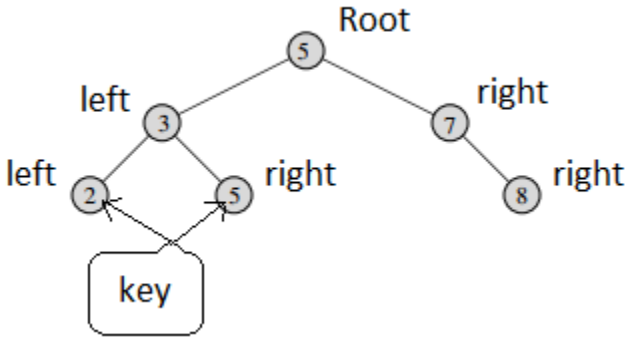
CHƯƠNG 11: Cây Nhị Phân Tìm kiếm (binary search tree, BST)

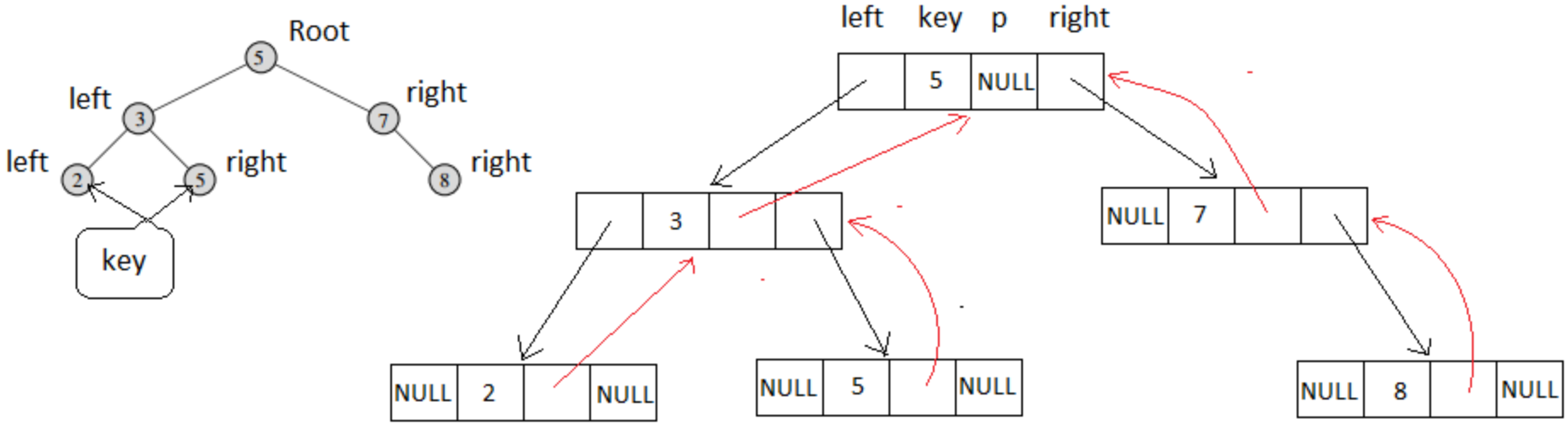
11.1 Định nghĩa:

Gọi x là một nút (node) trong BST. Nếu y là nút trong cây con bên trái của x thì $key[y] \leq key[x]$. Nếu y là nút trong cây con bên phải của x thì $key[x] \leq key[y]$.



11.2 Kiểu dữ liệu:

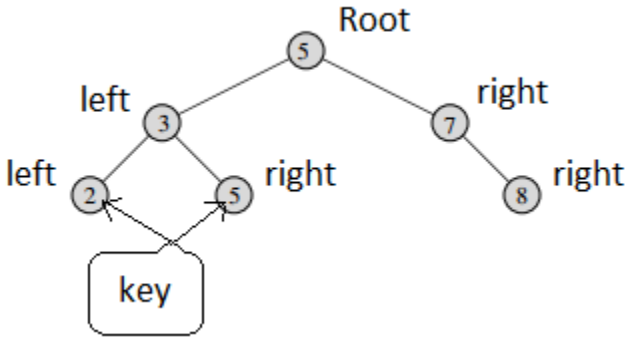
```
struct BST {
        int key;
        BST *left, *right, *parent; // cay con trai, cay con phai, nut cha
} *Root ;
```



11.2 Kiểu dữ liệu:

```
struct BST {
        int key;
        BST *left, *right, *parent; // cay con trai, cay con phai, nut cha
} *Root ;

void INIT_BST()
{
        Root=NULL;

}
```

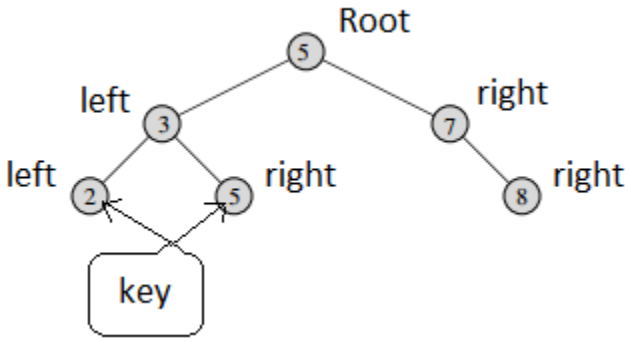
Ví dụ : Tạo cây như hình




```
BST *p, *pl, *pr;
p=(BST*)malloc(sizeof(BST));
Root=p;
p→key=5; p→parent=NULL;
pl=(BST*)malloc(sizeof(BST));
pr=(BST*)malloc(sizeof(BST));
p→left=pl; p→right=pr;
pl→key=3; pl→parent=p;
```

```
p=pl;
pl=(BST*)malloc(sizeof(BST));
pr=(BST*)malloc(sizeof(BST));
p→left=pl; p→right=pr;
pl→key=2; pl→parent=p;
pl→left=NULL; pl→right=NULL;
pr→key=5; pr→parent=p;
pr→left=NULL; pr→right=NULL;
```

Ví dụ : Duyệt các nút bên trái nhất của Root.



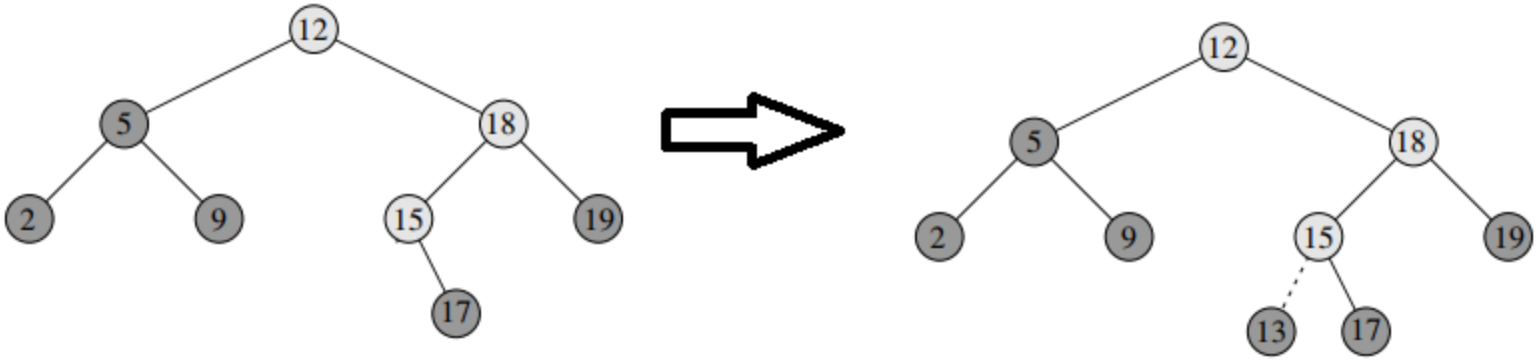

```
BST *p;
p=Root;
while (p!=NULL) {
        printf("%d\n", p→key);
        p=p→left;
}
```

11.3 Thêm một nút vào cây:

```
void TREE_INSERT(BST **T, int v)
{           BST *z, *y, *x;
            1. z=(BST*)malloc(sizeof(BST));
            2. z→key = v; z→left=NULL ; z→right=NULL;
            3. y = NULL;
            4. x = *T;
            5. while (x != NULL){
                        6. y = x;
                        7. if (z→key < x→key) x = x→left;
                        8. else x = x→right;
               } // end while
            9. z→parent = y;
            10. if (y == NULL)
                        11. *T = z;  // Tree T was empty
                else    12. if (z→key < y→key) y→left = z;
                        13. else y→right = z;
}
```

Ví dụ :

Thêm 13 và BST



11.4 Tìm *key* trong cây:

```
BST *TREE_SEARCH(BST *x, int k)
{
        while ((x != NULL) && (k != x→key))
                if (k < x→key)   x = x→left;
                else x = x→right;

        return x;
}
```

11.5 Duyệt cây theo *thứ tự giữa* :

```
void INORDER_TREE_WALK(BST *x)
{
        if (x != NULL) {
                INORDER_TREE_WALK(x→left);
                printf("%d\n", x→key);
                INORDER_TREE_WALK(x→right);
        }
}
```

11.6 Duyệt cây theo *thứ tự trước* :

```
void PREORDER_TREE_WALK(BST *x)
{
        if (x != NULL) {
                printf("%d\n", x→key);
                PREORDER_TREE_WALK(x→left);
                PREORDER_TREE_WALK(x→right);
        }
}
```

11.7 Duyệt cây theo *thứ tự sau* :

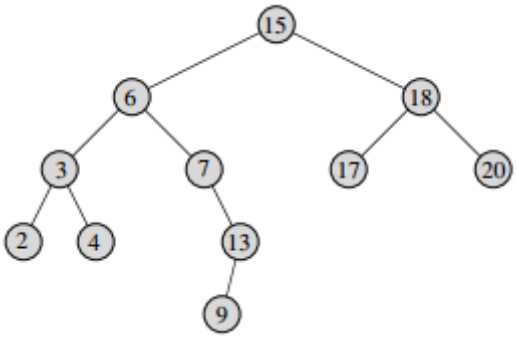
```
void POSTORDER_TREE_WALK(BST *x)
{
        if (x != NULL) {
                POSTORDER_TREE_WALK(x→left);
                POSTORDER_TREE_WALK(x→right);
                printf("%d\n", x→key);
        }
}
```

11.8 Successor của một nút :

Giả sử các key trong BST là khác nhau. Các key trong BST được sắp thứ tự được xác định bởi thủ tục *duyệt theo thứ tự giữa*. Cho một nút x thuộc BST. *Successor của nút x là nút có key bé nhất và lớn hơn key của x*.

Ví dụ :

- Successor của nút với key 15 là nút với key 17.
- Successor của nút với key 13 là nút với key 15.



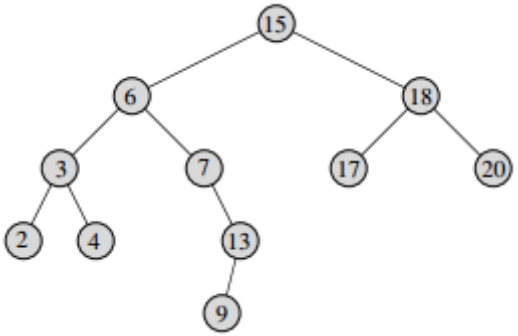
Mệnh đề :

- Nếu cây con phải của nút x khác rỗng thì successor của x là *nút phía trái nhất* trong cây con phải của x,
- Nếu cây con phải của nút x rỗng và x có successor là y thì y là tổ tiên (ancestor) gần nhất của x mà nút con trái của nó cũng là tổ tiên của x. (Để tìm y, đi lên phía trên từ x cho đến khi gặp một nút là nút con bên trái của nút y của nó.)

  Chú ý : x cũng là tổ tiên của x.

Ví dụ :

- Successor của nút với key 15 là nút với key 17.
- Successor của nút với key 13 là nút với key 15.



11.8 Successor của một nút :

Thuật toán :

```
BST *TREE_MINIMUM(BST *x)
{
        while (x→left != NULL) x = x→left;
        return x;
}
BST *TREE_MAXIMUM(BST *x)
{
        while (x→right != NULL) x = x→right;
        return x;
}
```

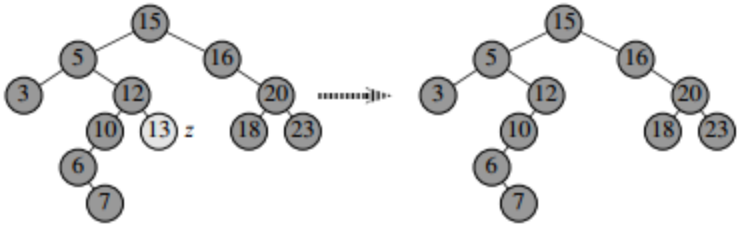
11.8 Successor của một nút :

Thuật toán :

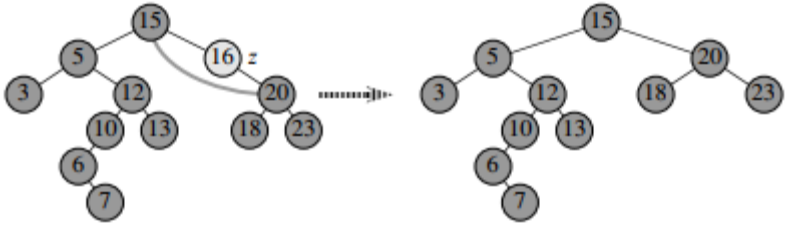
```
BST *TREE_MINIMUM(BST *x)
{
   . . .
}
BST *TREE_SUCCESSOR(BST *x)
{       BST *y;
        if (x→right != NULL) return TREE_MINIMUM(x→right);
        y = x→parent;
        while ((y != NULL) && (x == y→right)){
                x = y;    y = y→parent;
        }
        return y;
}
```

11.9 Xóa một nút : Xóa nút z.

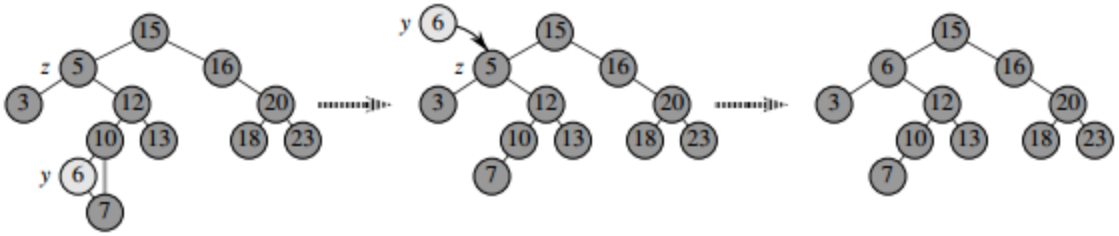
- Trường hợp 1 : Nút z không có con.



- Trường hợp 2 : Nút z có một con.



- Trường hợp 3 : Nút z có hai con. y là successor của z.



11.9 Xóa một nút : Xóa nút z.

```
BST *TREE_DELETE(BST **T, BST *z)
{       BST *x, *y;
        1. if ((z→left== NULL) || (z →right == NULL)) y = z;
        2. else y = TREE_SUCCESSOR(z);
        3. if (y→left != NULL) x = y→left ;
        4. else x = y→right ;
        5. if (x != NULL) x→parent = y→parent;
        6. if (y→parent == NULL) *T = x;
        7. else
                8. if (y == (y→parent)→left) (y→parent)→left = x;
                9. else (y→parent)→right = x;
        10. if (y != z) {z→key = y→key; copy y's satellite data into z}
            return y;
}
```

11.10 Red-Black Tree

11.10.1 Khai báo :

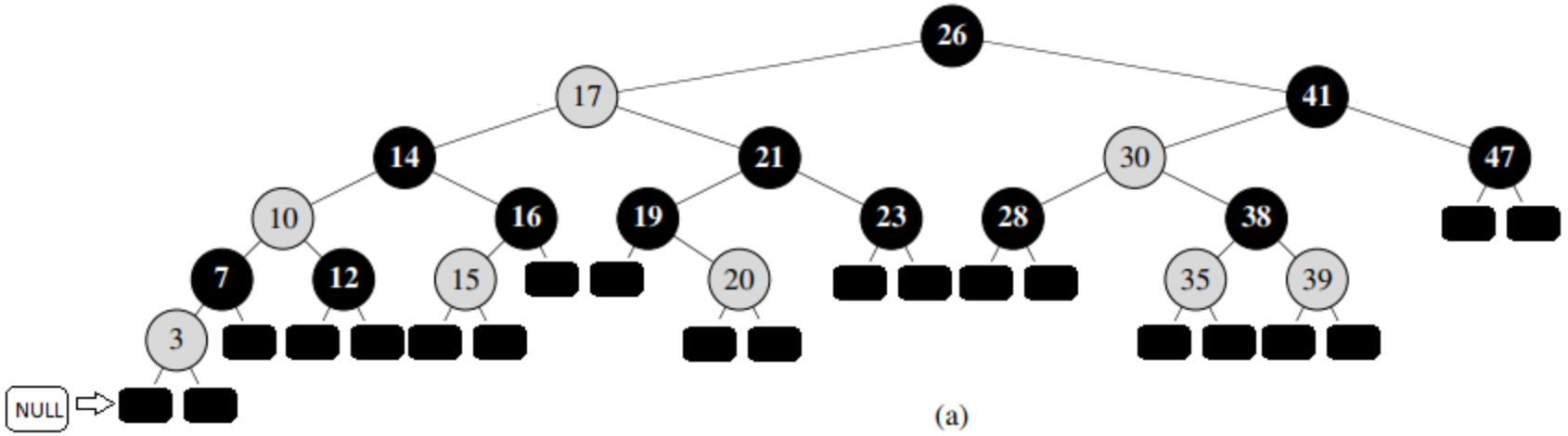
```
typedef enum COLOR { cRED, cBLACK };
struct BST {
        int key;
        BST *left, *right, *parent; // cay con tra, cay con phai, nut cha
        COLOR color ;  // 0 : cRED,   1 : cBLACK
} *Root, NULL_T;

void RB_INIT_BST()
{
        Root=&NULL_T;
        NULL_T.color=cBLACK;
        NULL_T.key=-1;
}
```

11.10.2 Tính chất của Red-Black Tree :

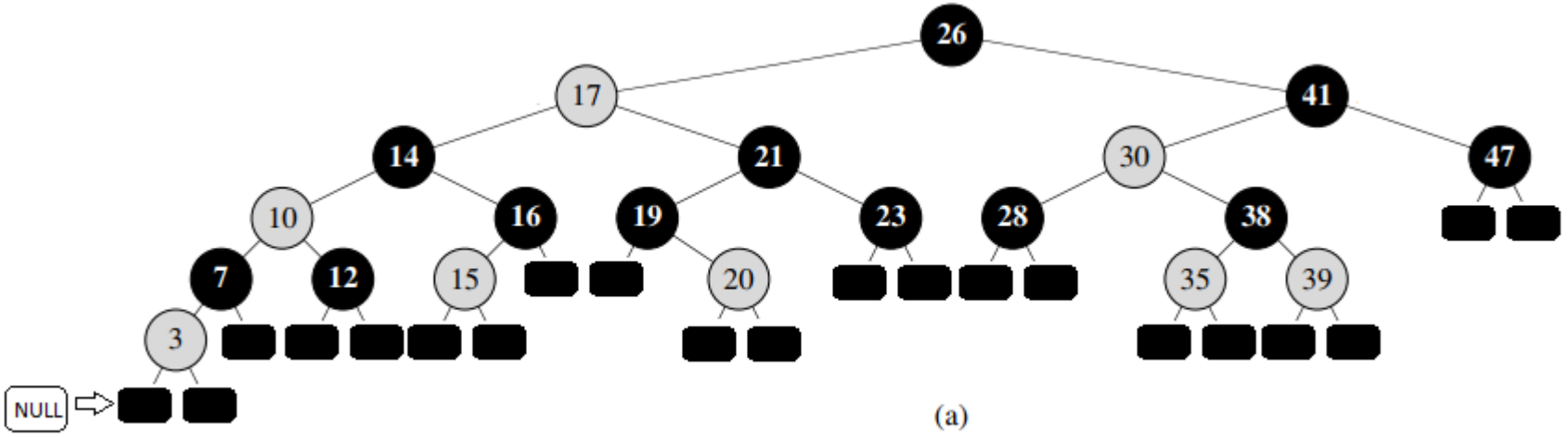
- Con trỏ là nút con bên trái hay nút con bên phải của một nút nếu là NULL thì con trỏ này được gọi là *nút lá*.

- Một cây BST là một Red-Black Tree (RBT) nếu

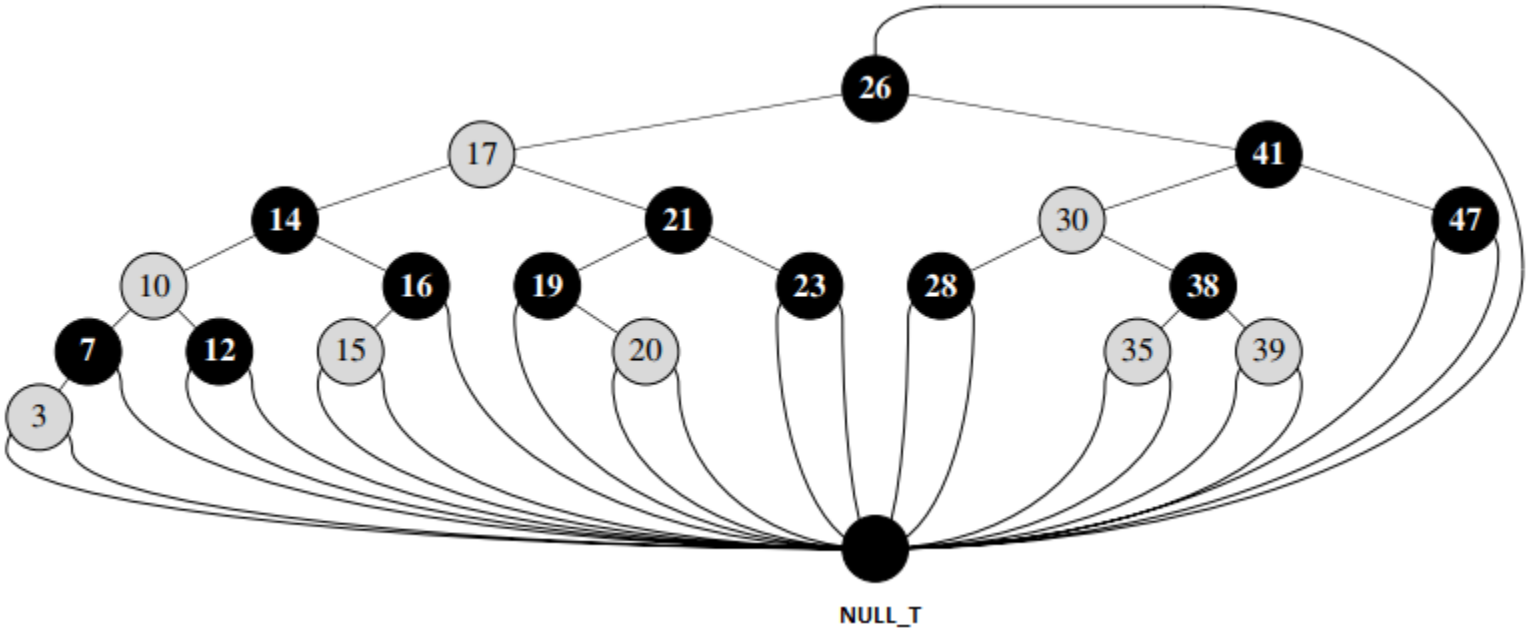
1. Mọi nút có màu Đỏ hay Đen.
2. Nút gốc (Root) có màu đen.
3. Mọi nút lá (NULL) là màu đen.
4. *Nếu một nút là màu đỏ* thì hai con của nó là màu đen.
5. Với mọi nút, tất cả các đường đi (sơ cấp) đi xuống nút lá (NULL) chứa cùng số nút đen.



(a)

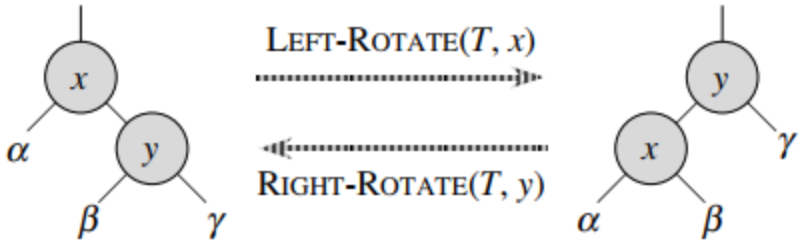

26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
NULL

(a)


26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
NULL_T

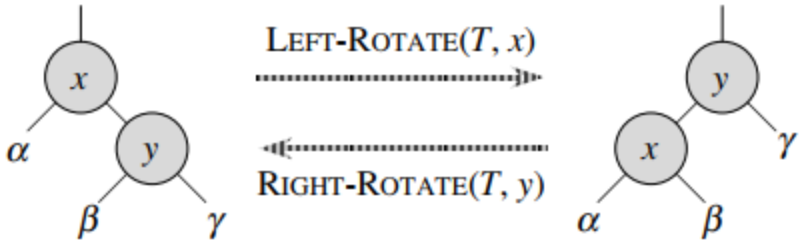
- A red-black tree with n internal nodes has height at most $2 \lg(n + 1)$.
- The basic dynamic-set operations take $O(\lg n)$ time in the worst case.

11.10.3 Rotations :



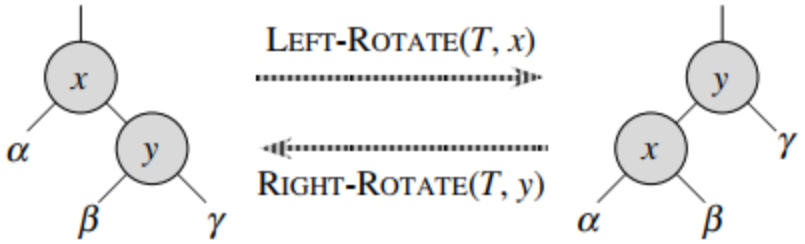

```
void LEFT_ROTATE(BST **T, BST *x)
{       BST *y;
        1. y = x->right;
        2. x->right = y->left;
        3. if (y->left != &NULL_T) (y->left)->parent = x;
        4. y->parent = x->parent;
        5. if (x->parent == &NULL_T) *T = y;
        6. else
                7. if (x == (x->parent)->left) (x->parent)->left = y;
                8. else (x->parent)->right = y;
        9. y->left = x ;  x->parent = y;
}
```

11.10.3 Rotations :




```
void RIGHT_ROTATE(BST **T, BST *x)
{       BST *y;
        y = x->left;
        x->left = y->right;
        if (y->right != &NULL_T) (y->right)->parent = x;
        y->parent = x->parent;
        if (x->parent == &NULL_T) *T = y;
        else
                if (x == (x->parent)->right) (x->parent)->right = y;
                else (x->parent)->left = y;
        y->right = x ;     x->parent = y;
}
```

11.10.3 Rotations (new):




```
void RIGHT_ROTATE(BST **T, BST *y)
{       BST *x;
        x = y->left;
        y->left = x->right;
        if (x->right != &NULL_T) (x->right)->parent = y;
        x->parent = y->parent;
        if (y->parent == &NULL_T) *T = x;
        else
                if (y == (y->parent)->right) (y->parent)->right = x;
                else (y->parent)->left = x;
        x->right = y ;     y->parent = x;
}
```

11.10.4 Insertion :

```
void RB_TREE_INSERT(BST **T, int v)
{       BST *z, *y, *x;
        z=(BST*)malloc(sizeof(BST)); z->key = v;
        z->left=&NULL_T ;  z->right=&NULL_T;  z->color=cRED;
        y = &NULL_T;   x = *T;
        while (x != &NULL_T) {
                y = x;
                if (z->key < x->key) x = x->left;
                else x = x->right; }
        z->parent = y;
        if (y == &NULL_T) *T = z;  // Tree T was empty
        else    if (z->key < y->key) y->left = z;
                else y->right = z;
        RB_INSERT_FIXUP(T, z);
}
```

11.10.4 Insertion :

Case 1: z's uncle y is red

Case 2: z's uncle y is black and z is a right child

Case 3: z's uncle y is black and z is a left child

```
void RB_INSERT_FIXUP(BST **T ,BST *z)
{BST *y;
 while ((z->parent)->color == cRED){
      if (z->parent == ((z->parent)->parent)->left) {
          y = ((z->parent)->parent)->right;
          if (y->color== cRED) {
Case 1      (z->parent)->color = cBLACK ; y->color = cBLACK ;
            ((z->parent)->parent)->color = cRED ; z = (z->parent)->parent; }
           else {   if (z == (z->parent)->right) {
                        z = z->parent; LEFT_ROTATE(T, z);        // Case 2
                    }
                   (z->parent)->color = cBLACK;                  // Case 3
                   ((z->parent)->parent)->color = cRED;          // Case 3
                   RIGHT_ROTATE(T, (z->parent)->parent);     // Case 3
                }
    }
```

```
void RB_INSERT_FIXUP(BST **T ,BST *z)
{ while ((z->parent)->color == cRED){
      if (z->parent == ((z->parent)->parent)->left)  { . . .}
      else{  y = ((z->parent)->parent)->left;
             if (y->color  == cRED) {
Case 1       (z->parent)->color = cBLACK ;   y->color = cBLACK ;
             ((z->parent)->parent)->color = cRED ; z = (z->parent)->parent ; }
             else {  if (z == (z->parent)->left){
                           z = z->parent; RIGHT_ROTATE(T, z); }  // Case 2
                     (z->parent)->color = cBLACK;                 // Case 3
                     ((z->parent)->parent)->color = cRED;         // Case 3
                     LEFT_ROTATE(T, (z->parent)->parent);         // Case 3
                  }
           }
        }// end while
        (*T)->color = cBLACK;
}
```

11.10.5 Xóa một nút : Xóa nút z.

```
BST *RB_TREE_DELETE(BST **T, BST *z)
{       BST *x, *y;
        if ((z->left== &NULL_T) || (z->right == &NULL_T)) y = z;
        else y = TREE_SUCCESSOR(z);
        if (y->left != &NULL_T) x = y->left ;
        else x = y->right ;
        x->parent = y->parent;
        if (y->parent == &NULL_T) *T = x;
        else
                if (y == (y->parent)->left) (y->parent)->left = x;
                else (y->parent)->right = x;
        if (y != z) {z->key = y->key; copy y's satellite data into z }
        if (y->color== cBLACK) RB_DELETE_FIXUP(T, x);
        return y;
}
```