CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

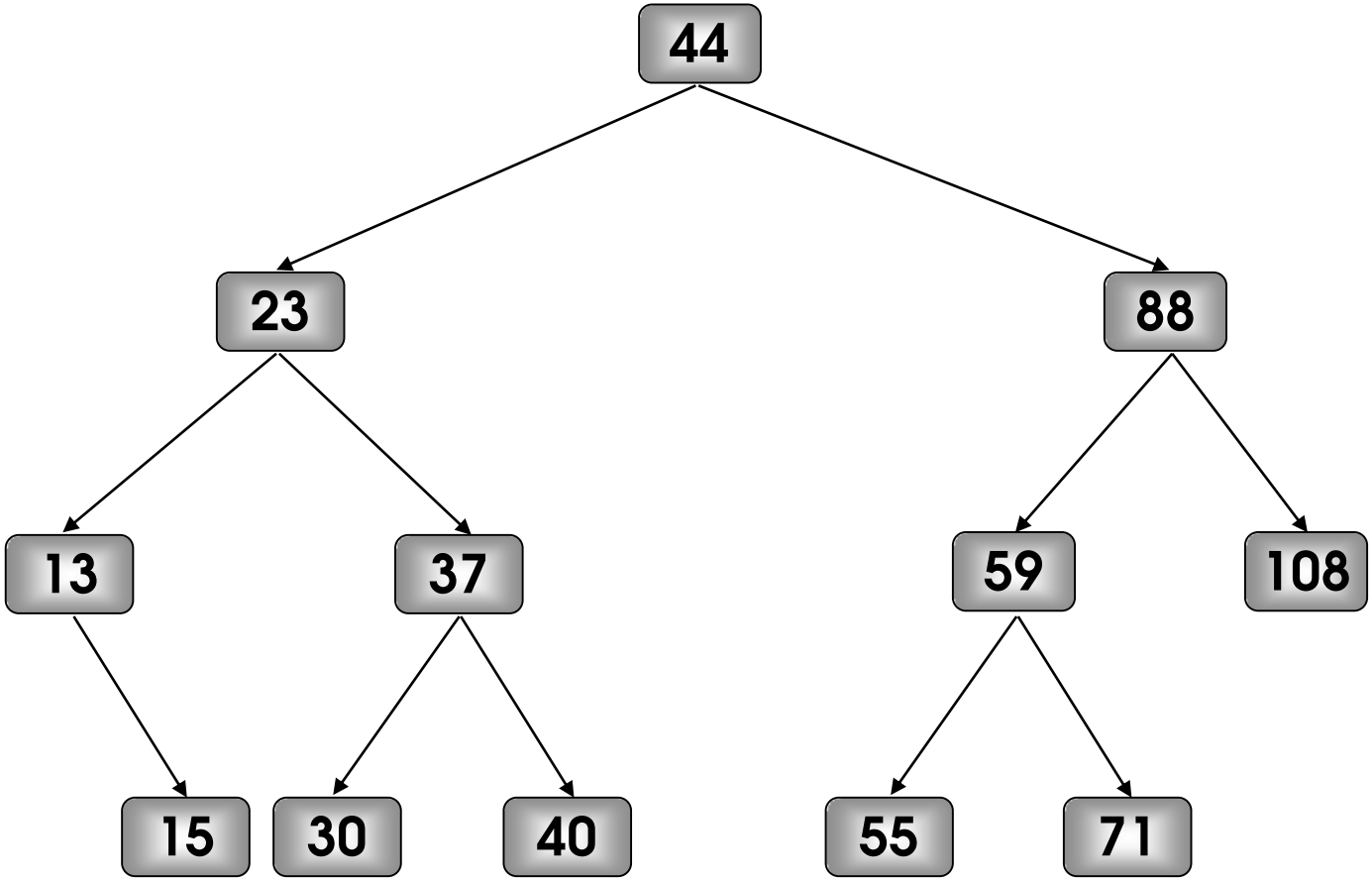
Nội dung

- Cấu trúc cây (Tree)
- Cấu trúc cây nhị phân (*Binary Tree*)
- Cấu trúc cây nhị phân tìm kiếm (*Binary Search Tree*)
- Cấu trúc cây nhị phân tìm kiếm cân bằng (AVL Tree)

AVL Tree - Định nghĩa

- Cây nhị phân tìm kiếm cân bằng là cây mà tại mỗi nút của nó độ cao của cây con trái và của cây con phải chênh lệch không quá một.

AVL Tree – Ví dụ



AVL Tree

- Lịch sử cây cân bằng (AVL Tree):
  - AVL là tên viết tắt của các tác giả người Nga đã đưa ra định nghĩa của cây cân bằng Adelson-Velskii và Landis (1962)
  - Từ cây AVL, người ta đã phát triển thêm nhiều loại CTDL hữu dụng khác như cây đỏ-đen (Red-Black Tree), B-Tree, …
- Cây AVL có chiều cao O(log2(n))

AVL Tree

- Chỉ số cân bằng của một nút:
  - Định nghĩa: Chỉ số cân bằng của một nút là hiệu của chiều cao cây con phải và cây con trái của nó.
  - Đối với một cây cân bằng, chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị sau đây:
    - CSCB(p) = 0 $\Leftrightarrow$ Độ cao cây trái (p) = Độ cao cây phải (p)
    - CSCB(p) = 1 $\Leftrightarrow$ Độ cao cây trái (p) < Độ cao cây phải (p)
    - CSCB(p) =-1 $\Leftrightarrow$ Độ cao cây trái (p) > Độ cao cây phải (p)
- Để tiện trong trình bày, chúng ta sẽ ký hiệu như sau: p->balFactor = CSCB(p);
- Độ cao cây trái (p) ký hiệu là hL
- Độ cao cây phải(p) ký hiệu là hR

AVL Tree – Biểu diễn


```
#define LH    -1    /* Cây con trái cao hơn    */
#define EH     0    /* Hai cây con bằng nhau */
#define RH     1    /* Cây con phải cao hơn   */

struct AVLNode{
  char          balFactor;  // Chỉ số cân bằng
  DataType      data;
  tagAVLNode*   pLeft;
  tagAVLNode*   pRight;
};
typedef AVLNode*   AVLTree;
```

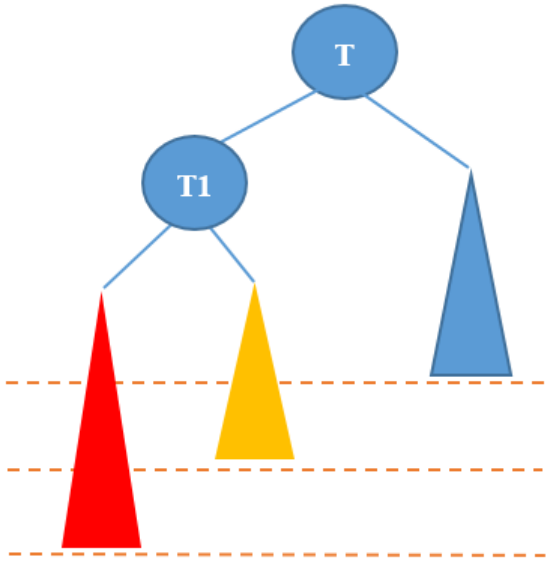
AVL Tree – Biểu diễn

- Trường hợp thêm hay hủy một phần tử trên cây có thể làm cây tăng hay giảm chiều cao, khi đó phải cân bằng lại cây
- Việc cân bằng lại một cây sẽ phải thực hiện sao cho chỉ ảnh hưởng tối thiểu đến cây nhằm giảm thiểu chi phí cân bằng
- Các thao tác đặc trưng của cây AVL:
  - Thêm một phần tử vào cây AVL
  - Hủy một phần tử trên cây AVL
  - Cân bằng lại một cây vừa bị mất cân bằng

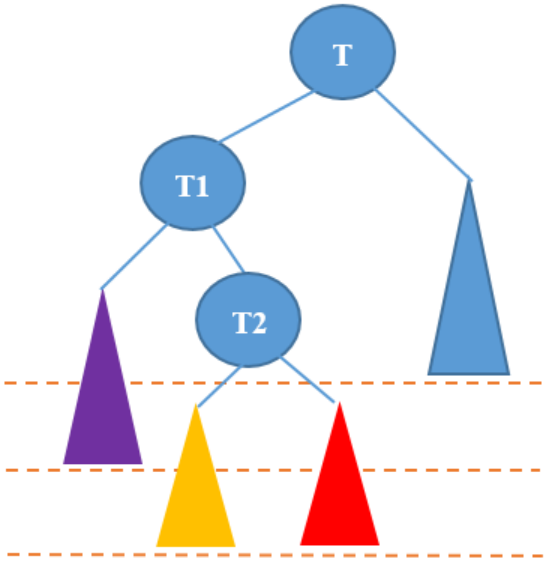
AVL Tree

- Các trường hợp mất cân bằng:
  - Ta sẽ không khảo sát tính cân bằng của 1 cây nhị phân bất kỳ mà chỉ quan tâm đến các khả năng mất cân bằng xảy ra khi thêm hoặc hủy một nút trên cây AVL
  - Như vậy, khi mất cân bằng, độ lệch chiều cao giữa 2 cây con sẽ là 2
  - Có 6 khả năng sau:
    - Trường hợp 1 - Cây T lệch về bên trái : 3 khả năng
    - Trường hợp 2 - Cây T lệch về bên phải: 3 khả năng

Trường hợp 1: lệch trái

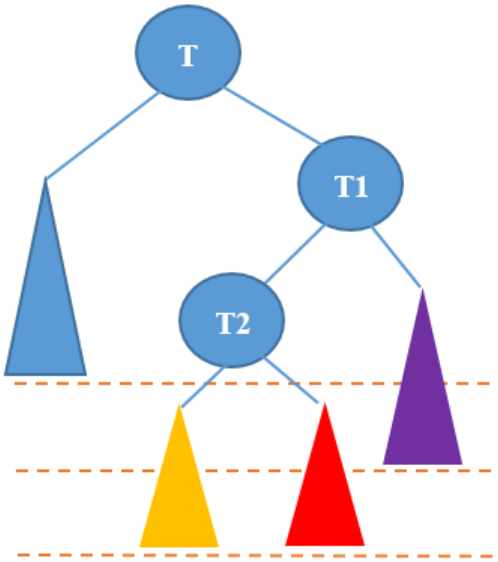


1.1 Lệch trái – trái

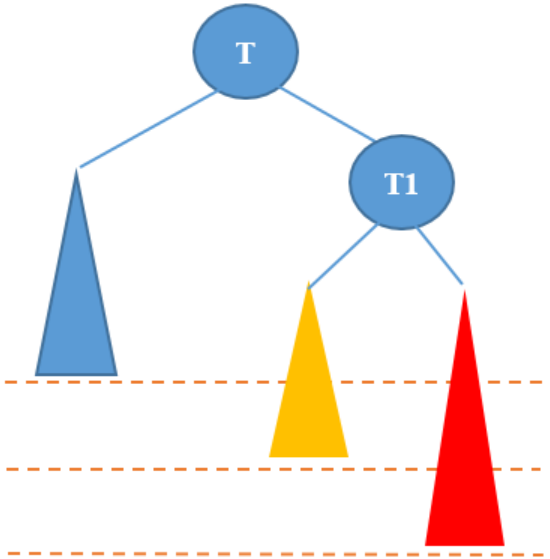


1.2 Lệch trái – phải

Trường hợp 2: lệch phải

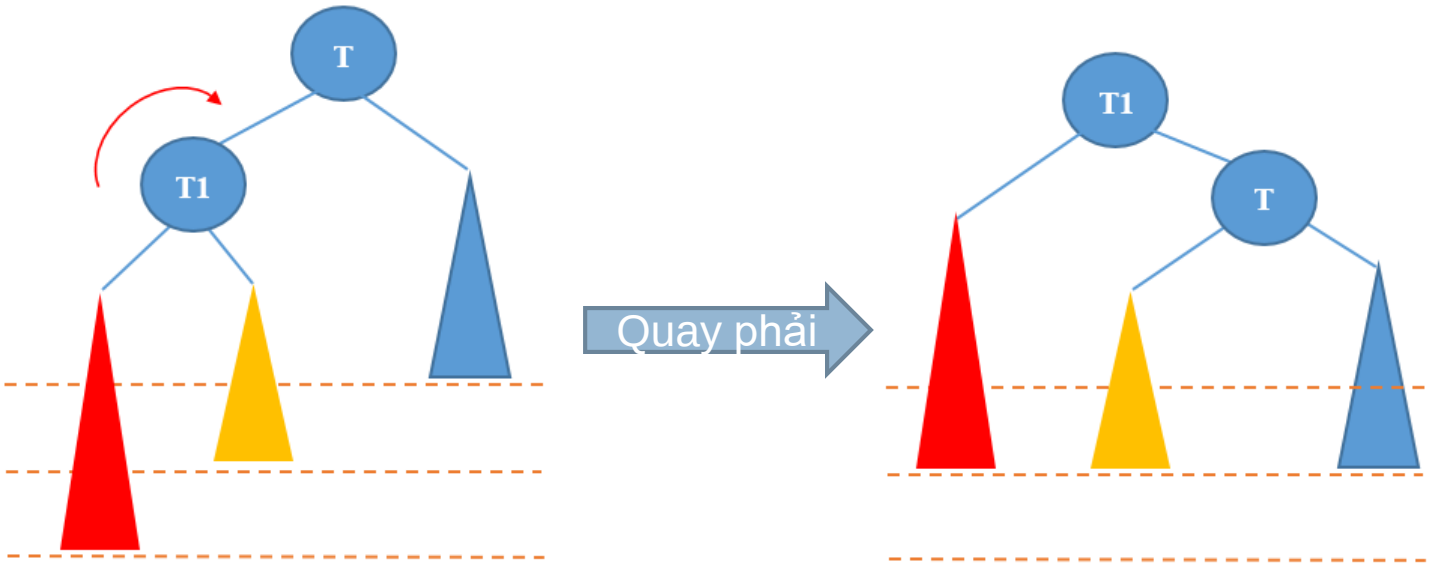


2.1 Lệch phải – trái

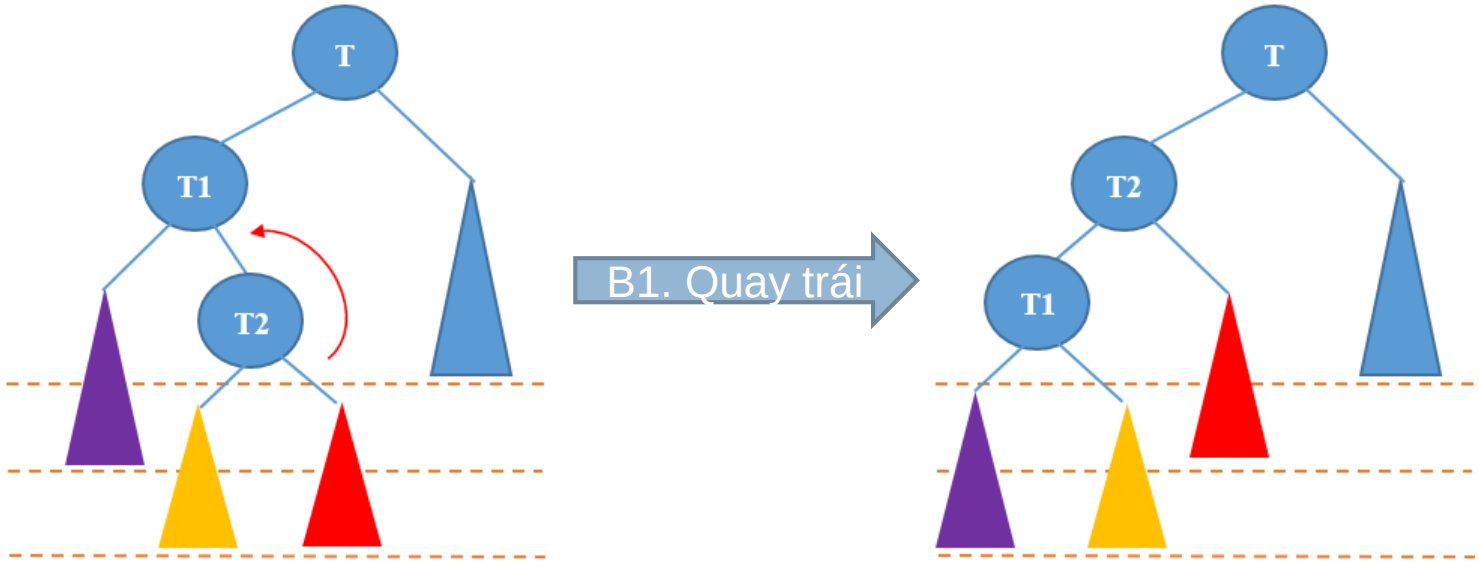


2.2 Lệch phải – phải

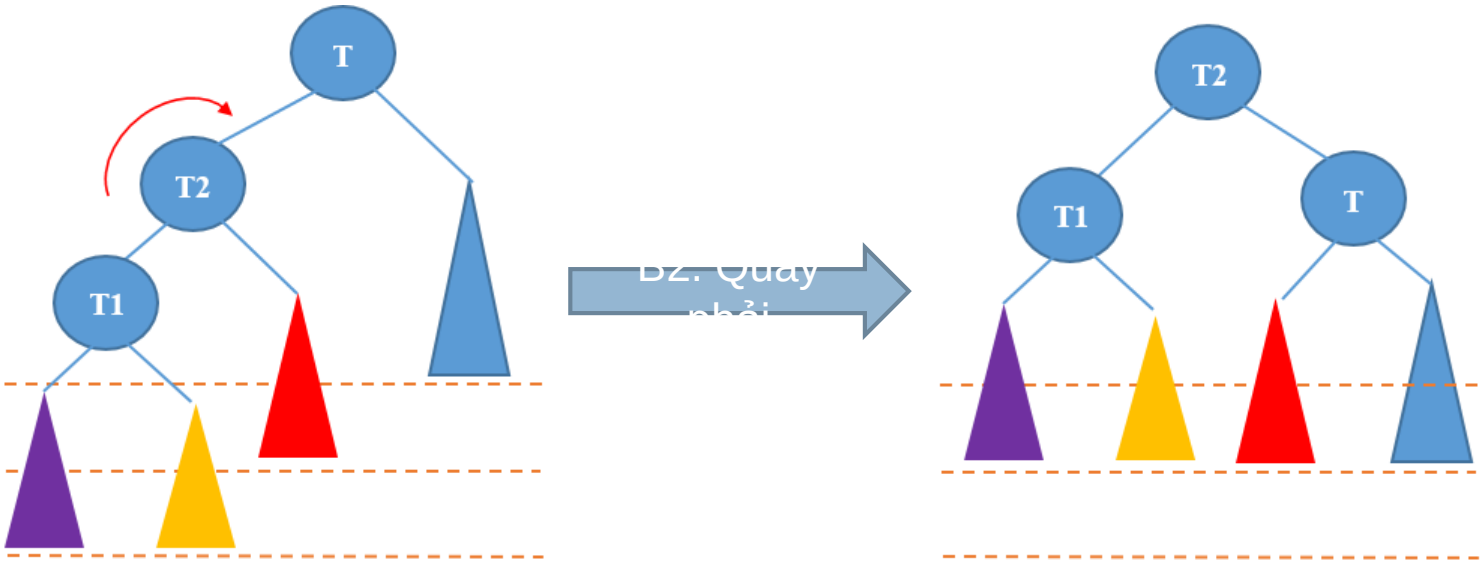
Lệch trái - trái



Lệch trái – phải:
Thực hiện quay kép



Lệch trái – phải:
Thực hiện quay kép



Bài tập

Cho dãy số: **45, 46, 70, 11, 13, 42, 21, 9, 25, 4, 10, 15, 30, 32**

Trình bày từng bước quá trình xây dựng cây AVL

Bài tập

Vẽ cây AVL cho dãy số sau: **1, 9, 2, 15, 12, 8, 4, 11, 7, 19, 18, 3, 15, 6, 21, 13, 10**

AVL Tree - Cân bằng lại cây AVL

- Quay đơn Left-Left:

```
void rotateLL(AVLTree &T) //quay đơn Left-Left
{
    AVLNode*  T1 = T->pLeft;
    T->pLeft          = T1->pRight;
    T1->pRight        = T;
    switch(T1->balFactor) {
    case LH: T->balFactor  = EH;
                      T1->balFactor = EH;
                      break;
    case EH: T->balFactor  = LH;
                      T1->balFactor = RH;
                      break;
    }
    T         = T1;
}
```

AVL Tree - Cân bằng lại cây AVL

- Quay đơn Right-Right:

```
void rotateRR (AVLTree &T) //quay đơn Right-Right
{
    AVLNode*        T1 = T->pRight;
    T->pRight       = T1->pLeft;
    T1->pLeft       = T;
    switch(T1->balFactor)   {
    case RH: T->balFactor = EH;
                    T1->balFactor= EH;
                    break;
    case EH: T->balFactor = RH;
                    T1->balFactor= LH;
                    break;
    }
    T = T1;
}
```

AVL Tree - Cân bằng lại cây AVL

- Quay kép Left-Right:

```
void rotateLR(AVLTree &T)//quay kép Left-Right
{       AVLNode*    T1 = T->pLeft;
        AVLNode*    T2 = T1->pRight;
        T->pLeft    = T2->pRight;
        T2->pRight  = T;
        T1->pRight  = T2->pLeft;
        T2->pLeft   = T1;
        switch(T2->balFactor)   {
          case LH: T->balFactor  = RH; T1->balFactor = EH; break;
          case EH: T->balFactor  = EH; T1->balFactor = EH; break;
          case RH: T->balFactor  = EH; T1->balFactor = LH; break;
        }
        T2->balFactor = EH;
        T = T2;
}
```

AVL Tree - Cân bằng lại cây AVL

- Quay kép Right-Left

```
void rotateRL(AVLTree &T)       //quay kép Right-Left
{       AVLNode*        T1 = T->pRight;
        AVLNode*        T2 = T1->pLeft;
        T->pRight       = T2->pLeft;
        T2->pLeft       = T;
        T1->pLeft       = T2->pRight;
        T2->pRight      = T1;
        switch(T2->balFactor)   {
          case RH: T->balFactor = LH; T1->balFactor = EH; break;
          case EH: T->balFactor = EH; T1->balFactor = EH; break;
          case LH: T->balFactor = EH; T1->balFactor = RH; break;
        }
        T2->balFactor = EH;
        T = T2;
}
```

AVL Tree - Cân bằng lại cây AVL

- Cân bằng khi cây bị lệch về bên trái:

```
int balanceLeft(AVLTree &T)
//Cân bằng khi cây bị lệch về bên trái
{
        AVLNode*        T1 = T->pLeft;

        switch(T1->balFactor)           {
        case LH:        rotateLL(T); return 2;
        case EH:        rotateLL(T); return 1;
        case RH:        rotateLR(T); return 2;
        }
        return 0;
}
```

AVL Tree - Cân bằng lại cây AVL

- Cân bằng khi cây bị lệch về bên phải

```
int balanceRight(AVLTree &T )
//Cân bằng khi cây bị lệch về bên phải
{
        AVLNode*        T1 = T->pRight;

        switch(T1->balFactor)           {
        case LH:        rotateRL(T); return 2;
        case EH:        rotateRR(T); return 1;
        case RH:        rotateRR(T); return 2;
        }
        return 0;
}
```

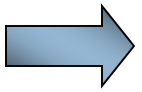
AVL Tree - Thêm một phần tử trên cây AVL

- Việc thêm một phần tử vào cây AVL diễn ra tương tự như trên CNPTK
- Sau khi thêm xong, nếu chiều cao của cây thay đổi, từ vị trí thêm vào, ta phải lần ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng không. Nếu có, ta phải cân bằng lại ở nút này
- Việc cân bằng lại chỉ cần thực hiện 1 lần tại nơi mất cân bằng
- Hàm *insertNode* trả về giá trị –1, 0, 1 khi không đủ bộ nhớ, gặp nút cũ hay thành công. Nếu sau khi thêm, chiều cao cây bị tăng, giá trị 2 sẽ được trả về

int insertNode(AVLTree &T, DataType X)

AVL Tree - Thêm một phần tử trên cây AVL

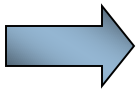

```
int insertNode(AVLTree &T, DataType X)
{  int res;
   if (T)
   {    if (T->key == X) return 0; //đã có
        if (T->key > X)
        {    res       = insertNode(T->pLeft, X);
             if(res < 2) return res;
             switch(T->balFactor)
             {  case RH: T->balFactor = EH;  return 1;
                case EH: T->balFactor = LH;  return 2;
                case LH: balanceLeft(T);     return 1;
             }
        }
.....................................................
}
```



insertNode2

AVL Tree - Thêm một phần tử trên cây AVL


```
int insertNode(AVLTree &T, DataType X)
{
.....................................................
       else // T->key < X
       {   res      = insertNode(T-> pRight, X);
           if(res < 2) return res;
           switch(T->balFactor)
           { case LH: T->balFactor    = EH; return 1;
             case EH: T->balFactor    = RH; return 2;
             case RH: balanceRight(T);      return 1;
             }
       }
.....................................................
}
```



insertNode3

AVL Tree - Thêm một phần tử trên cây AVL


```
int insertNode(AVLTree &T, DataType X)
{
.....................................
  ..........
  T  = new TNode;
  if(T == NULL) return -1; //thiếu bộ nhớ
  T->key = X;
  T->balFactor = EH;
  T->pLeft = T->pRight = NULL;
  return 2; // thành công, chiều cao tăng
}
```

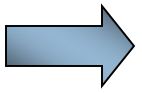
AVL Tree - Hủy một phần tử trên cây AVL

- Cũng giống như thao tác thêm một nút, việc hủy một phần tử X ra khỏi cây AVL thực hiện giống như trên CNPTK
- Sau khi hủy, nếu tính cân bằng của cây bị vi phạm ta sẽ thực hiện việc cân bằng lại
- Tuy nhiên việc cân bằng lại trong thao tác hủy sẽ phức tạp hơn nhiều do có thể xảy ra phản ứng dây chuyền
- Hàm *delNode* trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây. Nếu sau khi hủy, chiều cao cây bị giảm, giá trị 2 sẽ được trả về:

```
int delNode(AVLTree &T, DataType X)
```

AVL Tree - Hủy một phần tử trên cây AVL

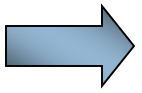

```
int delNode(AVLTree &T, DataType X)
{  int res;
   if(T==NULL)       return 0;
   if(T->key > X)
   {   res = delNode (T->pLeft, X);
       if(res < 2)   return res;
       switch(T->balFactor)
       {   case LH: T->balFactor = EH; return 2;
           case EH: T->balFactor = RH; return 1;
           case RH: return balanceRight(T);
       }
   } // if(T->key > X)
.....
}
```



delNode2

AVL Tree - Hủy một phần tử trên cây AVL


```
int delNode(AVLTree &T, DataType X)
{
.....................................................
  if(T->key < X)
  {  res     = delNode (T->pRight, X);
     if(res < 2)   return res;
     switch(T->balFactor)
     {  case RH: T->balFactor = EH; return 2;
        case EH: T->balFactor = LH; return 1;
        case LH: return balanceLeft(T);
     }
  } // if(T->key < X)
.....................................................
}
```



delNode3

AVL Tree - Hủy một phần tử trên cây AVL


```
int delNode(AVLTree &T, DataType X)
{.....................................................
   else  //T->key == X
   {  AVLNode*   p = T;
      if(T->pLeft == NULL)        {  T = T->pRight; res = 2; }
      else if(T->pRight == NULL) {  T = T->pLeft;  res = 2; }
           else //T có đủ cả 2 con
           {  res = searchStandFor(p,T->pRight);
              if(res < 2) return res;
              switch(T->balFactor)
              {  case RH: T->balFactor = EH; return 2;
                 case EH: T->balFactor = LH; return 1;
                 case LH: return balanceLeft(T);
              }
           }
      delete p;  return res;
   }
}
```

AVL Tree - Hủy một phần tử trên cây AVL


```
int searchStandFor(AVLTree &p, AVLTree &q)
//Tìm phần tử thế mạng
{  int res;
   if(q->pLeft)
   {  res     = searchStandFor(p, q->pLeft);
      if(res < 2)    return res;
      switch(q->balFactor)
      {  case LH: q->balFactor = EH; return 2;
         case EH: q->balFactor = RH; return 1;
         case RH: return balanceRight(T);
      }
   }  else
   {  p->key = q->key; p = q; q = q->pRight; return 2;
   }
}
```

AVL Tree

- Nhận xét:
  - Thao tác thêm một nút có độ phức tạp O(1)
  - Thao tác hủy một nút có độ phức tạp O(h)
  - Với cây cân bằng trung bình 2 lần thêm vào cây thì cần một lần cân bằng lại; 5 lần hủy thì cần một lần cân bằng lại

AVL Tree

- Nhận xét:
  - Việc hủy 1 nút có thể phải cân bằng dây chuyền các nút từ gốc cho đên phần tử bị hủy trong khi thêm vào chỉ cần 1 lần cân bằng cục bộ
  - Độ dài đường tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn $\log_2 n$, nhưng việc cân bằng lại đơn giản hơn nhiều
  - Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số nút trên cây là bao nhiêu