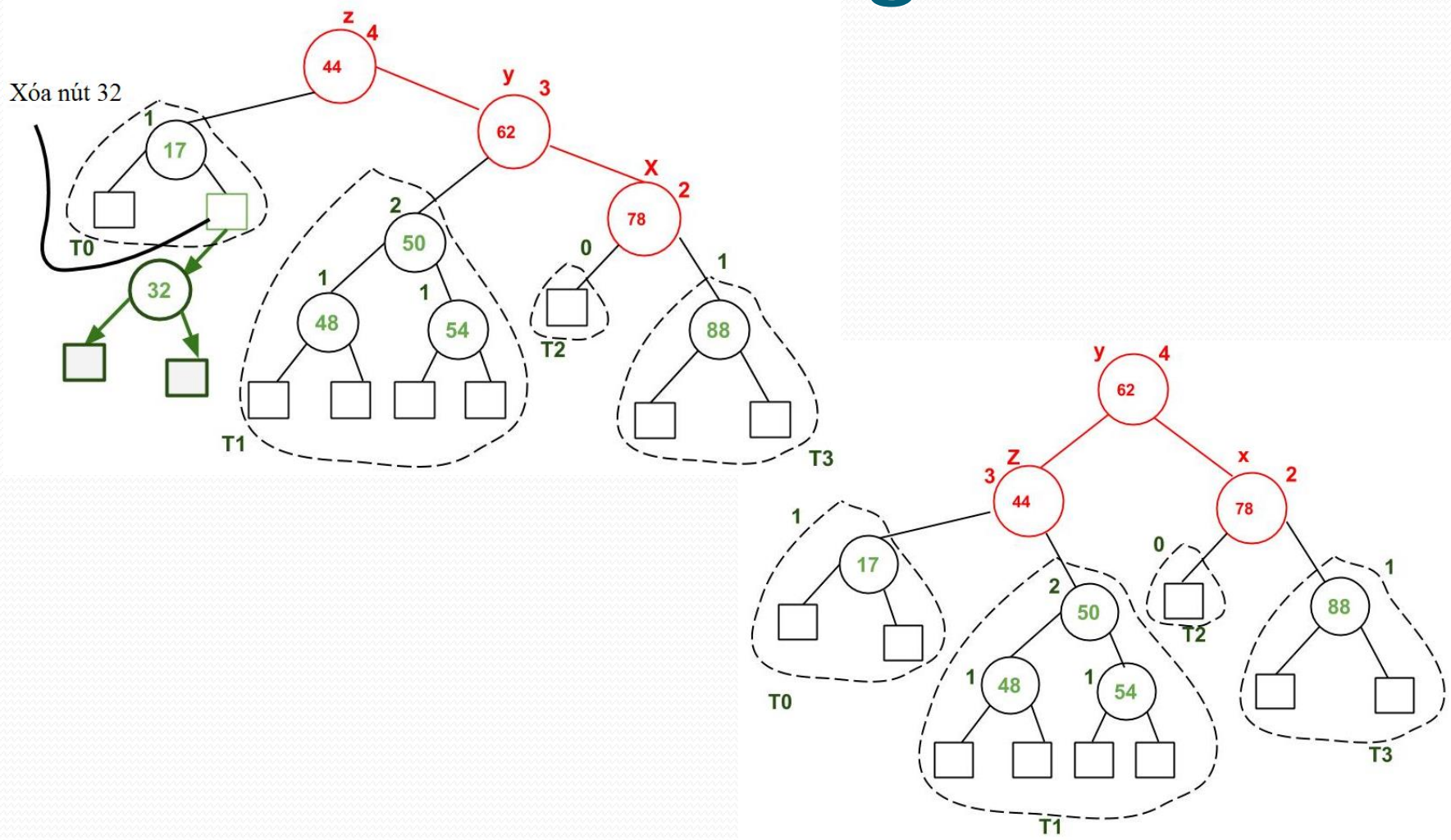




Độ phức tạp thuật toán

- Thao tác quay trái, quay phải độ phức tạp O(1)
- Thao tác thêm độ phức tạp O(h) với h là chiều cao cây cân bằng.
- Ta có h<= $\log_2 n + 1$, nên độ phức tạp thuật toán thêm vào cây cân bằng là O($\log_2$n)

Xoá nút trên cây cân bằng



```
BalanceTreeNode* deleteNode(BalanceTreeNode* root, KeyType key)
{

    // Bước 1: Thực hiện xóa trong cây tìm kiếm nhị phân
    if (root == nullptr)
        return root;

    // Nếu khóa cần xóa nhỏ hơn khóa tại nút gốc thì xóa ở
    // cây con trái
    if ( key < root->data.key )
        root->left = deleteNode(root->left, key);

    // Nếu khóa cần xóa lớn hơn khóa tại nút gốc thì xóa ở
    // cây con phải
    else if( key > root->data.key )
        root->right = deleteNode(root->right, key);
```

```
// Nếu khóa cần xóa trùng với khóa nút gốc
    else
    {
        // Nếu nút cần xóa không có hai cây con khác rỗng
        if((root->left == nullptr)||(root->right == nullptr))
        {
            Node *temp = root->left ? root->left : root->right;

            // Trường hợp nút lá
            if (temp == nullptr)
            {
                temp = root;
                root = nullptr;
            }
            else // Trường hợp có 1 nút con
            *root = *temp; // Copy nội dung của nút không rỗng
            delete temp;
        }
```

```
else
        {
            // Trường hợp nút có 2 nút con
            // tìm nút có khóa nhỏ nhất ở cây con phải
            Node* temp = minValueNode(root->right);

            // Copy dữ liệu nút temp lên root
            root->data = temp->data;

            // Xóa nút temp theo khóa từ cây con phải
            root->right = deleteNode(root->right, temp->data.key);
        }
    }

    // Nếu cây có 1 nút
    if (root == nullptr)
     return root;
```

```
// Bước 2: Cập nhật chiều cao của nút gốc
    root->height = 1 + max(height(root->left), height(root->right));

    // Bước 3: Lấy hệ số cân bằng của nút kiểm tra tính cân bằng
    int balance = getBalance(root);

    // Xét 4 trường hợp không cân bằng

    // Trường hợp Trái-Trái
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Trường hợp Trái-Phải
    if (balance > 1 && getBalance(root->left) < 0)
    {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
```

```
// Trường hợp Phải-Phải
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Trường hợp Phải-Trái
    if (balance < -1 && getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}
```

```
BalanceTreeNode * minValueNode(BalanceTreeNode* node)
{
    Node* current = node;

    /* Tìm nút con trái nhất */
    while (current->left != nullptr)
        current = current->left;

    return current;
```

Độ phức tạp thuật toán: $O(\log_2 n)$

Bài tập

Bài 6.1 Cho cây TKNP mỗi nút là 1 từ trong TDAV gồm từ tiếng Anh và nghĩa tiếng Việt. Khóa là từ tiếng Anh.

- Cài đặt các thao tác:
  1. Thêm 1 từ vào cây TD
  2. Tìm nghĩa của từ tiếng Anh x
  3. In từ điển theo thứ tự
  4. Xóa một từ trong cây TD

Bài 6.2 Sử dụng cây cân bằng để tổ chức dữ liệu cho các tập hợp số nguyên. Hãy trình bày thuật toán và cài đặt các thao tác:

a) Tạo một tập hợp rỗng

b) Kiểm tra một số nguyên x có thuộc tập hợp S không?

c) Liệt kê các số trong tập hợp S.

d) Kiểm tra S_1 có là tập con của S_2 không?

e) Tìm giao của hai tập hợp S_1 và S_2.

f) Tìm hợp của hai tập hợp S_1 và S_2.

g) Tìm hiệu của hai tập hợp S_1 và S_2.

Bài 6.3 Cho cây nhị phân mà mỗi nút là một số nguyên. Hãy trình bày thuật toán và cài đặt các thao tác sau:

a) Kiểm tra cây nhị phân có phải là cây tìm kiếm nhị phân không?

b) Kiểm tra cây nhị phân có phải là cây cân bằng không?

c) Kiểm tra cây nhị phân có phải là cây cân bằng hoàn toàn không?

Lời giải chi tiết

Bài 6.2a

- Tổ chức dữ liệu: như cây cân bằng.

```
struct Node{
  int key;
  Node *left, *right;
  int height;
}
```

- Kiểm tra x thuộc S?
- Sử dụng thao tác tìm x trong cây có nút gốc là tập S

```
bool in(int x, Node *S)
{
  if (!S)
     return false;
  else
     if (S->key == x)
        return true;
     else
        if (S->key > x)
        return in(x, S->left);
        else
        return in(x, S->right);
}
```

Bài 6.2b

- Duyệt và in các nút trong cây biểu diễn tập hợp

Bài 6.2c

Thuật toán kiểm tra tập con:

Input: Cây S1, S2

Output: True nếu S1 là tập con S2

False mếu S1 không là tập con S2

Action:

Nếu S1 rỗng thì trả về True

Ngược lại

+ Trả về (con trái của S1 con S2) và (con phải của S1 con S2) và (số tại nút gốc của S1 thuộc S2)

```
bool subset(Node* S1, Node* S2)
{
  if (!S1)
     return true;
  else
     return subset(S1->left, S2) &&
     subset(S1->right, S2) && in(S1->key, S2);
}
```

Bài 6.2d

Input: hai tập hợp S1, S2

Output: S3 là S1 giao S2

Action:

Nếu S1 khác rỗng thì:

Nếu số tại gốc S1 thuộc S2 thì thêm số tại gốc S1 vào S3

Tìm giao của con trái S1 với S2 đưa vào S3

Tìm giao của con phải S1 với S2 đưa vào S3

```
Node* intersectionSet(Node* S1, Node* S2, Node* S3)
{
  if (S1)
  {
     if (in(S1->key, S2))
        insert(S3, S1->key);
     S3 = intersectionSet(S1->left, S2, S3);
     S3 = intersectionSet(S1->right, S2, S3);
  }
  return S3;
}
```

Bài 6.2e

Input: S1, S2

Output: S3 là hợp S1 và S2

Action:

S3 = rỗng

Thêm các số của S1 vào S3

Thêm các cố của S2 vào S3

Trả về S3

Input: tập S1, S2

Output: S2 sau khi thêm S1 vào

Action:

Nếu S1 khác rỗng:

thêm số tại gốc S1 vào S2

thêm tập hợp con trái của S1 vào S2

thêm tập hợp con phải của S1 vào S2

```
Node* append(Node* S1, Node* S2)
{
  if (S1)
  {
     S2 = insert(S2, S1->key);
     S2 = append(S1->left, S2);
     S2 = append(S1->right, S2);
  }
  return S2;
}

Node* unionSet(Node* S1, Node*S2)
{
  Node* S3 = NULL;
  S3 = append(S1, S3);
  S3 = append(S2, S3);
  return S3;
}
```

Tổng kết

- Cây tìm kiếm nhị phân giúp cho việc tìm kiếm nhanh
- Cây cân bằng là cây TKNP tránh trường hợp cây suy biến là tổ chức dữ liệu tốt cho các tập hợp.