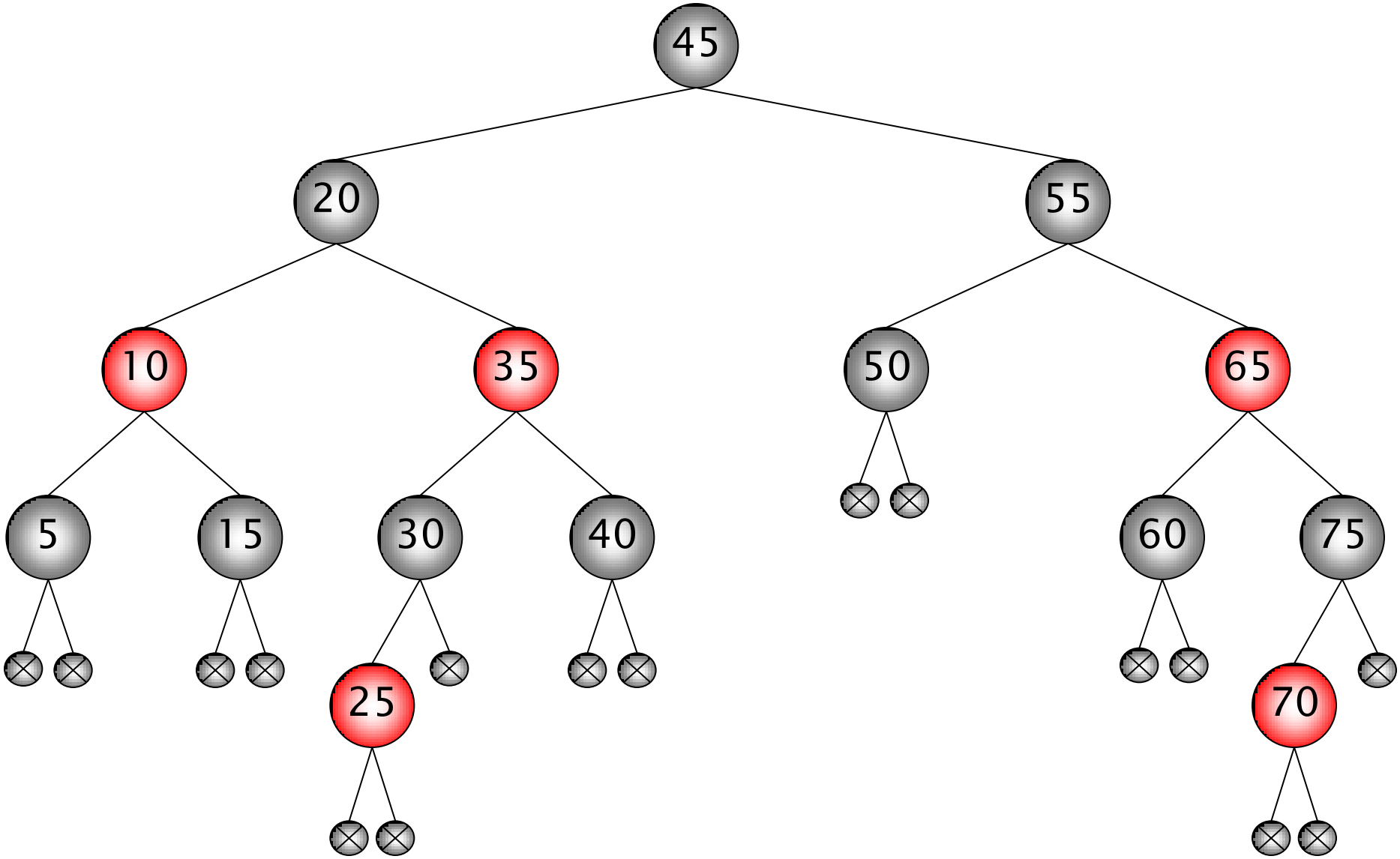
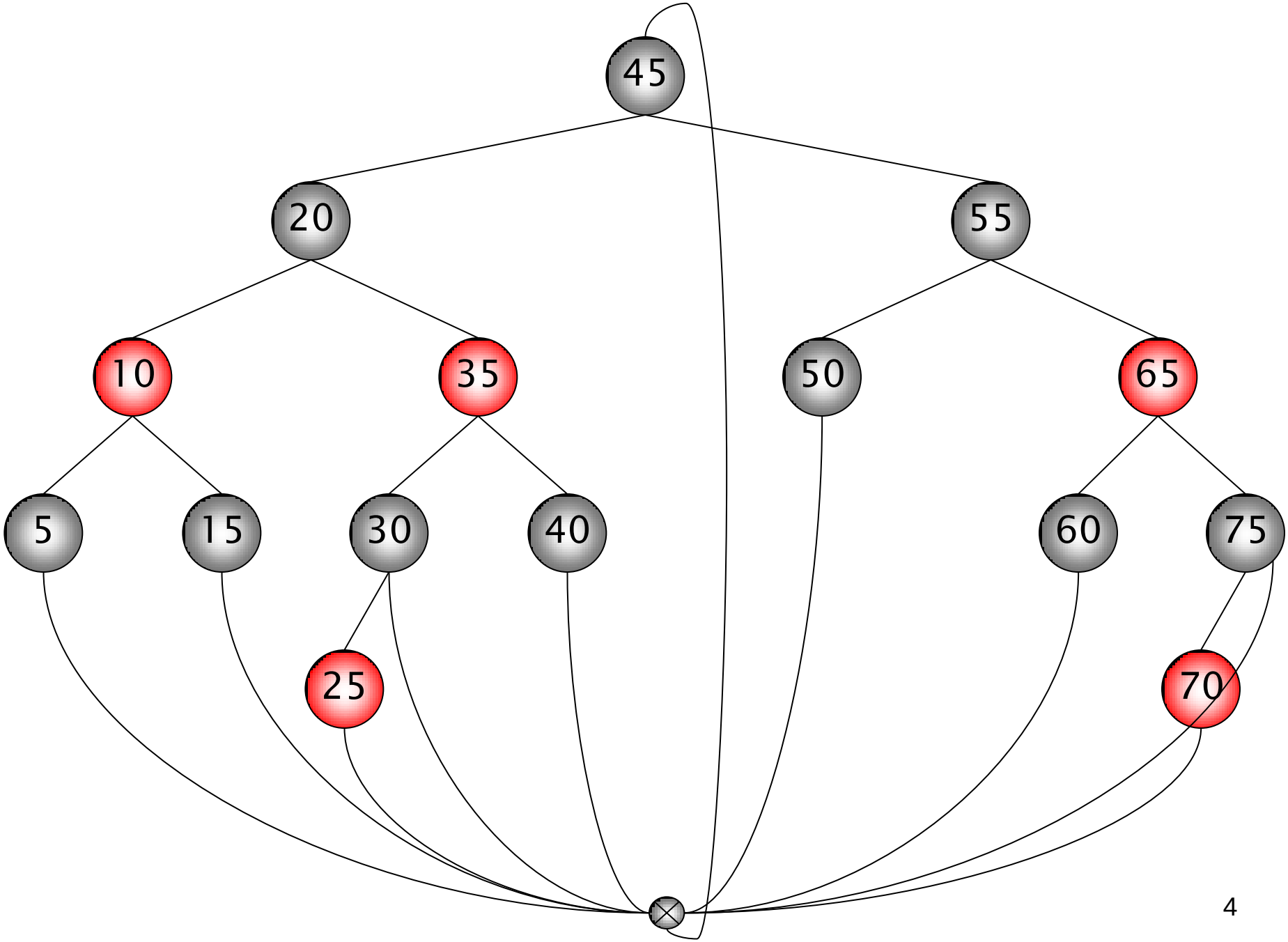
Cây Đỏ–Đen

Cây Đỏ–Đen

- Cây Đỏ-Đen thuộc nhóm cấu trúc cây tìm kiếm, duy trì tính "cân bằng" để đảm bảo cho các thao tác:
  - Thêm (*insertion*)
  - Hủy (*detetion*)
  - Tìm kiếm (*search*) có chi phí lớn nhất là O(log*n*).
- Với cây AVL, chiều duyệt xuống dùng để tìm và thêm/hủy phần tử, chiều duyệt lên để cập nhật tính cân bằng, sử dụng đệ qui.
- Với cây Đỏ-Đen, quá trình lặp thay cho đệ qui khiến việc thực thi đơn giản và hiệu quả hơn.


45
20
55
10
35
50
65
5
15
30
40
60
75
25
70


45
20
55
10
35
50
65
5
15
30
40
60
75
25
70

Định nghĩa: Cây Đỏ-Đen là cây nhị phân tìm kiếm thỏa các tiêu chuẩn sau:

- Mỗi nút hoặc là nút đỏ, hoặc là nút đen.
- Nút gốc luôn luôn là nút đen.
- Nếu một nút là đỏ thì nút con của nó phải là đen.
- Mỗi đường đi từ một nút bất kỳ đến các nút lá của cây đều có cùng số lượng các nút *đen* (chiều cao đen – *black height*).

Nhận xét:

- Cây Đỏ–Đen với n nút trong có chiều cao tối đa là $2\log_2(n + 1)$.
- Các nút lá **`NIL`** (nút ngoài) luôn là nút đen.
- Một cạnh dẫn đến một nút đen gọi là cạnh đen (*black edge*).
- Nếu không tồn tại nút cha hay nút con của một nút thì những con trỏ tương ứng sẽ trỏ về **`NIL`**.
  - Để thuận tiện, một lính canh (**`Sent`**) đại diện cho tất cả các nút **`NIL`**.

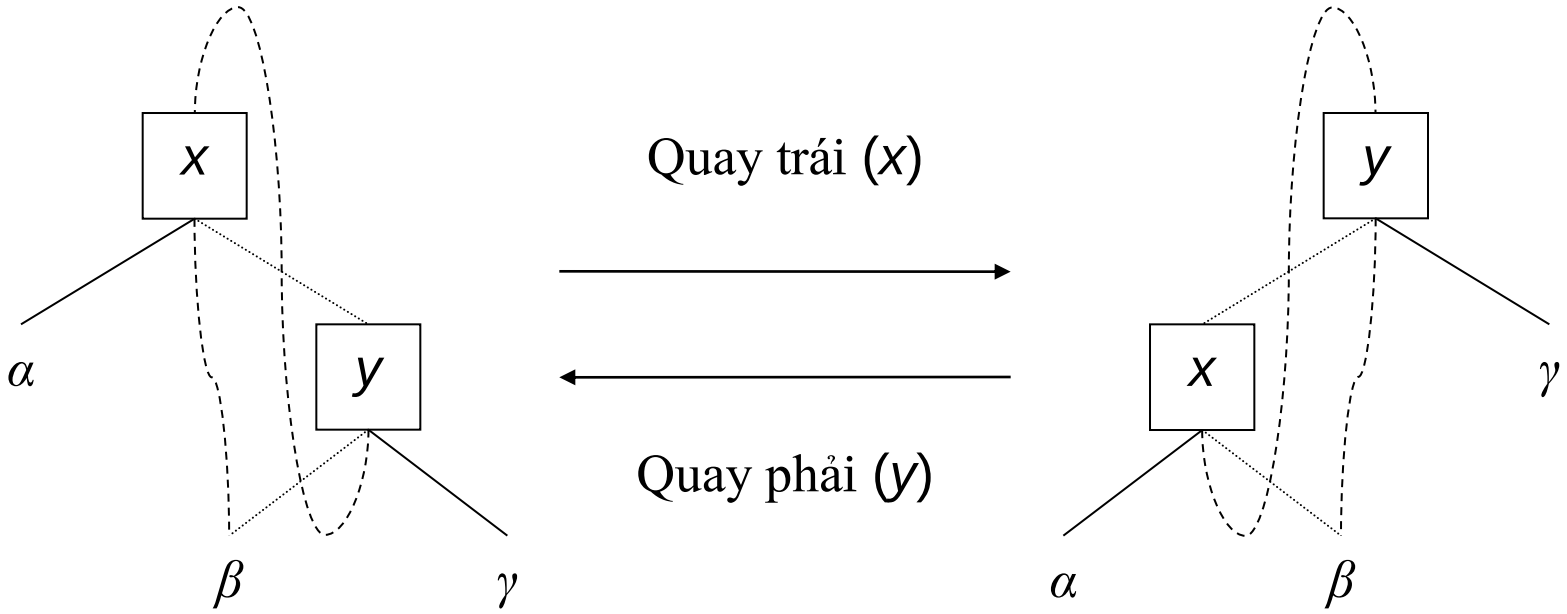
```
typedef struct  Node * ref;
struct  Node {
  int key;
  int color;
  ref parent;
  ref left;
  ref right;
}

ref getNode(int key, int color, ref nil) {
  p = new Node;
  p->key   = key;
  p->color = color;
  p->left  = p->right = p->parent = nil;
  return   p;
}
```

Trạng thái ban đầu

```
ref nil, root;
…
nil = new Node;
nil->color = BLACK;
nil->left =
nil->right =
nil->parent = nil;
…
root = nil;
```

Thao tác quay



```
void  leftRotate(ref &root, ref x) {
  y = x->right;
  x->right = y->left;

  if (y->left != nil)
    y->left->parent = x;
  y->parent = x->parent;

  if (x->parent == nil) root = y;
  else
    if (x == x->parent->left)
      x->parent->left = y;
    else
      x->parent->right = y;
  y->left = x;
  x->parent = y;
}
```

Thao tác chèn (Insertion)

Phần tử chèn vào cây Đỏ–Đen luôn luôn có màu đỏ.

- Thêm nút đen sẽ vi phạm tiêu chuẩn "cân bằng".

Giải thuật:

Bước 1: Chèn phần tử (tương tự cây nhị phân tìm kiếm).

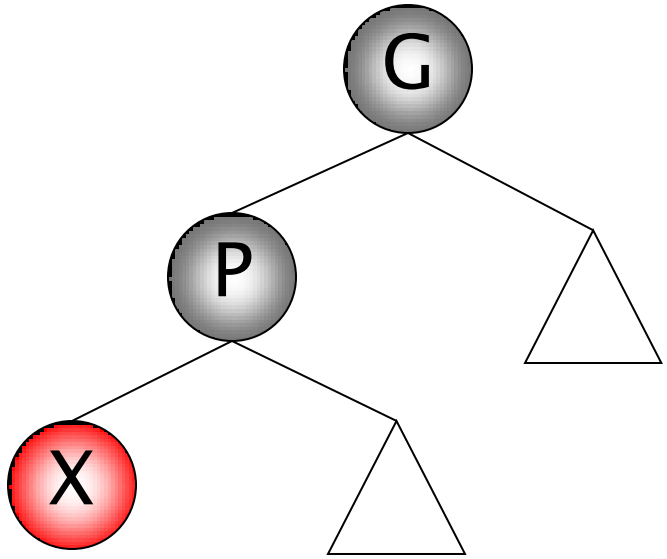
Bước 2: Cập nhật các trường thông tin cho nút mới.

Bước 3: Để duy trì tính "cân bằng", nếu cần, thao tác tô màu nút và phép quay sẽ được thực hiện.

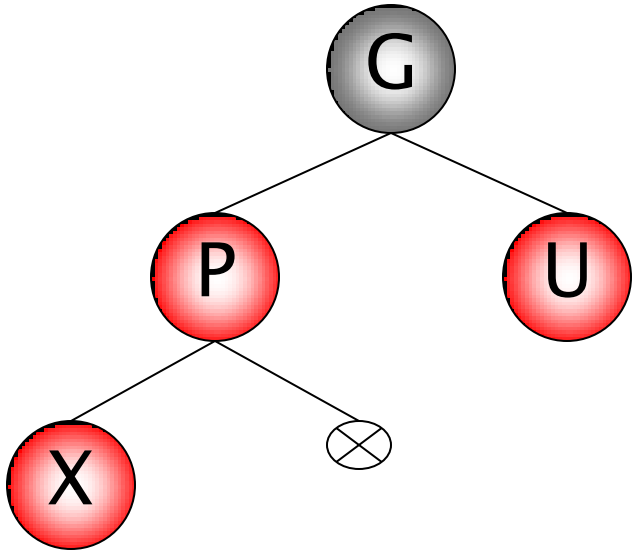
```
void  RBT_Insertion(ref & root, int key) {
  x = getNode(key, RED, nil);
  BST_Insert(root, x);
  Insertion_FixUp(root, x);
}
```

```
void  BST_Insert(ref &root, ref x) {
  y = nil;
  z = root;
  while (z != nil) {
    y = z;
    if (x->key < z->key)  z = z->left;
    else                  z = z->right;
  }
  x->parent = y;
  if (y == nil)
    root = x;
  else
    if (x->key < y->key)  y->left = x;
    else                  y->right = x;
}
```

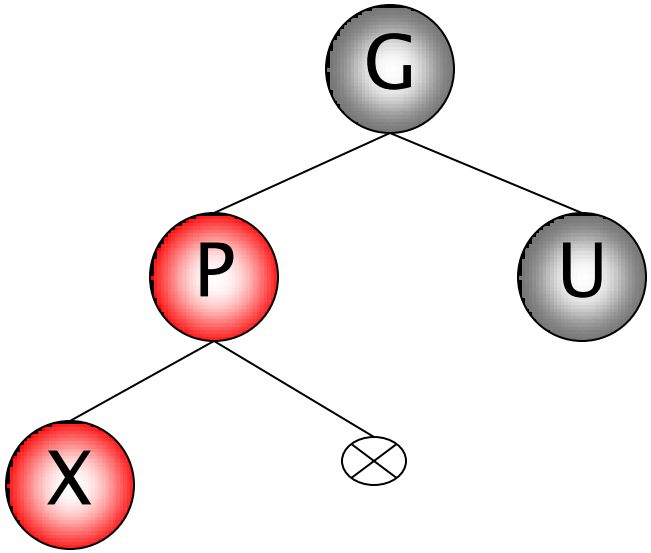
- Gọi:
  - *x*: con trỏ, trỏ đến nút vừa chèn vào.
  - *p*: con trỏ, trỏ đến nút cha của nút trỏ bởi *x*.
  - *u*: con trỏ, trỏ đến nút anh em của nút trỏ bởi *p*.
  - *g*: con trỏ, trỏ đến nút cha của nút trỏ bởi *p*.
- Sau khi chèn, xảy ra 1 trong các trường hợp sau:
  - Nếu *p* đen: Dừng (1).
  - Nếu *p* đỏ:
    - Nếu *u* đỏ: đảo màu 3 nút *p*, *u* và *g* (2).
    - Nếu *u* là đen:
      - *x* là cháu ngoại của *g* (3): đảo màu cha và ông, thực hiện phép quay tại ông.
      - *x* là cháu nội của *g* (4): quay tại cha $\rightarrow$ (3).



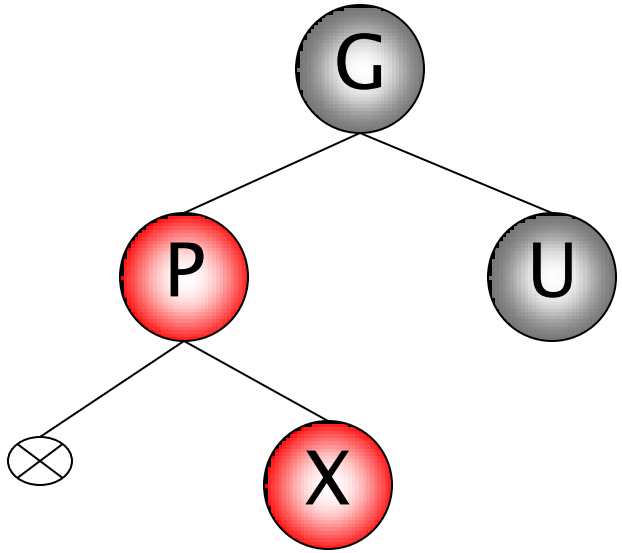
Trường hợp 1



Trường hợp 2

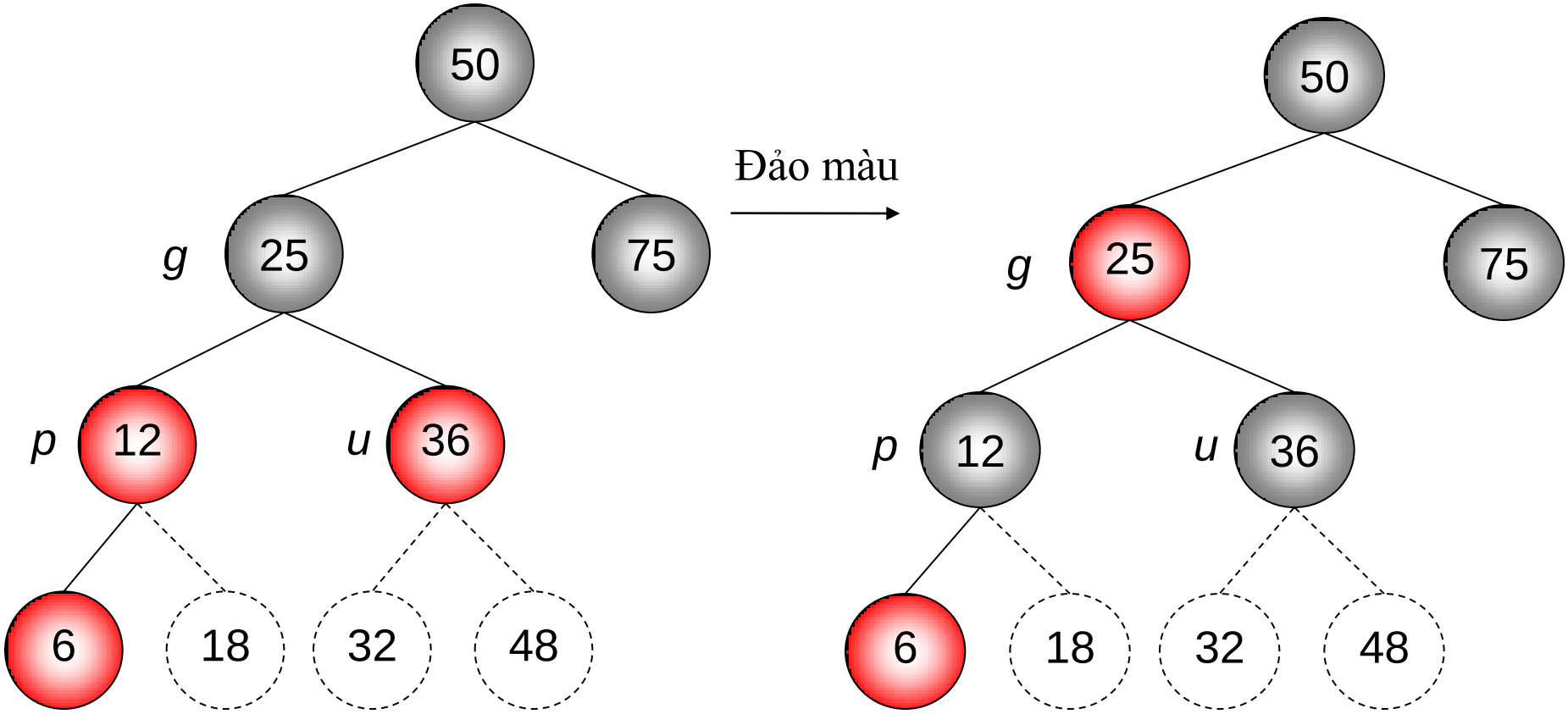


Trường hợp 3

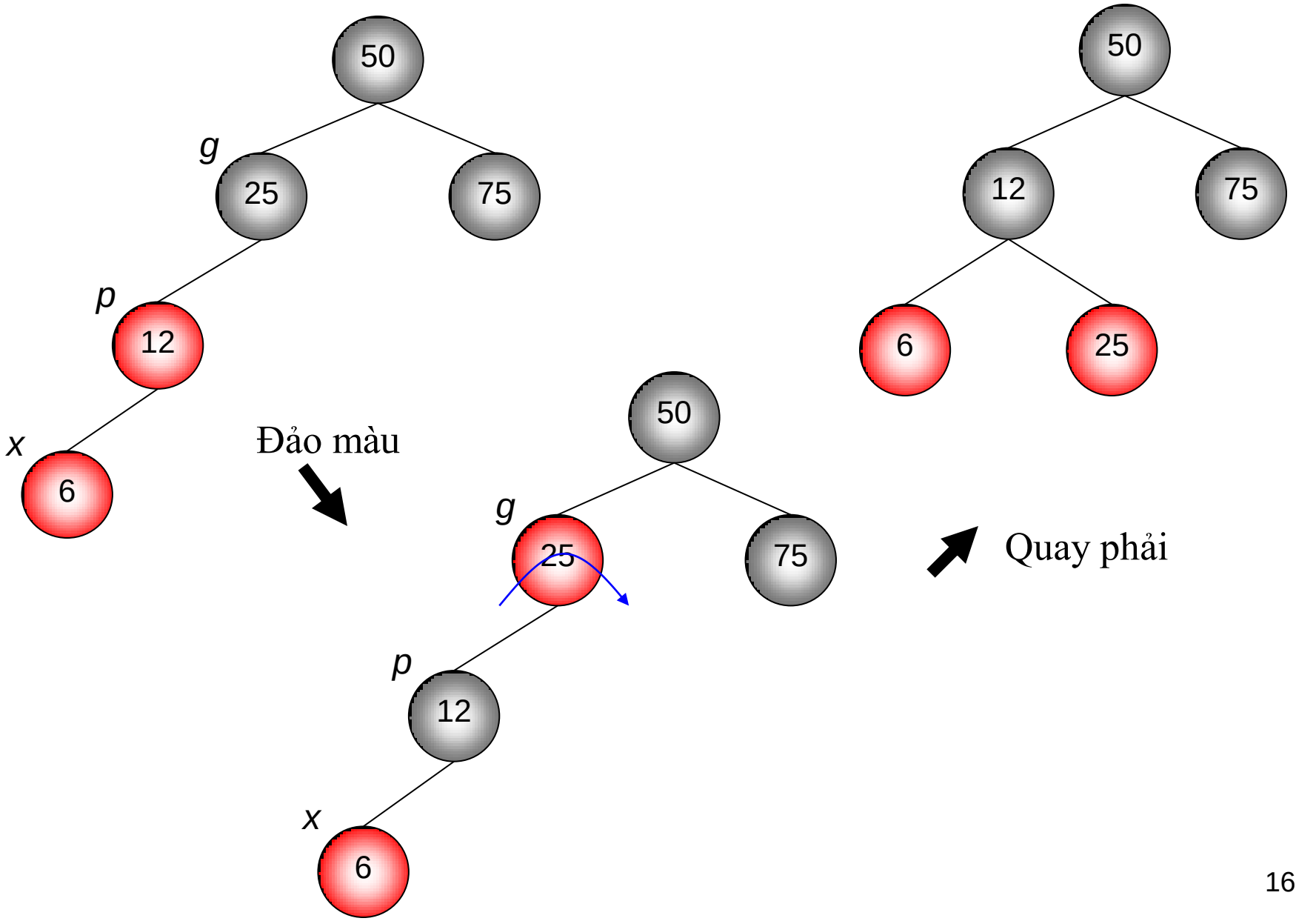


Trường hợp 4

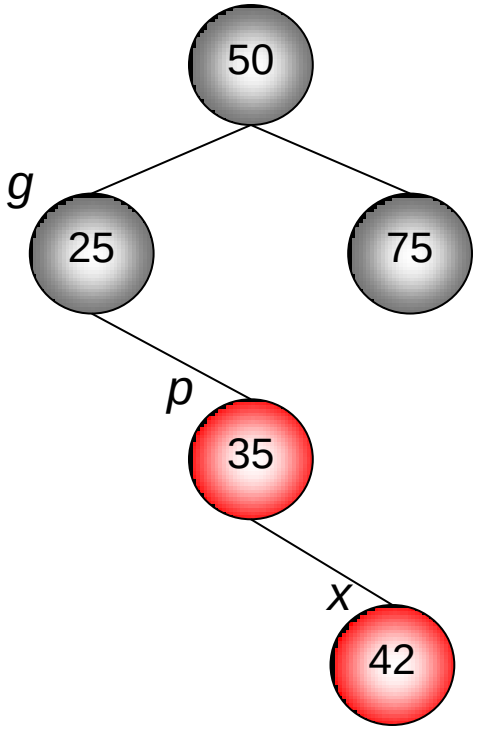
Trường hợp 2



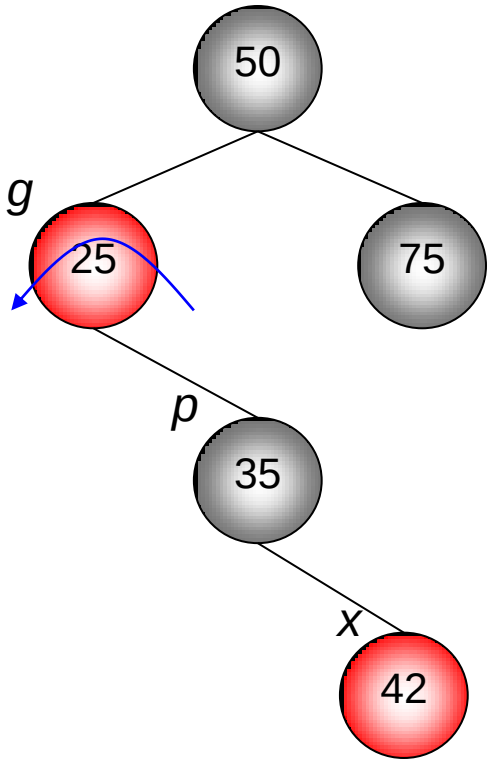
Trường hợp 3



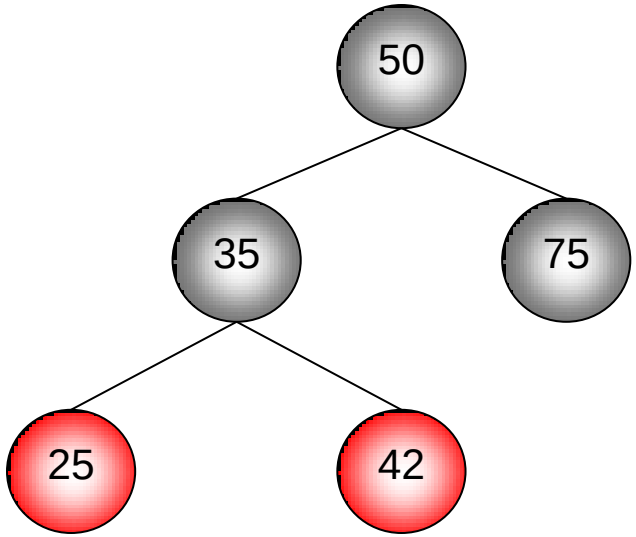
Trường hợp 3



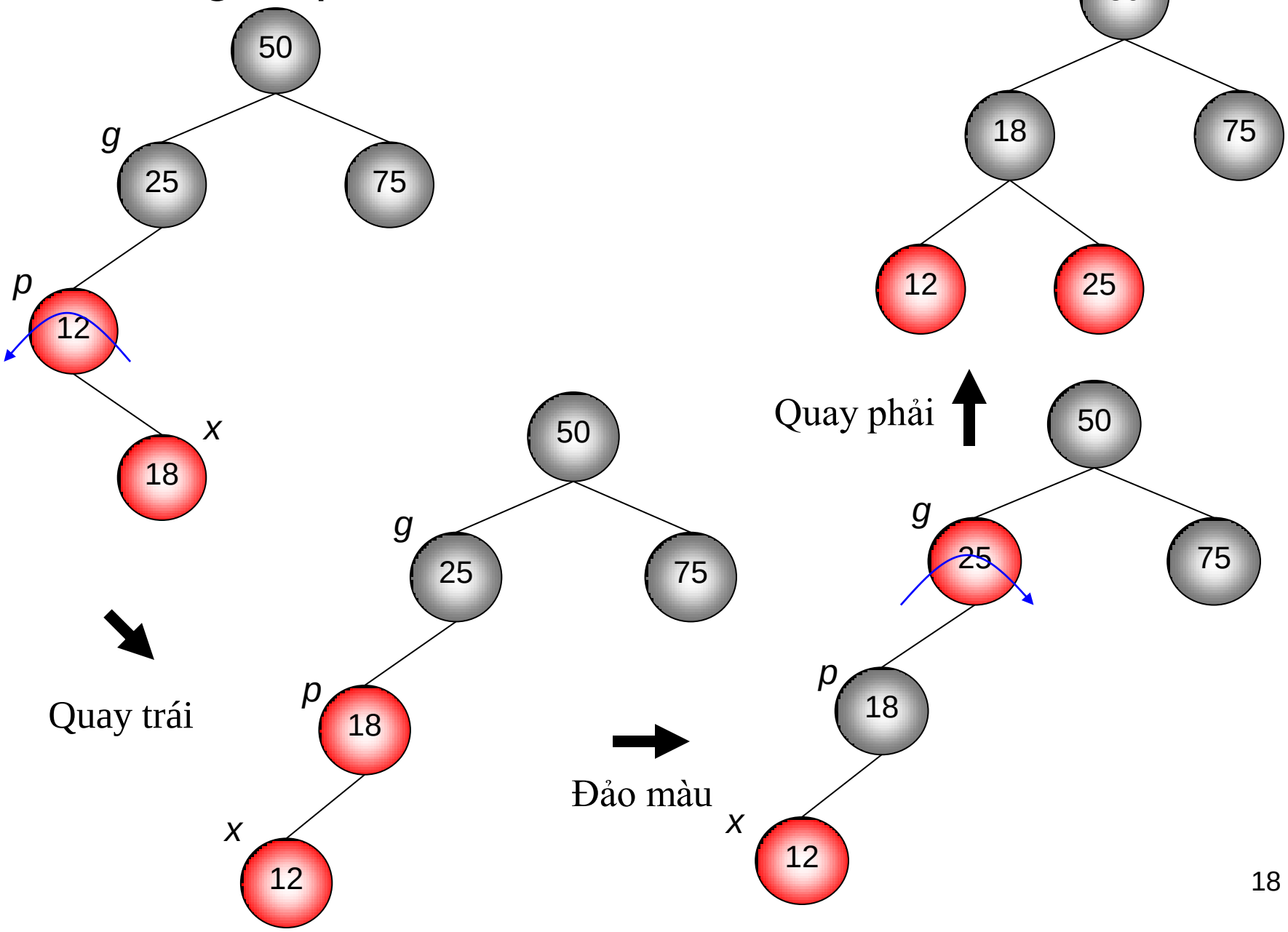
Đảo màu



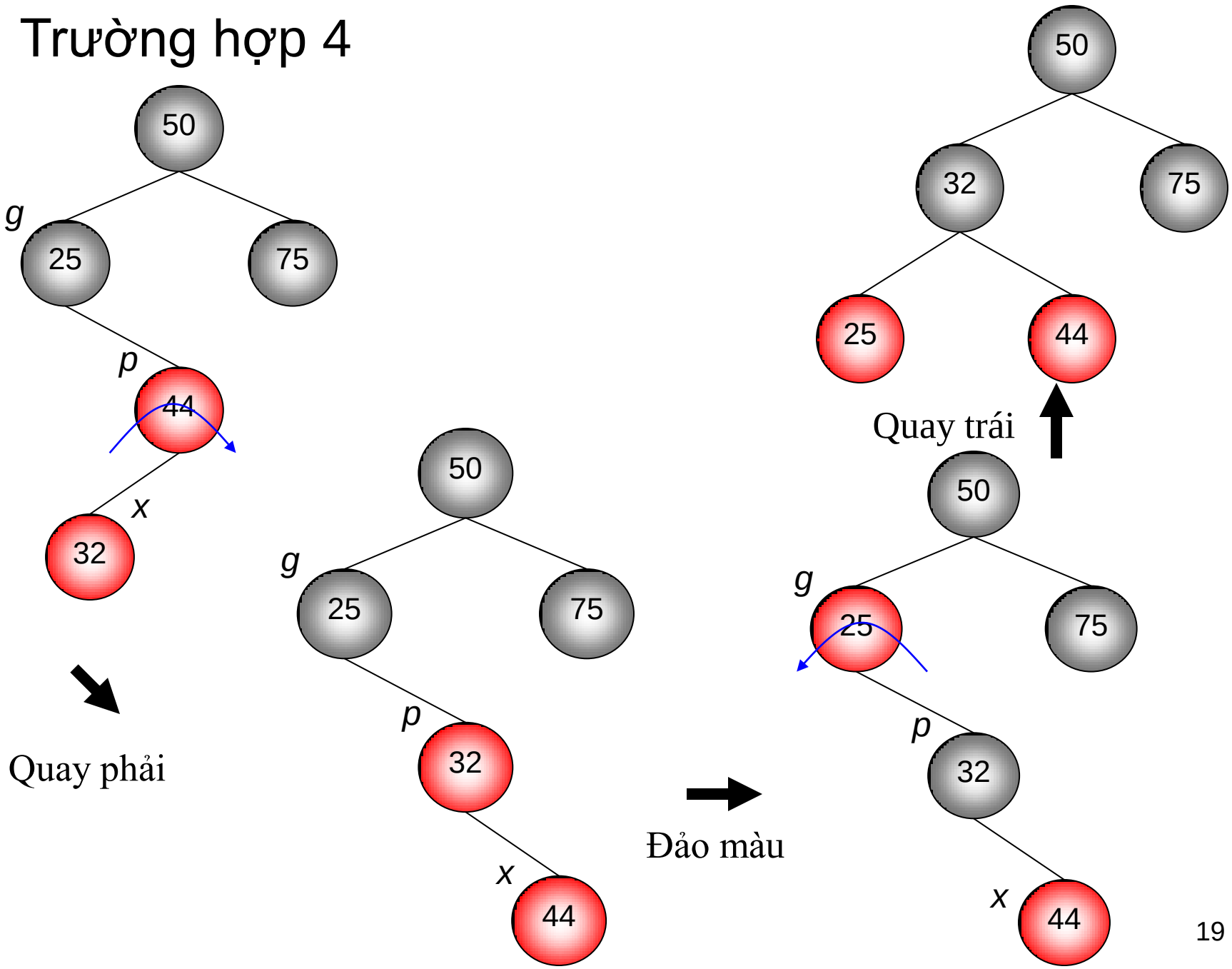
Quay trái



Trường hợp 4


50
g
25
75
p
12
x
18
Quay trái
50
g
25
75
p
18
x
12
Đảo màu
50
18
75
12
25
Quay phải
50
g
25
75
p
18
x
12

Trường hợp 4


50
g
25
75
p
44
x
32
Quay phải
50
g
25
75
p
32
x
44
Đảo màu
50
32
75
25
44
Quay trái
50
g
25
75
p
32
x
44

```
void  Insertion_FixUp(ref &root, ref x) {
  while (x->parent->color == RED)
    if (x->parent == x->parent->parent->left)
      ins_leftAdjust(root, x);
    else
      ins_rightAdjust(root, x);
  root->color = BLACK;
}
```

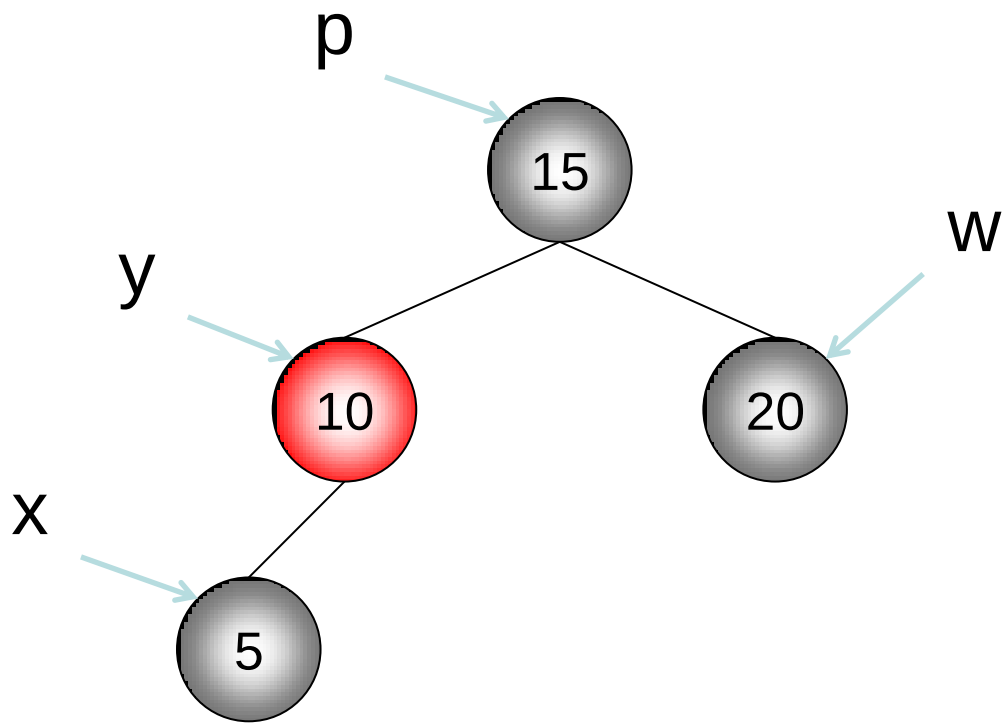
```
void   ins_leftAdjust(ref &root, ref &x) {
  u = x->parent->parent->right;
  if (u->color == RED) {
    x->parent->color   = BLACK;
    u->color           = BLACK;
    x->parent->parent->color = RED;
    x = x->parent->parent;
  }
  else {
    if (x == x->parent->right) {
      x = x->parent;
      leftRotate(root, x);
    }
    x->parent->color = BLACK;
    x->parent->parent->color = RED;
    rightRotate(root, x->parent->parent);
  }
}
```

Thao tác hủy

- Diễn ra theo 2 giai đoạn:
  - Hủy nút: Tương tự cây nhị phân tìm kiếm
    - ✓ Nút lá
    - ✓ Nút có một cây con
    - ✓ Nút có đầy đủ hai cây con: tìm nút thay thế
  - Cân bằng lại cây
    - ✓ Chỉ quan tâm đến nút bị xóa thật sự

- Gọi:
  - *y*: con trỏ, trỏ đến nút bị xóa thật sự
  - *x*: con trỏ, trỏ đến nút con của nút trỏ bởi *y*

    $\rightarrow$ Sẽ thay thế nút trỏ bởi *y*
  - *w*: con trỏ, trỏ đến nút anh em của nút trỏ bởi *y*
  - *p*: con trỏ, trỏ đến nút cha của nút trỏ bởi *y*



- Sau khi xóa:
  - Nếu *y* là nút đỏ: dừng
  - Nếu *y* là nút đen:
    - Mọi con đường đi qua *y* sẽ giảm chiều cao đen
    - (thậm chí) Nếu *p* và *x* cùng là nút đỏ

  $\rightarrow$ Mất cân bằng

```
void RBT_Deletion(ref &root, int k) {
  z = searchTree(root, k);
  if (z == nil) return;
  y = ((z->left == nil) || (z->right == nil)) ?
                              z : TreeSuccessor(root, z);

  x = (y->left == nil) ? y->right : y->left;
  x->parent = y->parent;
  if (y->parent == nil)          root = x;
  else
    if (y == y->parent->left)  y->parent->left = x;
    else                       y->parent->right = x;

  if (y != z)                  z->key = y->key;
  if (y->color == BLACK)       Deletion_FixUp(root, x);
  delete y;
}
```

- *Dấu hiệu đen* (*black token*)
  - Gán cho nút trỏ bởi *x* (con của nút bị xoá thật sự)
  - *Dấu hiệu đen* đi ngược lên cây cho đến khi tiêu chuẩn về *chiều cao đen* được giải quyết
  - Nếu nút chứa *dấu hiệu đen* là:
    - Nút đen $\rightarrow$ nút đen kép (*doubly black node*)
    - Nút đỏ $\rightarrow$ nút đỏ-đen (*red-black node*)
- *Dấu hiệu đen* chỉ là khái niệm trừu tượng

Trường hợp 1

Nút chứa *dấu hiệu đen* là:

- Nút đỏ (nút đỏ–đen), hoặc
- Nút gốc

Xử lý:

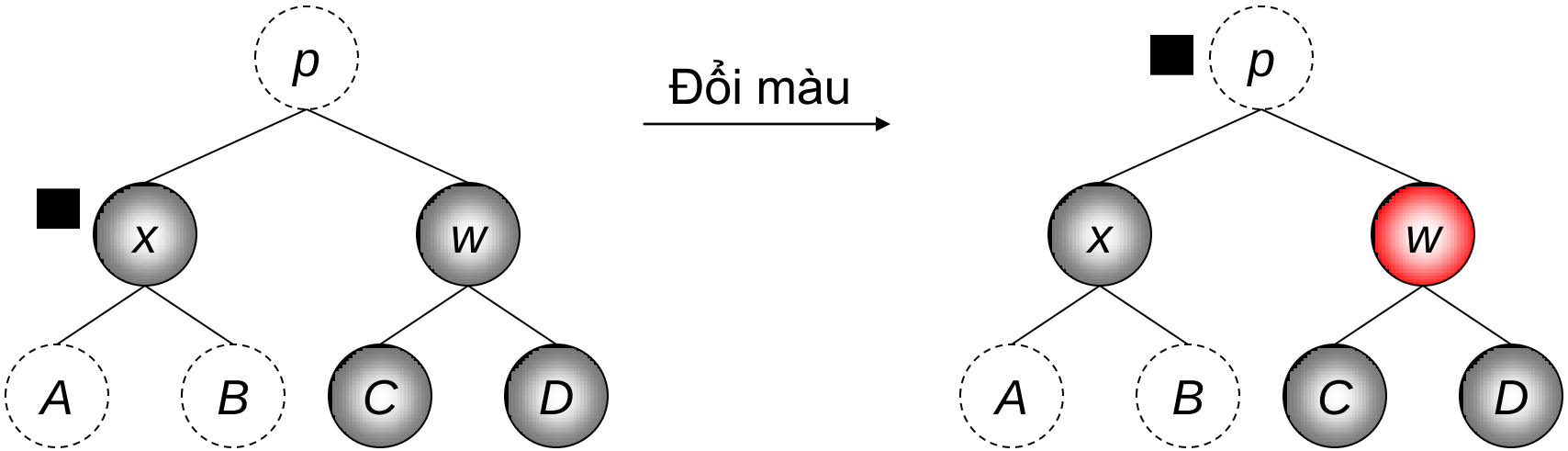
- Tô màu nút đỏ-đen thành đen
- Loại bỏ *dấu hiệu đen* và kết thúc

Trường hợp 2

- Nút đang xét (*x*) là nút đen kép
- Nút anh em *w* màu đen
- Hai nút con của nút anh em *w* (nút cháu) màu đen

Xử lý:

- Đổi màu nút anh em *w* sang đỏ
- Di chuyển *dấu hiệu đen* lên một cấp

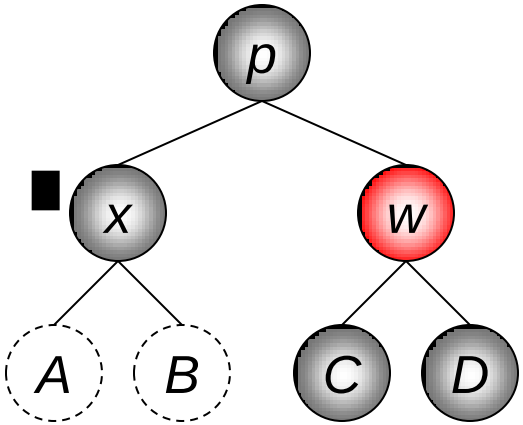

p
x
w
A
B
C
D
Đổi màu
p
x
w
A
B
C
D

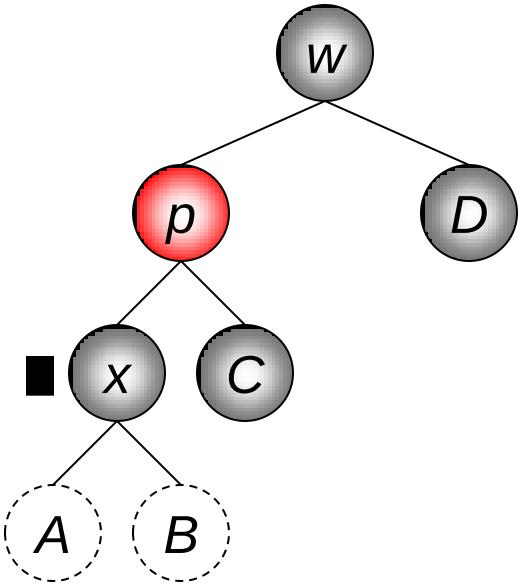
Trường hợp 3

- Nút đang xét (*x*) là nút đen kép
- Nút anh em *w* màu đỏ

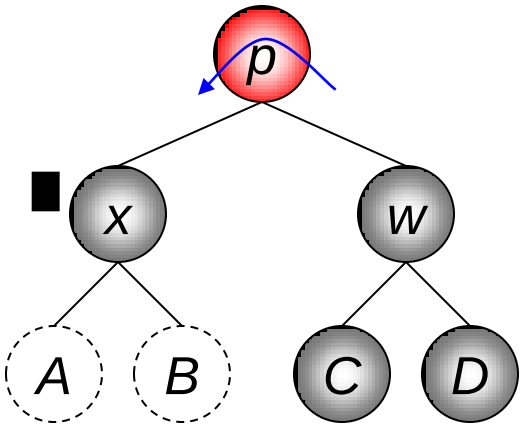
Xử lý:

- Đảo màu nút cha *p* và nút anh em *w*
- Thực hiện phép quay tại cha
- *Dấu hiệu đen* vẫn thuộc nút *x* ban đầu





Đổi màu



Quay trái

Trường hợp 4

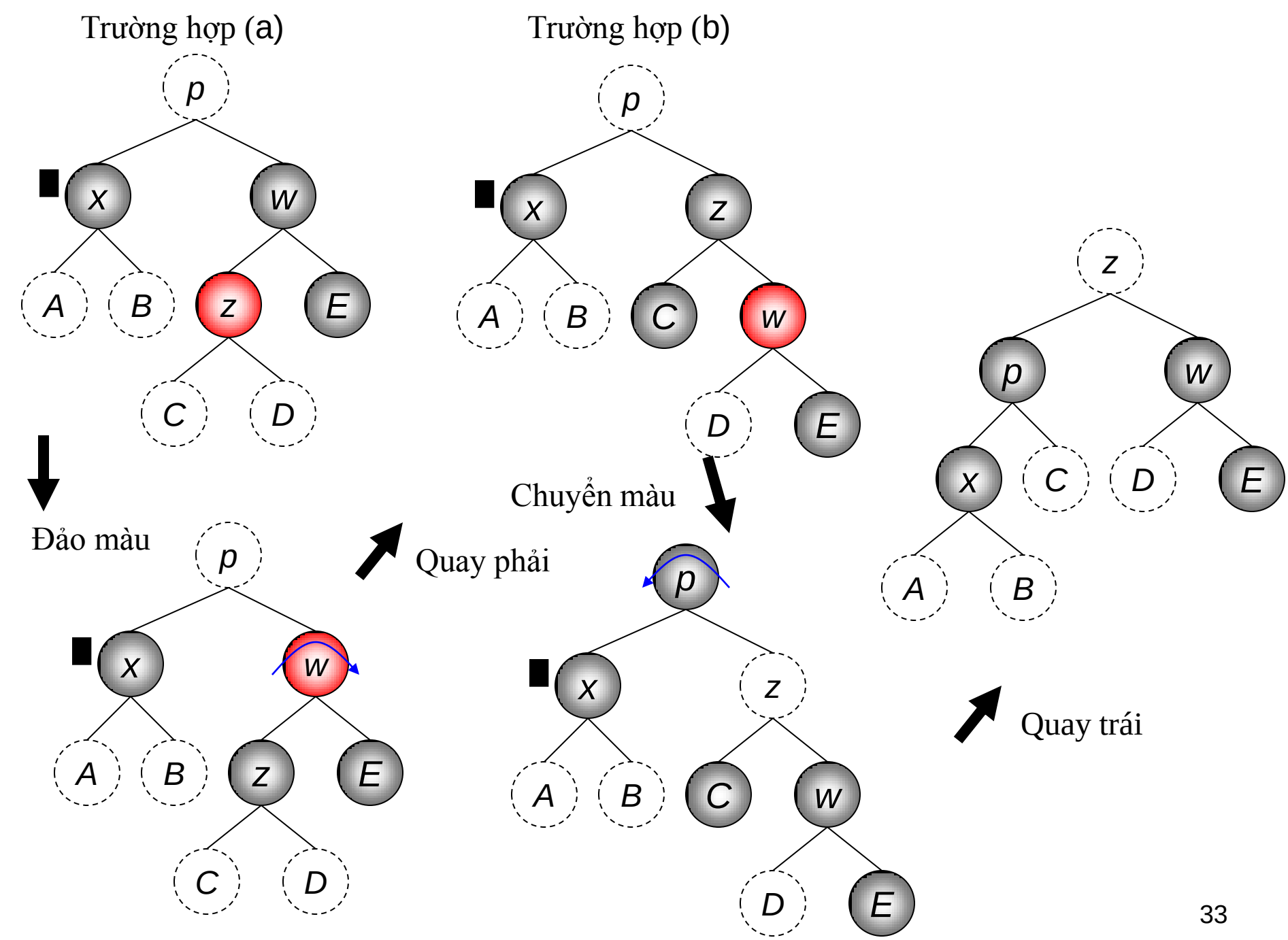
- Nút đang xét (*x*) là nút đen kép
- Nút anh em *w* là đen
- Nút anh em có tối thiểu một nút con là đỏ

Xử lý: Xét 2 nút cháu của cha của nút đen kép

a) Nút cháu ngoại là đen
   - Đảo màu cháu nội và nút anh em (*w*)
   - Quay tại nút anh em (*w*)

   → Chuyển sang trường hợp b

b) Nút cháu ngoại là đỏ
   - Nút anh em *w* (sẽ là gốc cục bộ) nhận màu nút cha
   - Nút ông và nút cháu ngoại nhận màu đen
   - Quay tại nút ông


Trường hợp (a)
Trường hợp (b)
p
x
w
A
B
z
E
C
D
Đảo màu
Quay phải
Chuyển màu
Quay trái

```
void   Deletion_FixUp(ref root, ref x) {
  while ((x->color == BLACK) && (x != root)) {
    if (x == x->parent->left)
      del_leftAdjust(root, x);
    else
      del_rightAdjust(root, x);
  }
  x->color = BLACK;
}

void   del_leftAdjust(ref & root, ref & x) {
  w = x->parent->right;
  if (w->color == RED) {
    w->color          = BLACK;
    x->parent->color  = RED;
    leftRotate(root, x->parent);
    w = x->parent->right;
  }
```

```
  if ((w->right->color == BLACK) &&
                              (w->left->color == BLACK)) {
    w->color = RED;
    x = x->parent;
  }
  else {
    if (w->right->color == BLACK) {
      w->left->color  = BLACK;
      w->color        = RED;
      rightRotate(root, w);
      w = x->parent->right;
    }
    w->color          = x->parent->color;
    x->parent->color  = BLACK;
    w->right->color   = BLACK;
    leftRotate(root, x->parent);
    x = root;
  }
}
```

B – cây

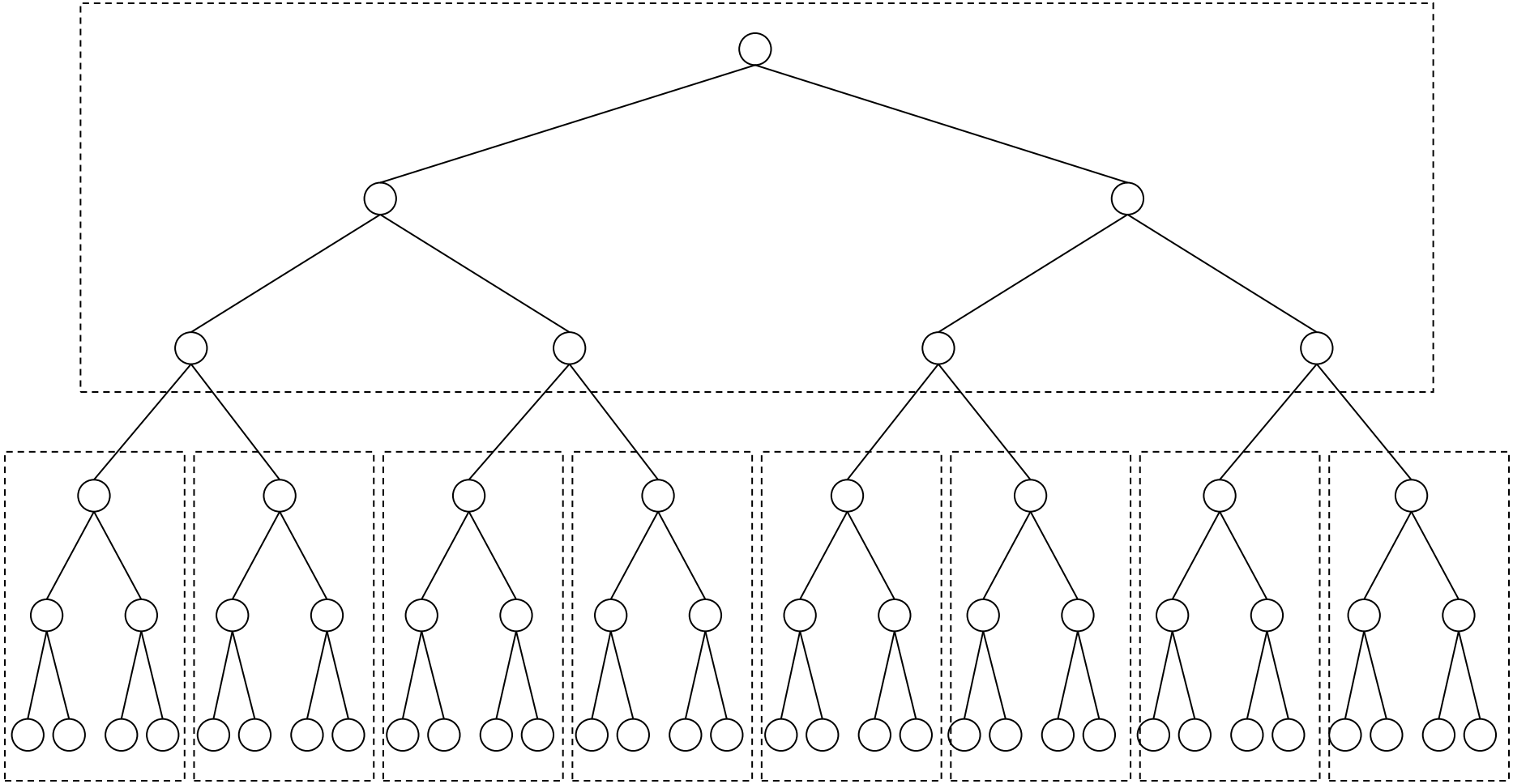
- Cây nhị phân đáp ứng khá đầy đủ các yêu cầu về biểu diễn cấu trúc dữ liệu
- Trong thực tế, kích thước dữ liệu thường là lớn hoặc vô cùng lớn. Đòi hỏi phải lưu trữ ở bộ nhớ ngoài
  - Dung lượng “vô hạn”
  - Tốc độ chậm hơn khoảng 10^5 lần so với bộ nhớ trong, ngoài ra còn là chi phí di chuyển đầu đọc

- Giả sử cây nhị phân với 10^6 nút thì chi phí truy cập khoảng $\log_2 10^6 \approx 20$
  - Sử dụng *cây nhiều nhánh* (lưu trữ trên đĩa) để biểu diễn cây tìm kiếm cỡ lớn
  - Để tăng tốc, mỗi lần truy xuất một nhóm dữ liệu

$\rightarrow$ Cây được chia thành các cây con, là một đơn vị truy xuất và gọi là trang

$\rightarrow$ Nếu cây có 10^6 nút và một trang có 10^2 nút thì chi phí truy cập một đối tượng khoảng $\log_{100} 10^6 = 3$.

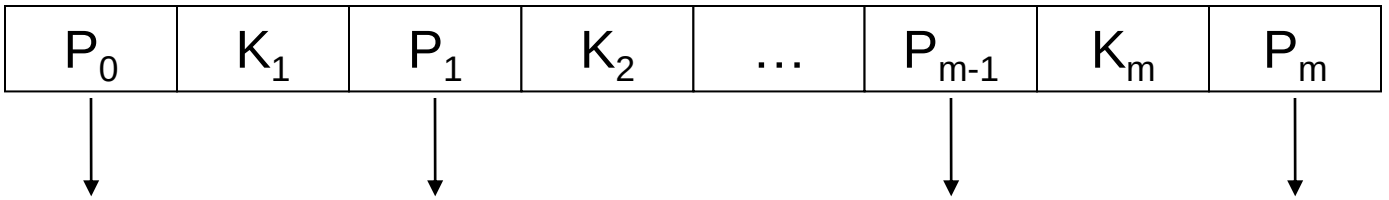
$\rightarrow$ Tuy nhiên, nếu cho cây này tăng trưởng tự nhiên thì trong trường hợp xấu nhất, chi phí có thể lên đến O(n!)



Định nghĩa: B-cây là cây cỡ lớn, được lưu trên đĩa.

Tổ chức của B–cây cấp *n* là:

- Mỗi trang chứa tối đa 2*n* khóa
- Mỗi trang, trừ trang gốc, chứa tối thiểu *n* khóa
- Mỗi trang, trừ trang lá, nếu có *m* khóa thì có *m* + 1 trang con
- Mọi trang lá xuất hiện ở cùng mức
- Các khóa được sắp xếp tăng dần từ trái qua phải



Các cấu trúc dữ liệu

```
const int n = 2;
const int nn = 4;
typedef struct page * ref;
struct item {
  int   key;
  ref   p;
  int   count;
};
struct page {
  int   m;
  item  e[nn + 1];
};
```

Tìm kiếm trên B-cây

1. Đọc trang vào bộ nhớ. Nếu *m* đủ lớn, dùng tìm kiếm nhị phân hay tuần tự
2. Nếu không tìm thấy (gọi *key* là khóa cần tìm):
   a) $k_i < key < k_{i+1}$ với $1 \le i < m$: tìm tiếp trên trang trỏ bởi P_i
   b) $k_m < key$: tìm tiếp trên trang trỏ bởi P_m
   c) $key < k_1$: tìm tiếp trên trang trỏ bởi P_0
3. Nếu con trỏ trỏ đến trang mới bằng NULL: không có *key* $\rightarrow$ kết thúc tìm kiếm

Thêm phần tử vào B-cây

1. Tìm trang lá thích hợp và chèn vào đúng vị trí
2. Nếu trang đầy: tiến hành tách trang
   - Mỗi trang chứa *n* khóa, phần tử đứng giữa được đưa lên trang cha
3. Trường hợp trang cha cũng trở nên đầy: Tiến trình tiếp tục theo chiều đi lên

   $\rightarrow$ Cây lớn lên từ lá đến gốc

Xóa phần tử trên B-cây

Giai đoạn 1: Tìm và xóa phần tử

- ở trang lá: đơn giản
- ở trang trong:
  - Tìm *phần tử thay thế* (ở trang lá)
    - Đi theo con trỏ phải nhất của trang con bên trái, hoặc
    - Đi theo con trỏ trái nhất của trang con bên phải
  - Sao chép dữ liệu của *phần tử thay thế* cho phần tử định xóa ban đầu
  - Xóa *phần tử thay thế*

Giai đoạn 2: Kiểm tra "tính cân bằng"

- Giảm số phần tử (*m*) của trang lá đi 1 đơn vị
- Nếu $m \geq n$: Dừng
- Nếu $m < n$: Xét *trang kế*

  > *n* phần tử: Cân bằng số lượng phần tử của 2 trang (thông qua phần tử trung gian ở trang cha)

  = *n* phần tử: Kéo phần tử trung gian ở trang cha xuống và ghép 2 trang $\rightarrow$ trang đầy

  Quá trình có thể lan truyền ngược … (nếu trang cha trở nên thiếu) … về đến gốc $\rightarrow$ cây giảm chiều cao

```
ref   root, q;
bool  h;
item  u;
  …
root = NULL;
for (i = 0; i < n; i++) {
  timthem(a[i], root, h, u);
  if (h) {
    q = root;
    root = new page;
    root->m = 1;
    root->e[0].p = q;
    root->e[1] = u;
  }
}
```

```
void  timthem(int x, ref a, bool &h, item &v) {
  int   r;
  ref   q;
  item  u;

  if (a == NULL) {
    h = true;
    v.key = x;
    v.count = 1;
    v.p = NULL;
  }
  …
}
```

```
else {
  a->e[0].key = x;  // Lính canh
  for (r = a->m; a->e[r].key > x; r--) ;
  if ((r) && (a->e[r].key == x)) {  // Đã có
    a->e[r].count ++;
    h = false;
  }
  else {
    if (r == 0)     q = a->e[0].p;
    else            q = a->e[r].p;
    timthem(x, q, h, u);
    if (h)    // a trỏ vào trang lá
      them(r, a, h, u, v);
  }
}
```

```
void  them(int r, ref a, bool &h, item &u,
                                        item &v) {

  if (a->m < nn) {    // Trang chưa đầy
    a->m ++;
    h = false;
    for (i = a->m; i >= r + 2; i --)
      a->e[i] = a->e[i - 1];
      a->e[r + 1] = u;
  }
  …
}
```

```
else {   // Trang trỏ bởi a đầy
  ref b = new page;
  if (r <= n) {
    if (r == n)
      v = u;
    else {
      v = a->e[n];
      for (i = n; i >= r + 2; i--)
         a->e[i] = a->e[i - 1];
      a->e[r + 1] = u;
    }
    for (i = 1; i <= n; i++)
      b->e[i] = a->e[i + n];
  }
  …
}
```

```
else {
  r -= n;
  v = a->e[n + 1];
  for (i = 1; i <= r - 1; i++)
    b->e[i] = a->e[i + n + 1];
  b->e[r] = u;
  for (i = r + 1; i <= n; i++)
    b->e[i] = a->e[i + n];
}
a->m = b->m = n;
b->e[0].p = v.p;
v.p = b;
```

```
ref   root, q;
bool  h;
int   x;
…
cout << "\nNhap vao khoa can xoa: ";
cin >> x;
timxoa(x, root, h);
if (h)
  if (root->m == 0) {
    q = root;
    root = q->e[0].p;
    delete q;
  }
…
```

```
void  timxoa(int x, ref a, bool &h) {
  if (a == NULL)
    h = false;
  else {
    a->e[0].key = x;
    for (r = a->m; a->e[r].key > x; r--)
      ;
    if (r == 0)
      q = a->e[0].p;
    else
      if ((r) && (a->e[r].key == x))
        q = a->e[r - 1].p;
      else
        q = a->e[r].p;
    …
} // void
```

```
  if ((r) && (a->e[r].key == x))
    if (q == NULL) {      // Trang la
      a->m --;
      h = a->m < n;
      for (i = r; i <= a->m; i++)
        a->e[i] = a->e[i + 1];
    }
    else {
      xoa(a, q, r, h);// Tim phan tu thay the
      if (h)  canbang(a, q, r - 1, h);
    }
  else {
    timxoa(x, q, h);
    if (h)  canbang(a, q, r, h);
  }
}
```

```
void  xoa(ref a, ref p, int r, bool &h) {
  ref q = p->e[p->m].p;

  if (q) {
    xoa(a, q, r, h);
    if (h)
      canbang(p, q, p->m, h);
  }
  else {  // p trỏ đến trang lá
    p->e[p->m].p = a->e[r].p;
    a->e[r] = p->e[p->m]; // Sao chép
    p->m --;
    h = p->m < n;
  }
}
```

```
void  canbang(ref c, ref a, int s, bool &h)
{
  int mc = c->m;
  if (s < mc) {
    s++;
    b = c->e[s].p;
    mb = b->m;

    k = (mb - n + 1)/2; //Số phần tử chuyển giao
    a->e[n] = c->e[s]; // Lấy phần tử ở trang cha
    a->e[n].p = b->e[0].p;
  …
} // void
```

```
if (k > 0) { // Khong can phai ghep trang
  for (i = 1; i <= k - 1; i++)
    a->e[n + i] = b->e[i];
  c->e[s] = b->e[k];
  c->e[s].p = b;
  b->e[0].p = b->e[k].p;
  mb -= k;
  for (i = 1; i <= mb; i++)
    b->e[i] = b->e[i + k];
  b->m = mb;
  a->m = (n - 1) + k;
  h = false;
}
```

```
  else {  // Trộn trang a và b
    for (i = 1; i <= n; i++)
      a->e[n + i] = b->e[i];
    for (i = s; i <= mc - 1; i++)
      c->e[i] = c->e[i + 1];
    a->m = nn;
    c->m = mc - 1;

    delete  b;
  }
}
…
```

```
else {  // s = mc
  if (s == 1)
    b = c->e[0].p;
  else
    b = c->e[s - 1].p;

  mb = b->m + 1;
  k = (mb - n) / 2; // Số phần tử chuyển giao
  …
```

```
 if (k > 0) {
  for (i = n - 1; i >= 1; i--)
    a->e[i + k] = a->e[i];
  a->e[k] = c->e[s];
  a->e[k].p = a->e[0].p;
  mb -= k;
  for (i = k - 1; i >= 1; i--)
    a->e[i] = b->e[i + mb];
  a->e[0].p = b->e[mb].p;
  c->e[s] = b->e[mb];
  c->e[s].p = a;
  b->m = mb - 1;
  a->m = (n - 1) + k;
  h = false;
}
```

…

```
  else {  // Trộn hai trang a và b
    b->e[mb] = c->e[s];
    b->e[mb].p = a->e[0].p;
    for (i = 1; i <= n - 1; i++)
      b->e[i + mb] = a->e[i];

    b->m = nn;
    c->m = mc - 1;

    delete a;
  }
}
```