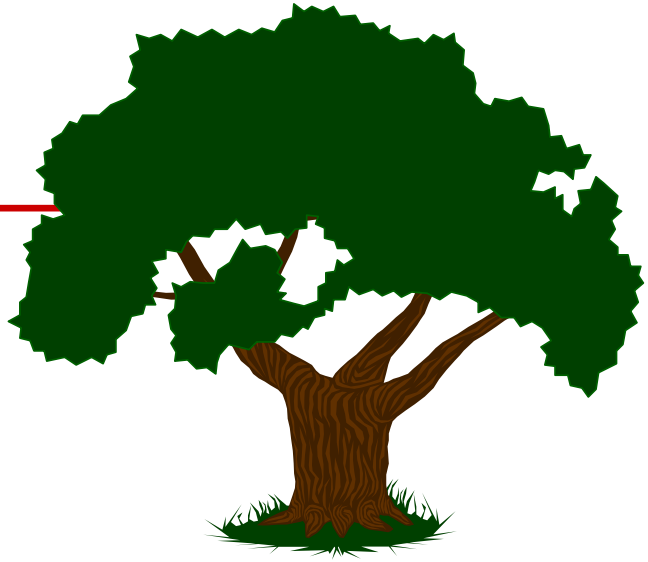
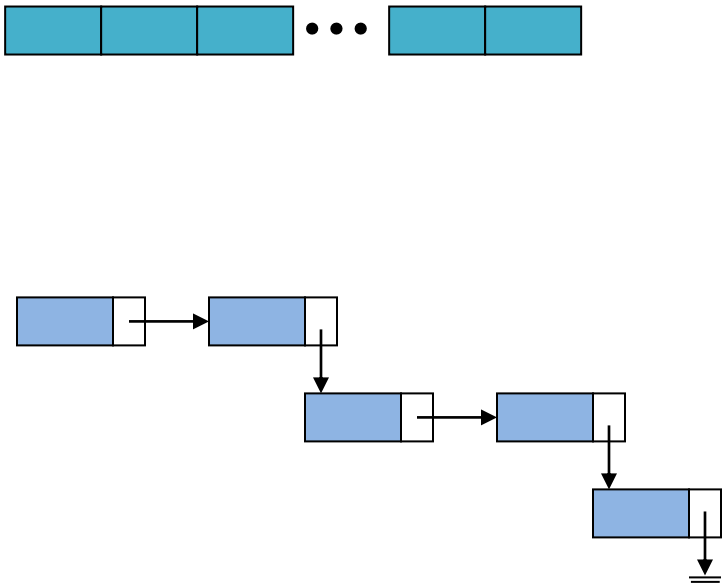
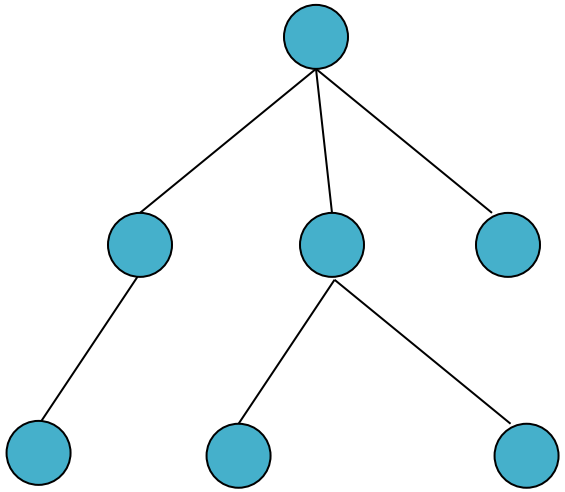
Chương 6: Cấu trúc cây (Tree structure)



Cấu trúc dữ liệu

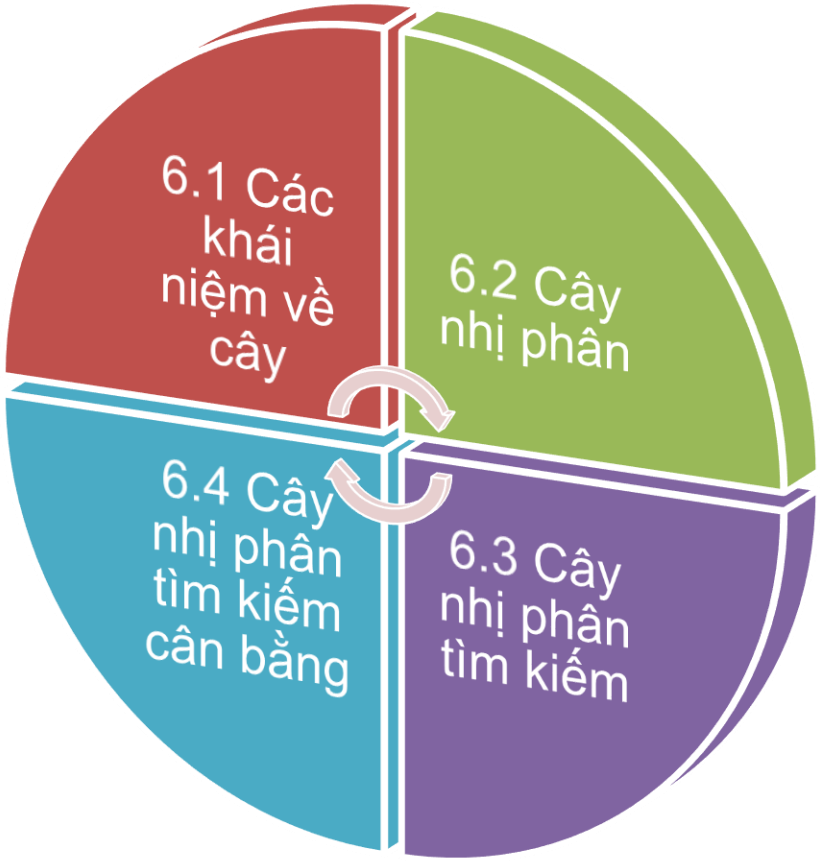


Linear



Hierarchical structures

Nội dung


6.1 Các khái niệm về cây
6.2 Cây nhị phân
6.3 Cây nhị phân tìm kiếm
6.4 Cây nhị phân tìm kiếm cân bằng

Nội dung

6.1 Các khái niệm về cây

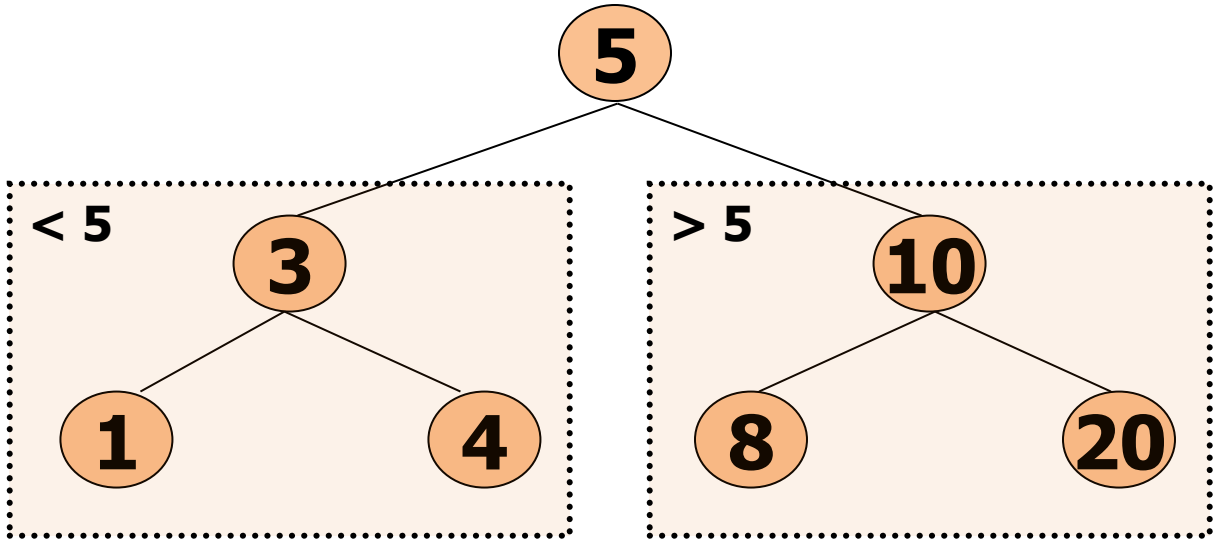
6.2 Cây nhị phân

6.3 Cây nhị phân tìm kiếm

6.4 Cây nhị phân tìm kiếm cân bằng

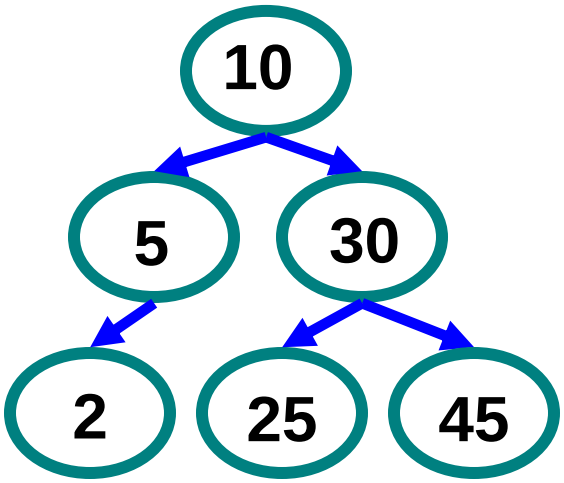
6.3 Binary Search Tree

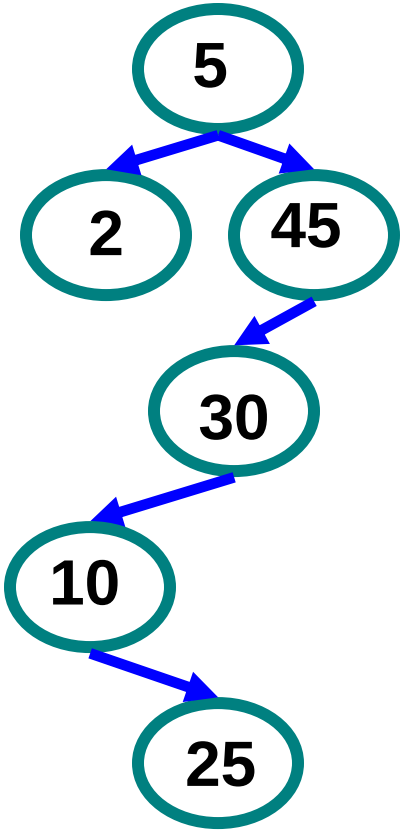
- BST là cây nhị phân mà mỗi nút thoả
  - Giá trị của tất cả nút con trái < nút gốc
  - Giá trị của tất cả nút con phải > nút gốc



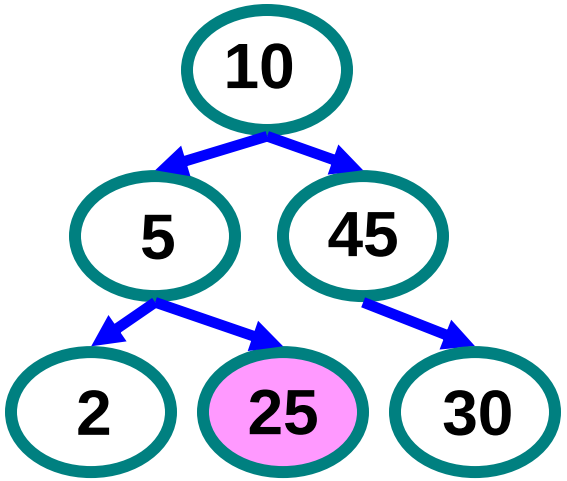
6.3 Binary Search Tree

- Ví dụ



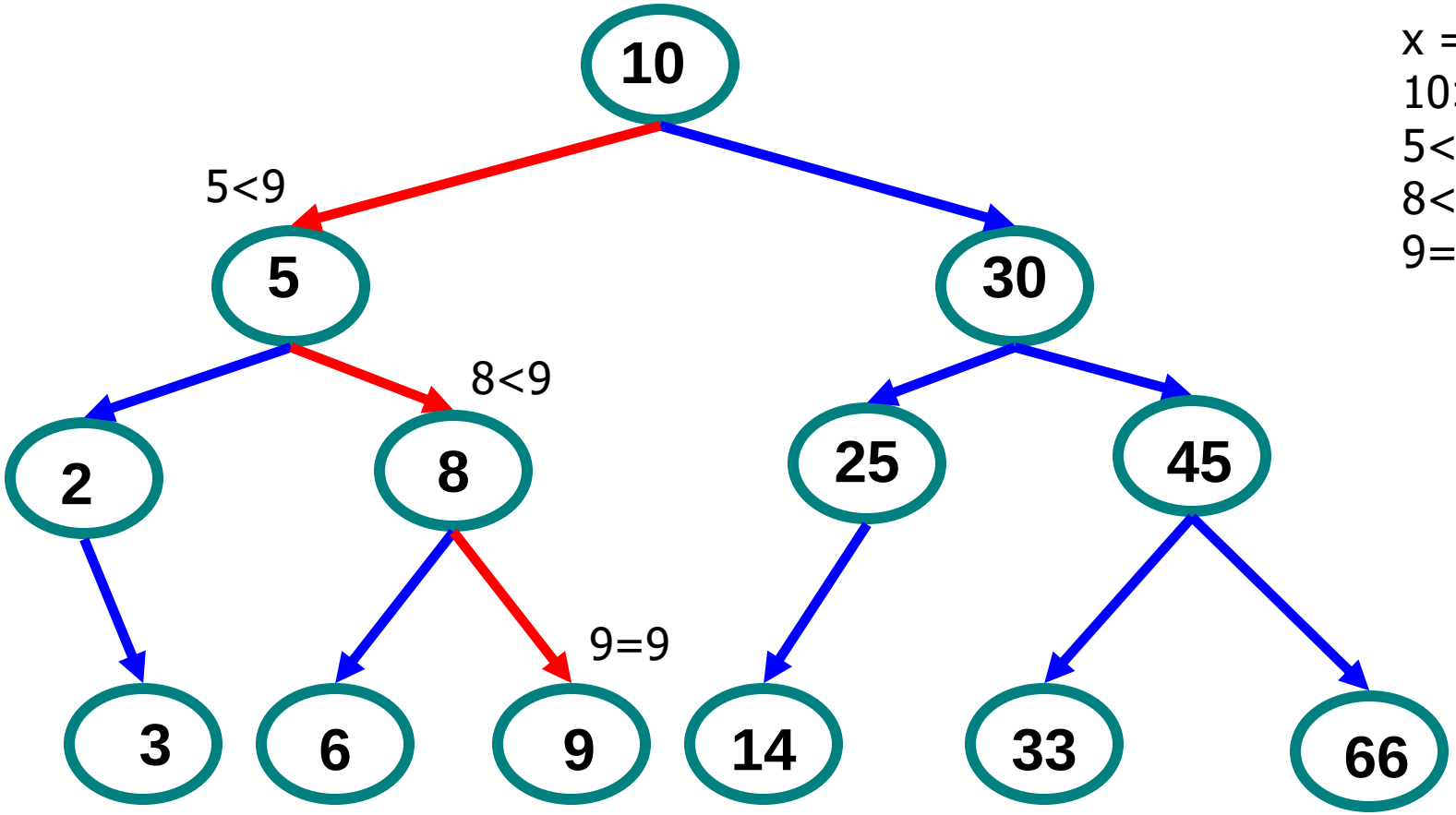


Binary search trees



Non-binary search tree

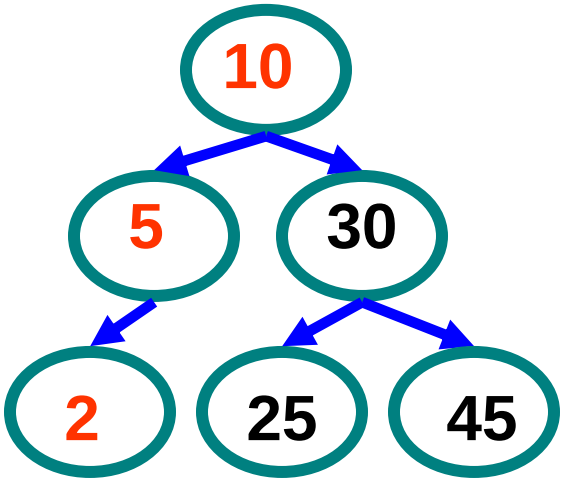
6.3 Binary Search Tree



x = 9
10>9, left
5<9, right
8<9, right
9=9, Tìm thấy

6.3 Binary Search Tree

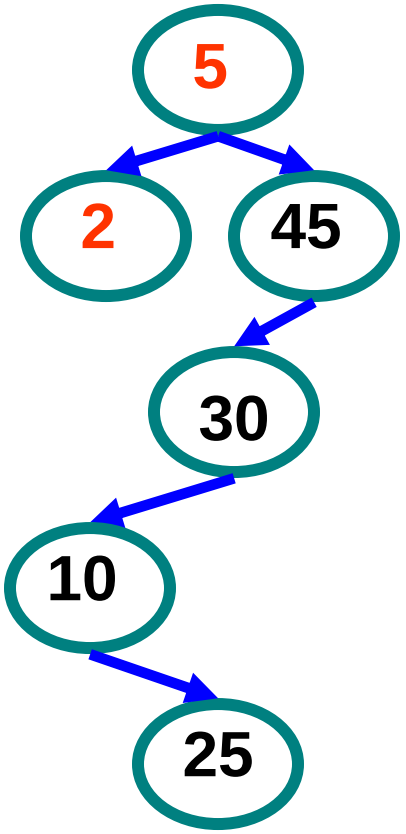
- Tìm (2)



10 > 2, left

5 > 2, left

2 = 2, found

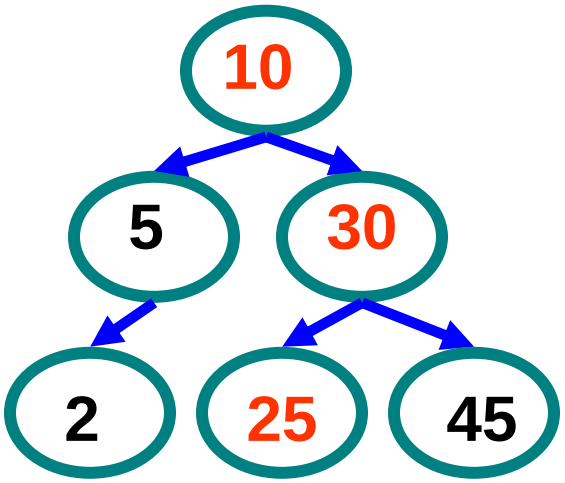


5 > 2, left

2 = 2, found

6.3 Binary Search Tree

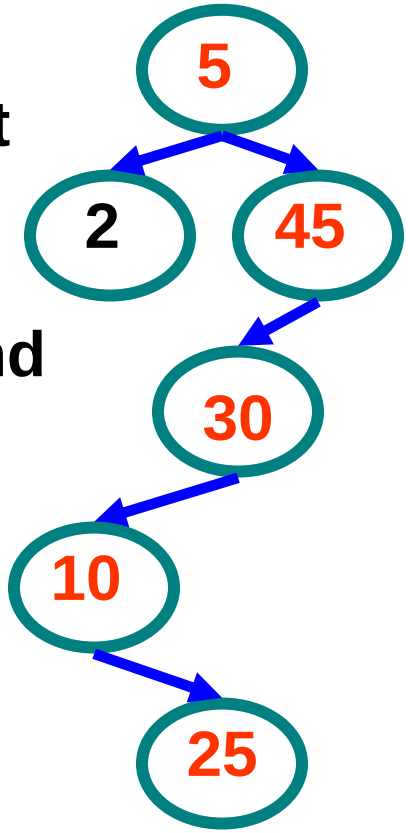
- Tìm (25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

30 > 25, left

10 < 25, right

25 = 25, found

6.3 Binary Search Tree

- Thời gian tìm kiếm
  - Dựa trên chiều cao của cây
  - Cây cân bằng
    - O(log(n))
  - Cây ko cân bằng
    - O(n)
    - Tương tự tìm kiếm trên danh sách, mảng ko sắp

6.3 Binary Search Tree

- Search
  - Xuất phát từ gốc
    - Nếu nút = NULL => ko tìm thấy
    - Nếu khoá x = khóa nút gốc => tìm thấy
    - Ngược lại nếu khoá x < khoá nút gốc => Tìm trên cây bên trái
    - Ngược lại => tìm trên cây bên phải

6.3 Binary Search Tree

- Search

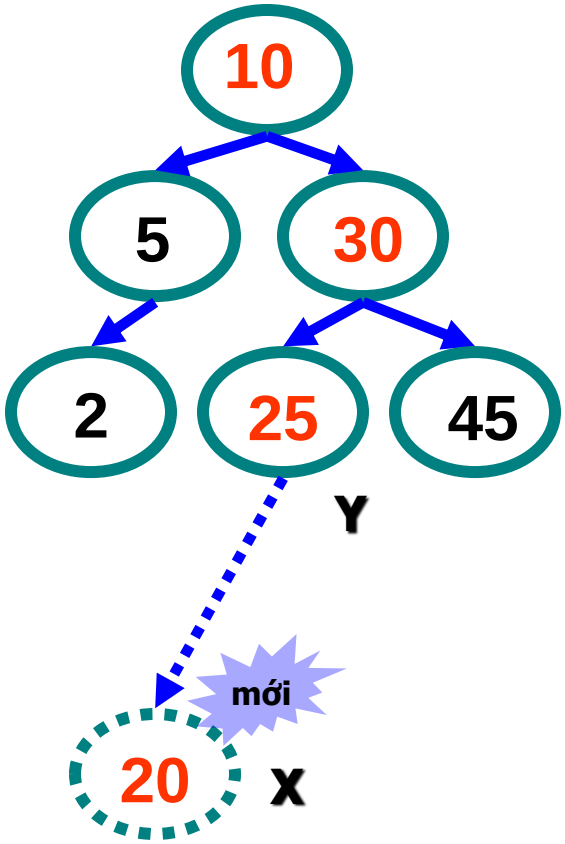
```
1. NodePtr Search(NodePtr node, int x)
2. {
3.     NodePtr p = node;
4.     if (p != NULL)
5.    if (node->info >x)
6.          p = Search(node->left, x);
7.    else
8.          if (node->info < x)
9.               p = Search(node->right, x);
10.    return  p;
11.}
```

6.3 Binary Search Tree

- Xây dựng cây BST
  - Chèn
  - Xóa
- Luôn duy trì tính chất
  - Giá trị nhỏ hơn ở bên cây con trái
  - Giá trị lớn hơn ở bên cây con phải

6.3 Binary Search Tree

- Insert
  - Thực hiện tìm kiếm giá trị x
  - Tìm đến cuối nút Y (nếu x ko tồn tại trong cây)
  - Nếu x < y, thêm nút lá x bên trái của Y
  - Nếu x > y, thêm nút lá x bên phải của Y



6.3 Binary Search Tree

```
1. void Insert(NodePtr pTree, int x) {
2.    NodePtr node;
3.    if (pTree == NULL)              return;
4.    if (pTree->info == x)           return;
5.    if (pTree->info > x) {
6.          if (pTree->left == NULL) {
7.                node = CreateNode(x);
8.                pTree->left = node;
9.          }
10.         else  Insert(pTree->left, x);
11.   }
12.   else
13.         if (pTree->right == NULL) {
14.               node = CreateNode(x);
15.               pTree->right = node;
16.         }
17.         else  Insert(pTree->right, x);
18.}
```

6.3 Binary Search Tree

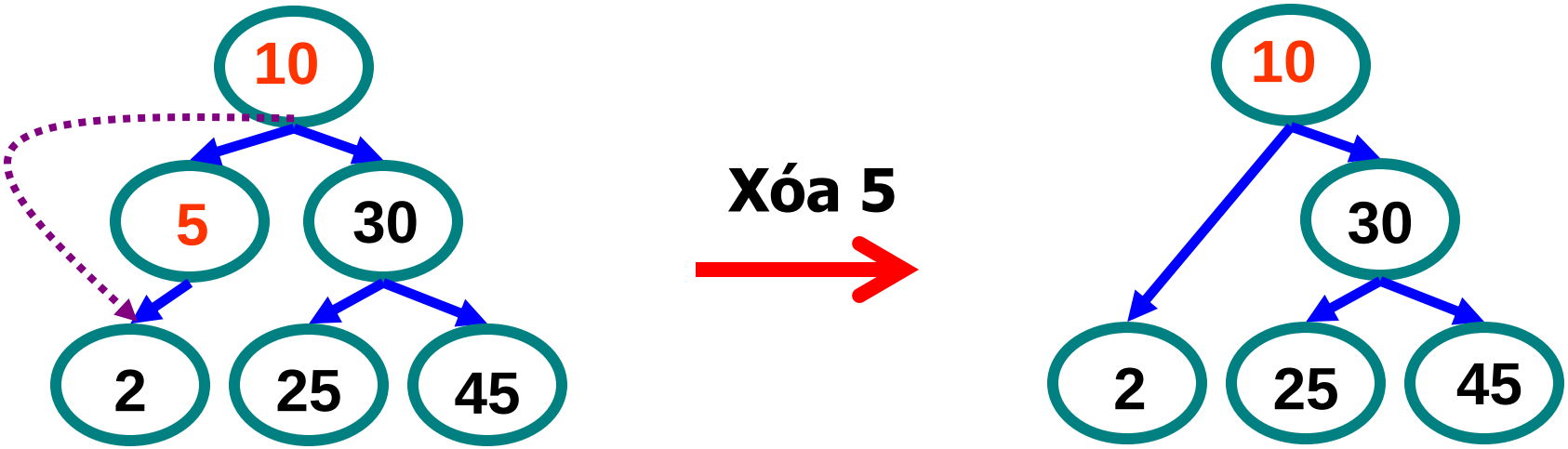
- Delete: xóa nhưng phải đảm bảo vẫn là cây BST
  - Thực hiện tìm nút có giá trị x
  - Nếu nút là nút lá, delete nút
  - Ngược lại
    - Thay thế nút bằng một trong hai nút sau
      - Y là nút lớn nhất của cây con bên trái
      - Z là nút nhỏ nhất của cây con bên phải
    - Chọn nút Y hoặc Z để thế chỗ
    - Giải phóng nút

6.3 Binary Search Tree

- Trường hợp 1: nút p là nút lá, xoá bình thường

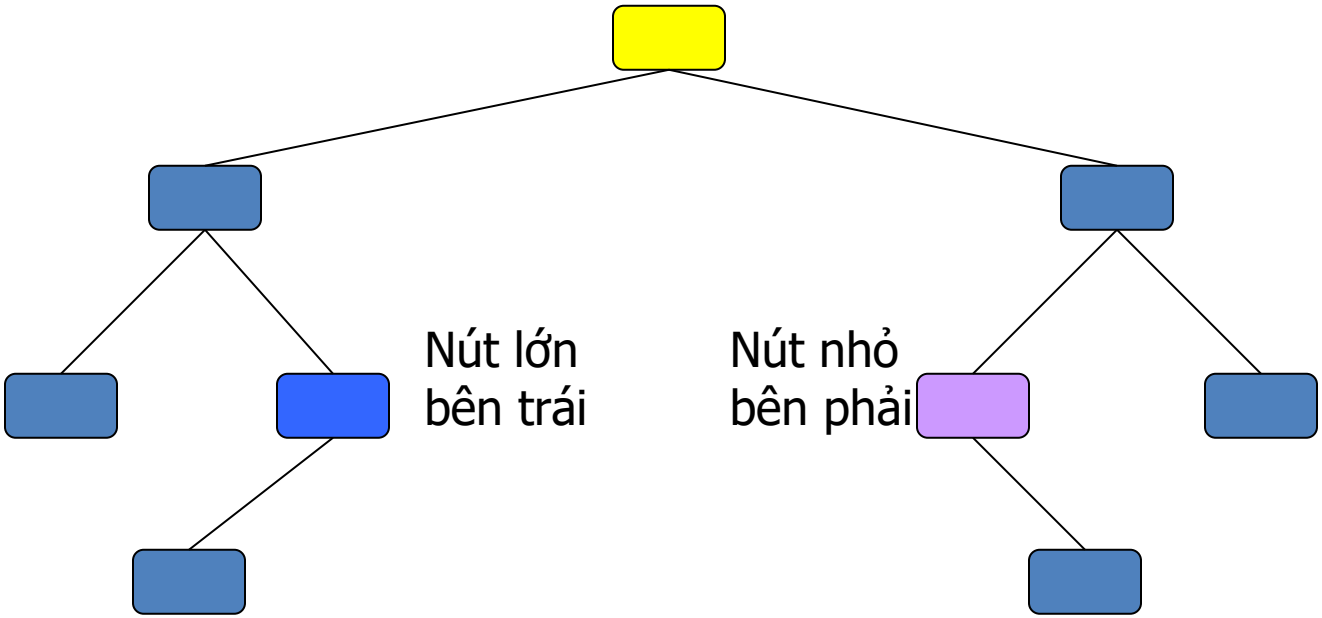
6.3 Binary Tree Search

- Trường hợp 2: p chỉ có 1 cây con, cho nút cha của p trỏ tới nút con duy nhất của nó, rồi hủy p



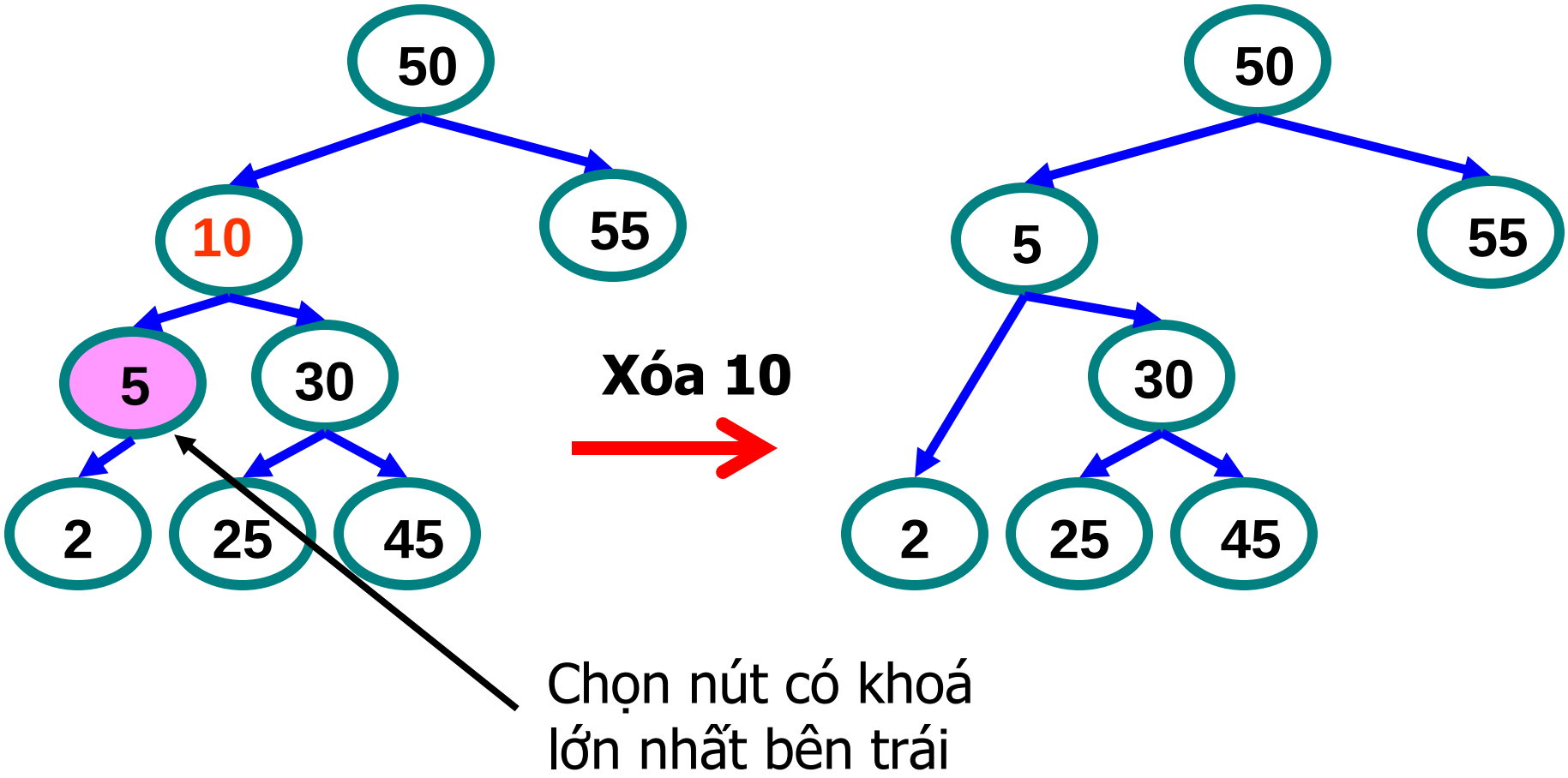
6.3 Binary Search Tree

- Trường hợp 3: nút p có 2 cây con, chọn nút thay thế theo 1 trong 2 cách như sau
  - Nút lớn nhất trong cây con bên trái
  - Nút nhỏ nhất trong cây con bên phải



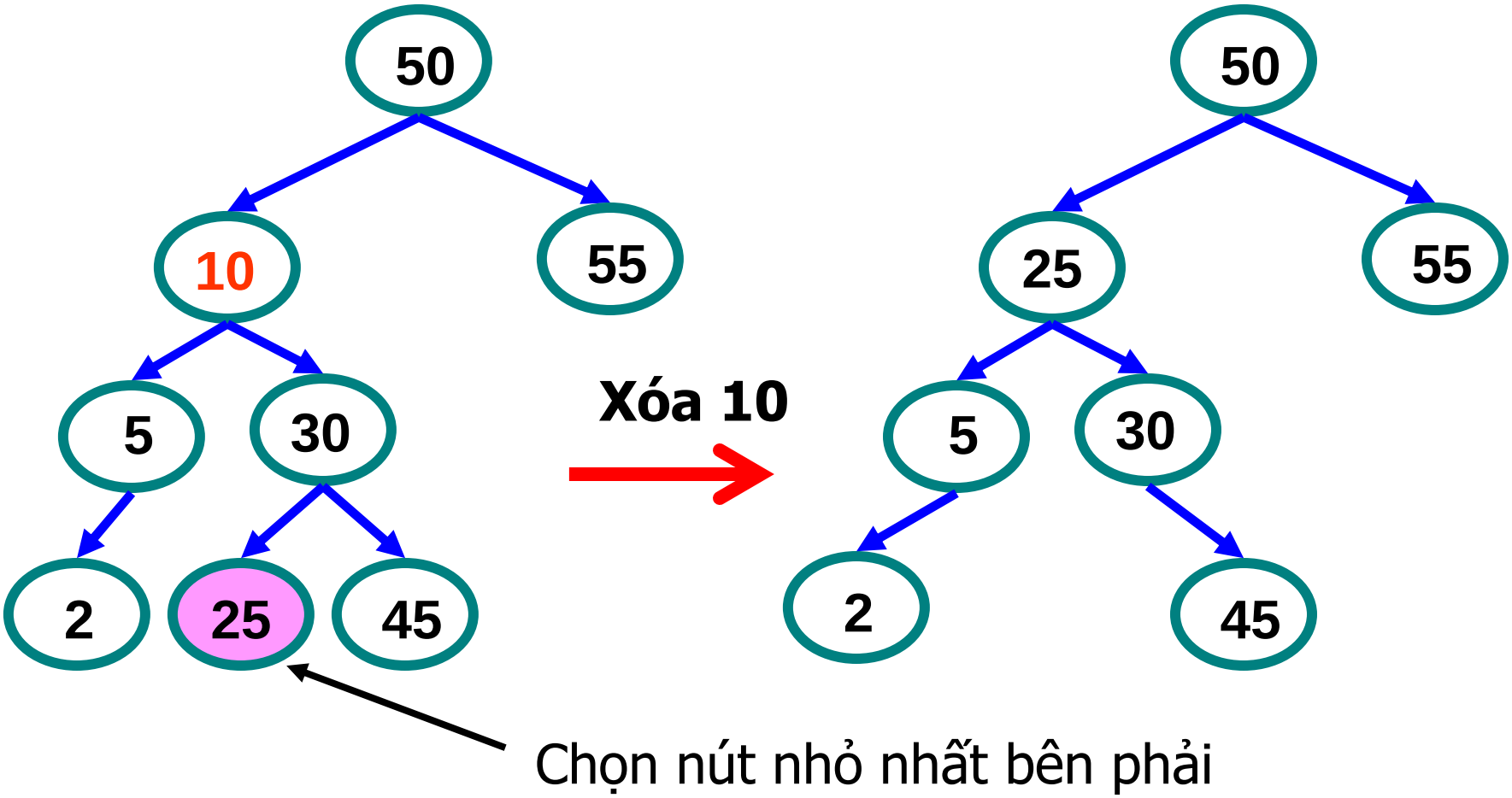
6.3 Binary Search Tree

- Delete: nút 10 cách 1



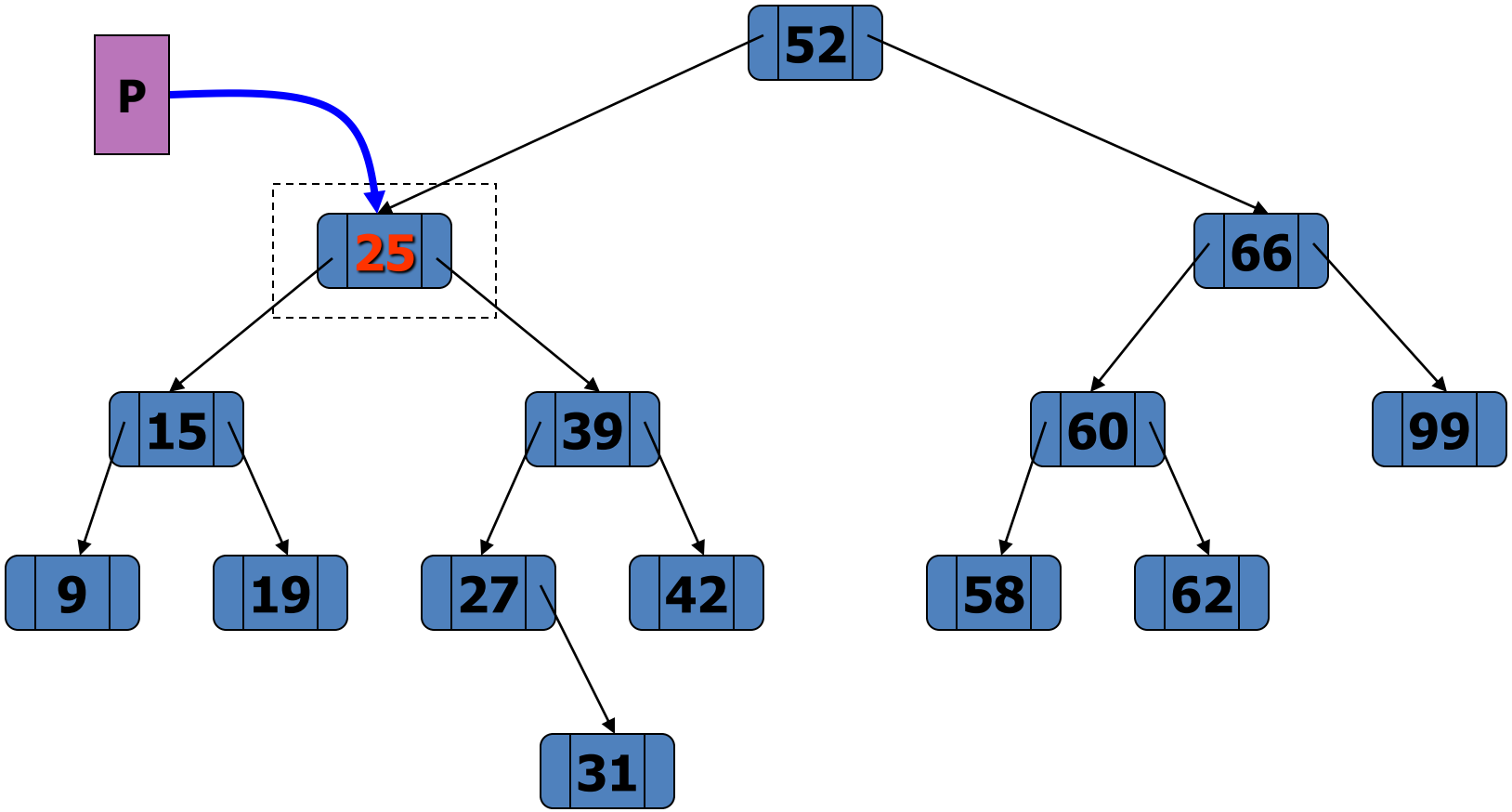
6.3 Binary Search Tree

- Delete: nút 10 cách 2



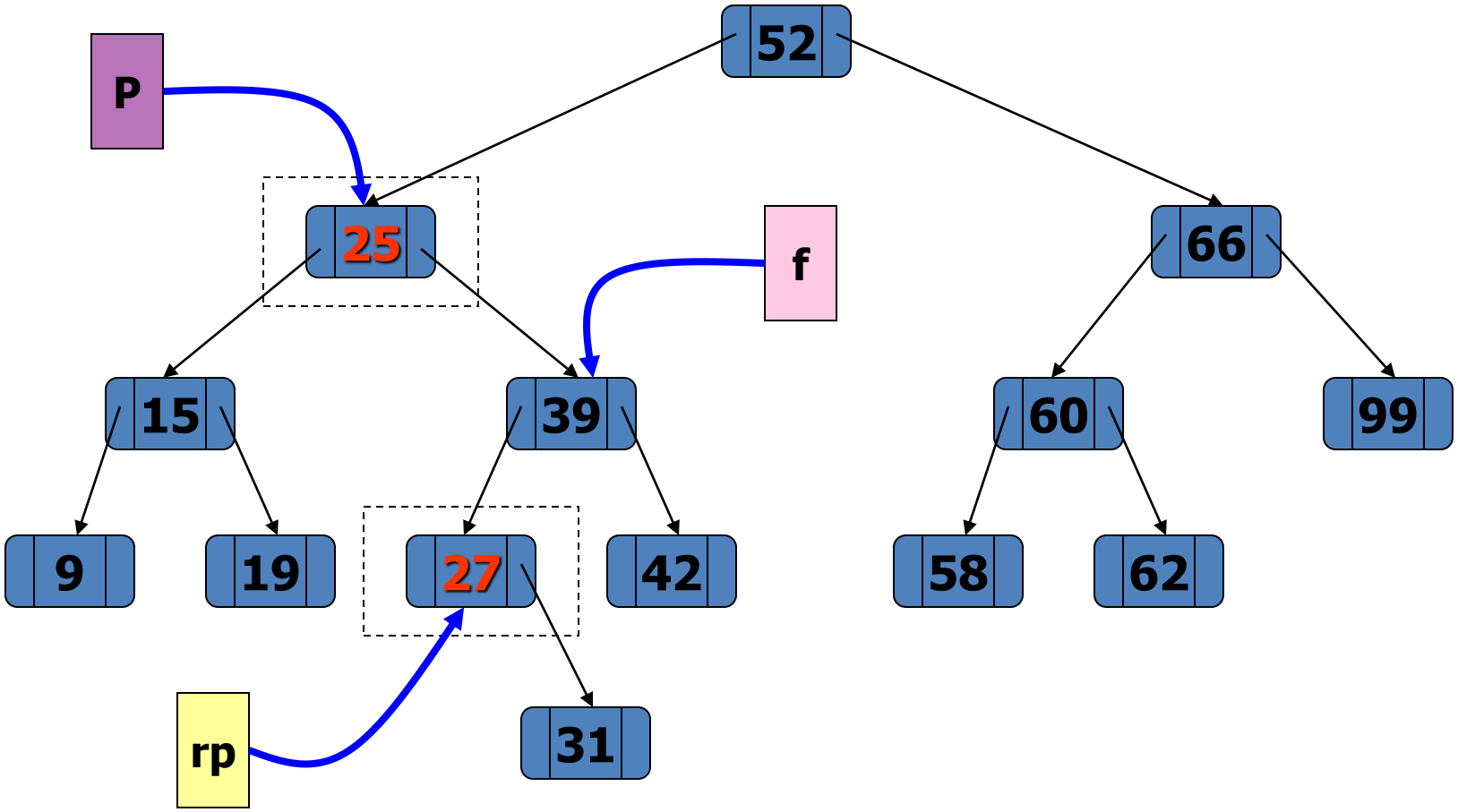
6.3 Binary Search Tree

- Minh họa xóa (25)



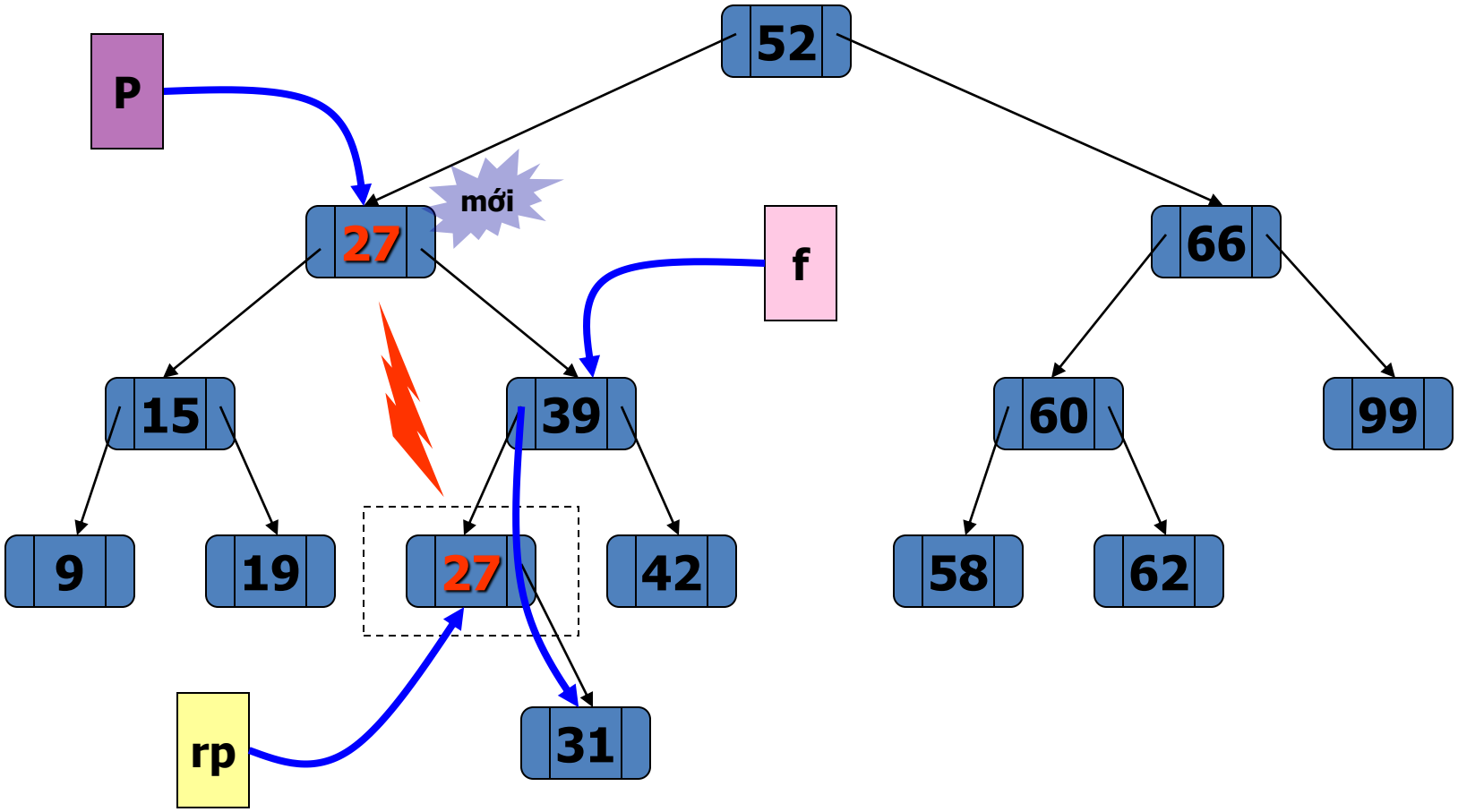
6.3 Binary Search Tree

- Minh họa xóa (25)



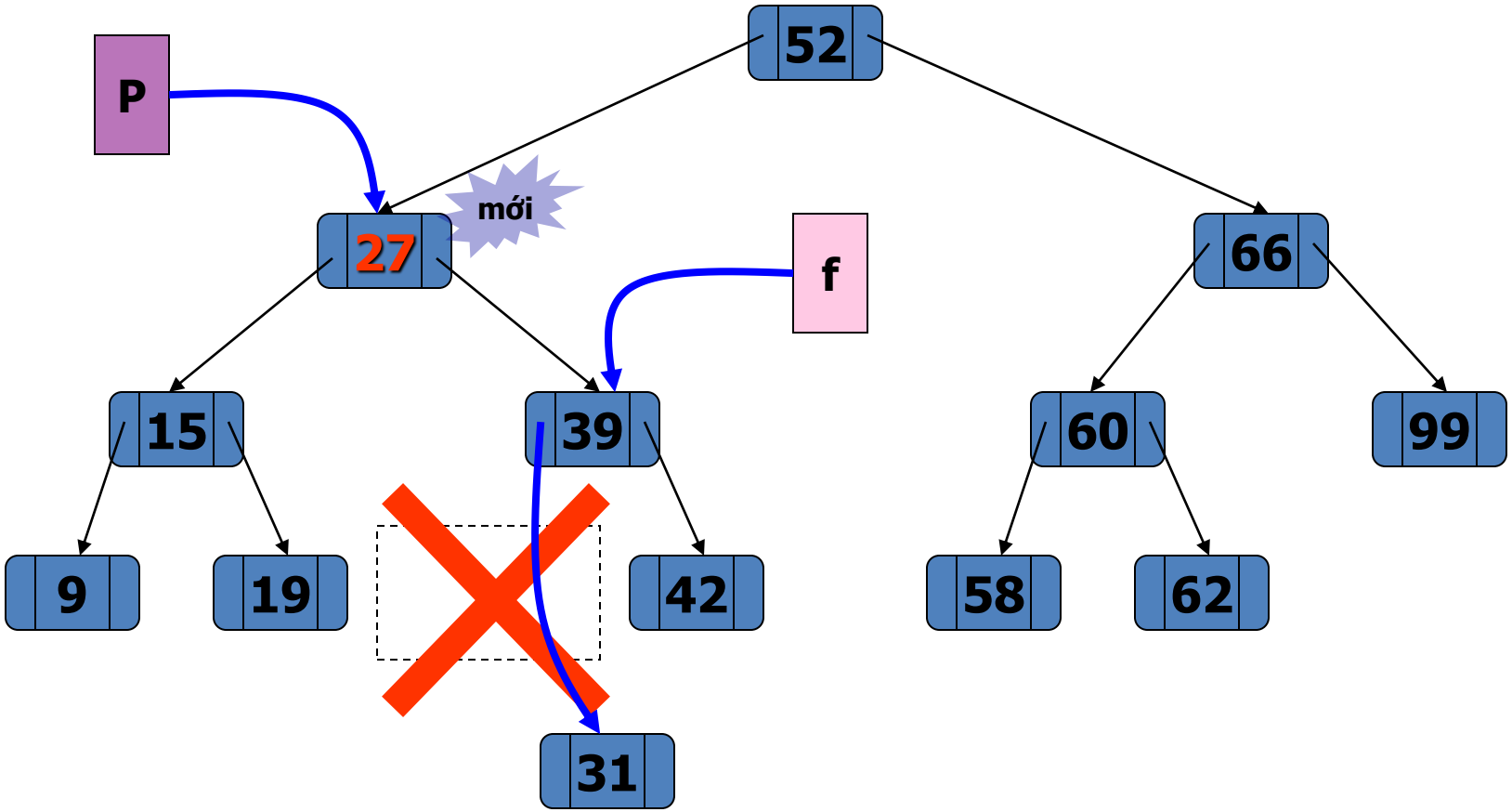
6.3 Binary Search Tree

- Minh họa xóa (25)



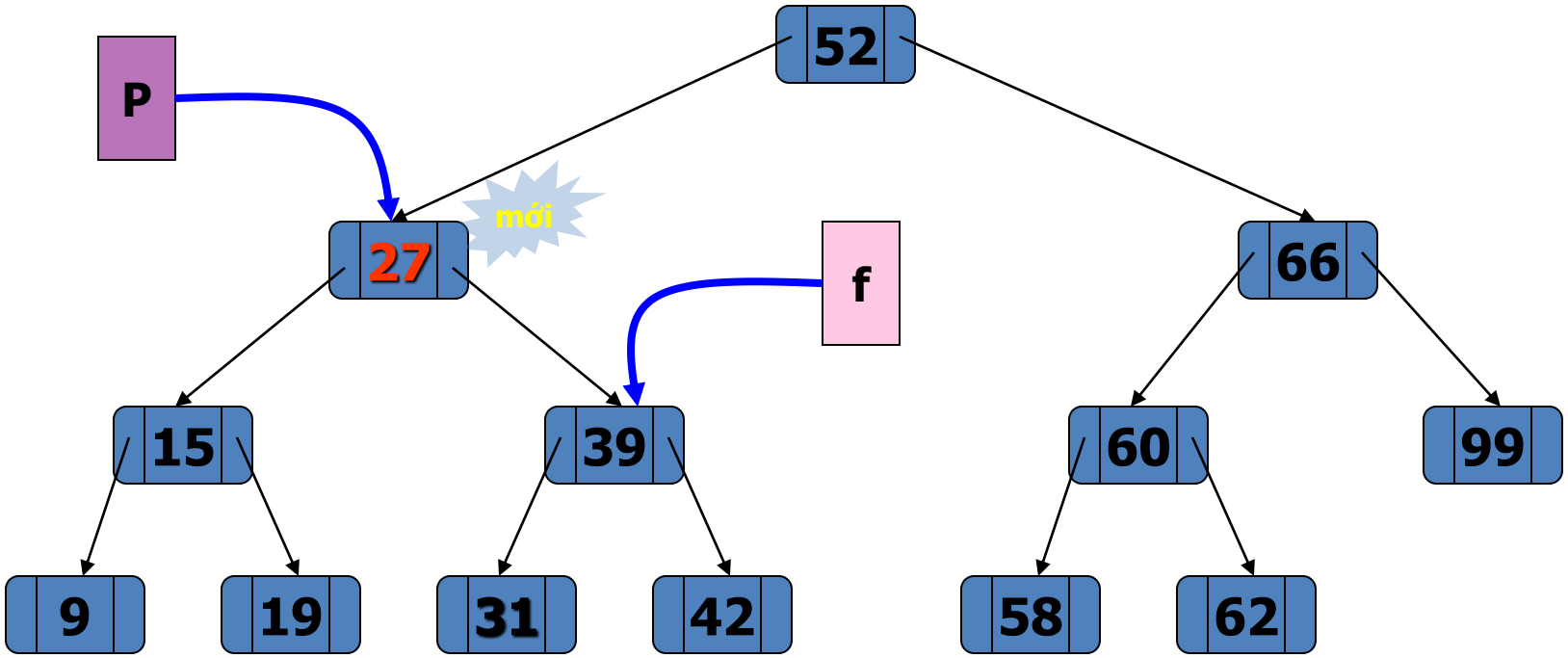
6.3 Binary Search Tree

- Minh họa xóa (25)

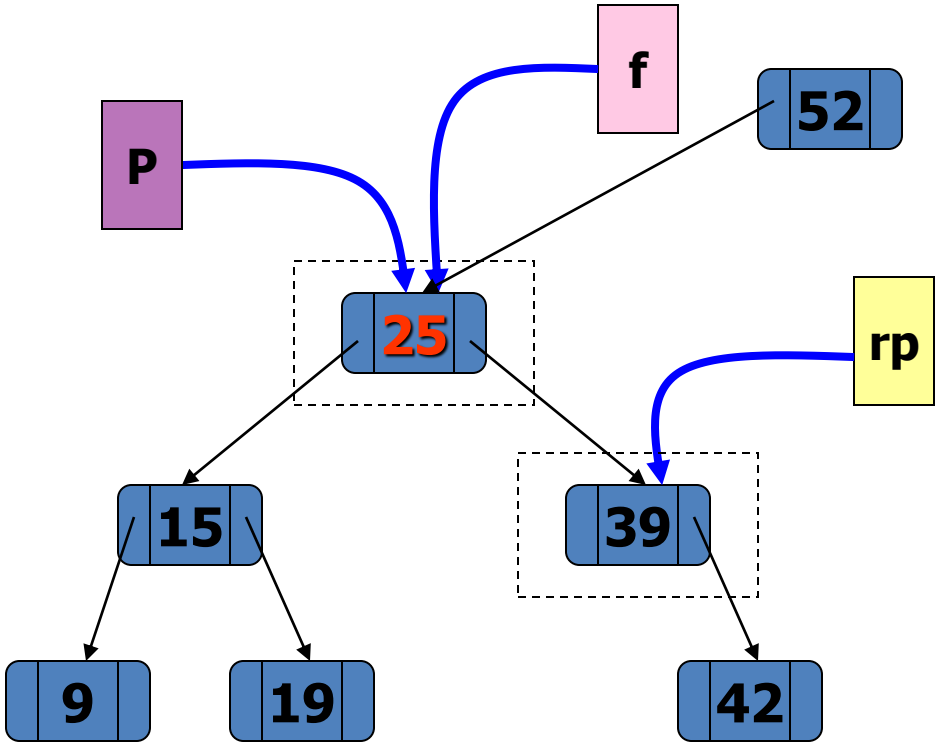


6.3 Binary Search Tree

- Minh họa xóa (25)



6.3 Binary Search Tree

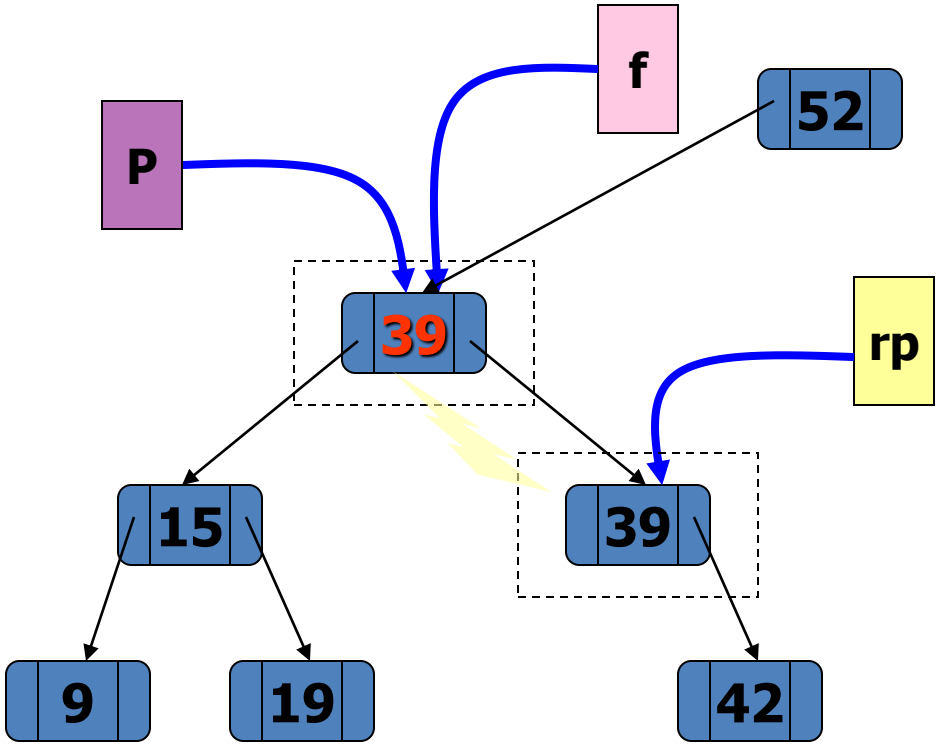


Trường hợp đặc biệt:

f == p

Nút thế mạng rp là nút con phải của nút p cần xoá

6.3 Binary Search Tree



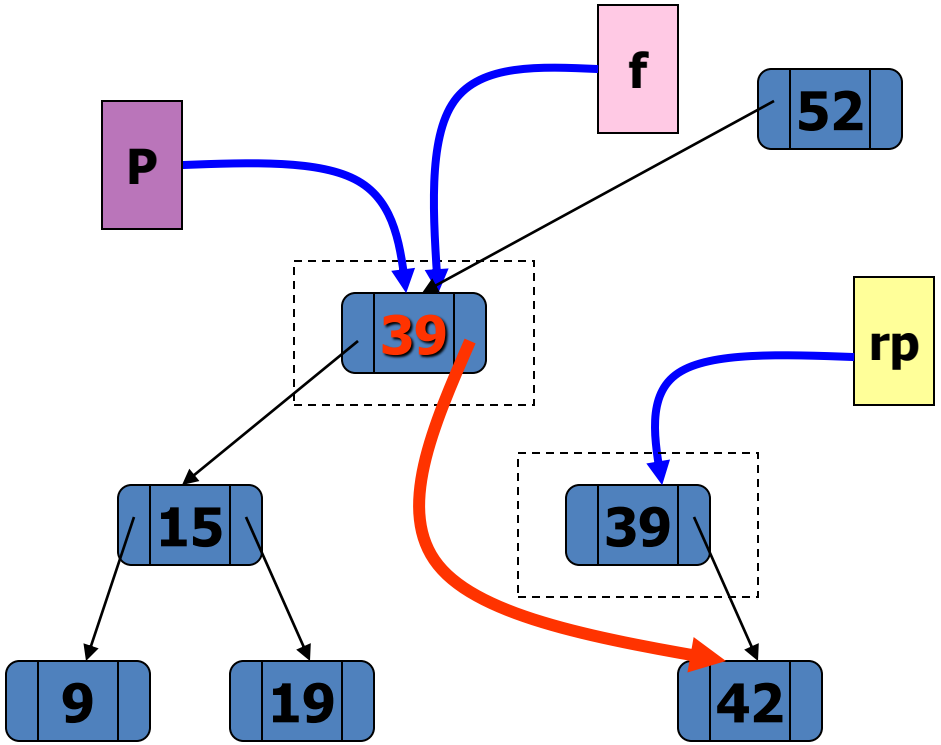
Trường hợp đặc biệt:

f == p

Nút thế mạng rp là nút con phải của nút p cần xoá

- Đưa giá trị của nút rp lên nút p

6.3 Binary Search Tree



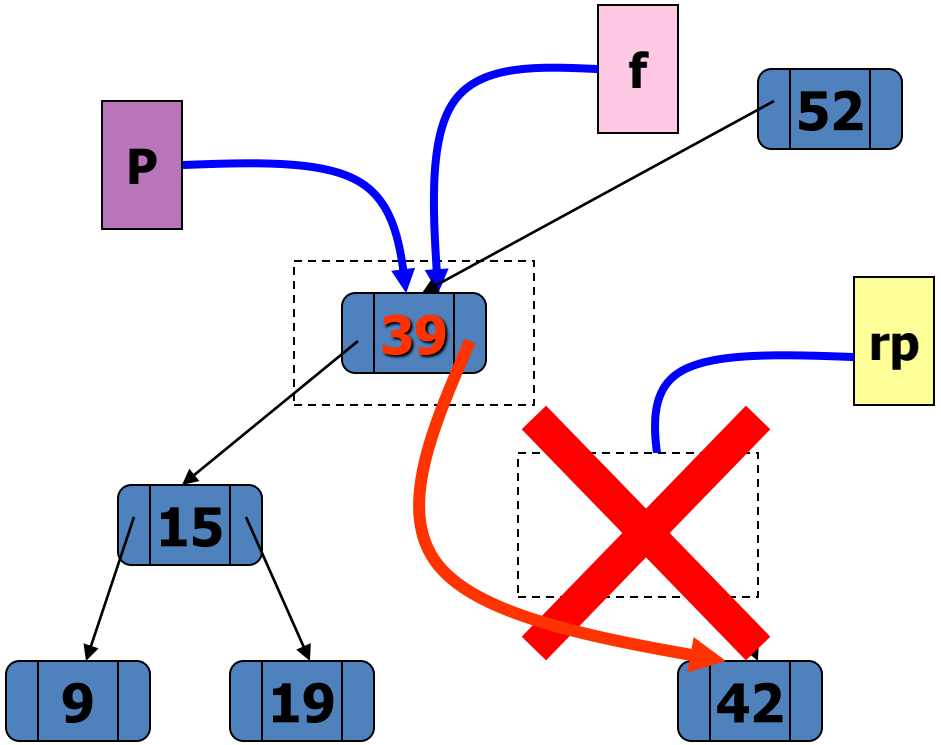
Trường hợp đặc biệt:

f == p

Nút thế mạng rp là nút con phải của nút p cần xoá

- Chuyển liên kết phải của p đến liên kết phải của rp

6.3 Binary Search Tree



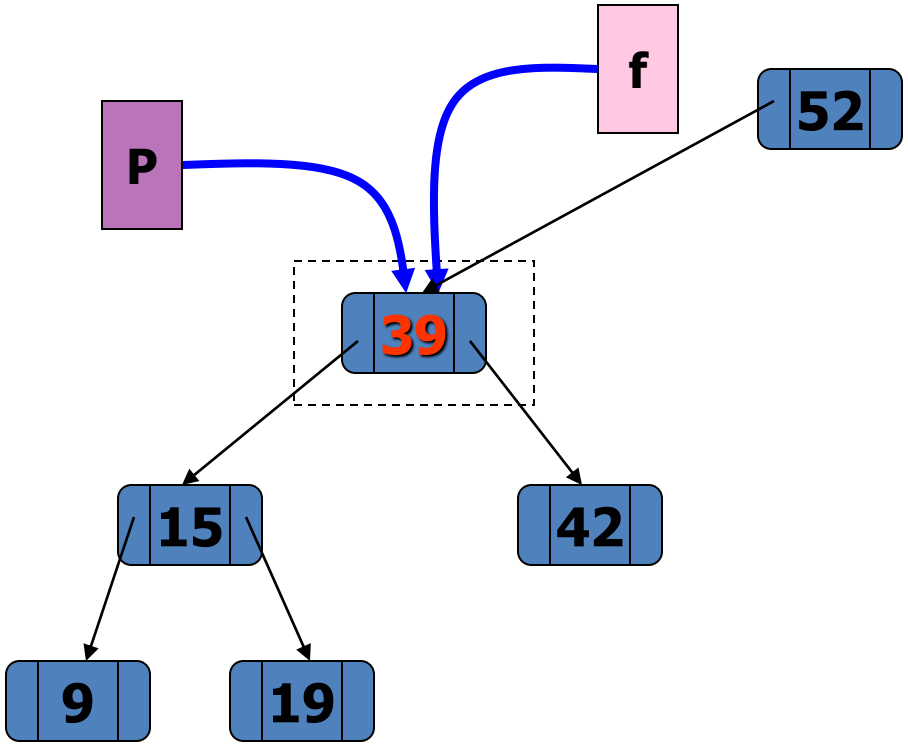
Trường hợp đặc biệt:

f == p

Nút thế mạng rp là nút con phải của nút p cần xoá

- Xoá nút rp

6.3 Binary Search Tree



Trường hợp đặc biệt:

f == p

Nút thế mạng rp là nút con phải của nút p cần xoá

- Sau khi xoá

6.3 Binary Search Tree

- Remove (NodePtr &T, int x)
  - Nếu T = NULL ⇒ thoát
  - Nếu T->info > x ⇒ Remove(T->left, x)
  - Nếu T->info <x ⇒ Remove(T->right, x)
  - Nếu T->info = x
    - P = T
    - Nếu T có 1 nút con thì T trỏ đến nút con đó
    - Ngược lại có 2 con
      - Gọi f = p và rp = p->right;
      - Tìm nút rp sao cho rp->left = null và nút f là nút cha nút rp
      - Thay đổi giá trị nội dung của T và rp
      - Nếu f = p (trường hợp đặc biệt) thì: f->right = rp->right;
      - Ngược lại: f->left = rp->right;
      - P = rp; // p trỏ tới rp để xoá
    - Xoá P

6.3 Binary Search Tree

```
1.  int  Remove(NodePtr &T, int x)
2.  {
3.      if ( T == NULL)
4.            return FALSE; //không tìm thấy nút cần xoá
5.      if (T->info >x)      // tìm bên trái
6.            return Remove(T->left, x);
7.      if (T->info <x)      // tìm bên phải
8.            return Remove(T->right, x);
9.      NodePtr p, f, rp;
10.     p = T;               // p biến tạm trỏ đến T
11.     if ( T->left == NULL)          // có 1 cây con
12.           T = T->right;
13.     else
14.           if (T->right == NULL)    // có 1 cây con
15.                 T = T->left;
```

6.3 Binary Search Tree

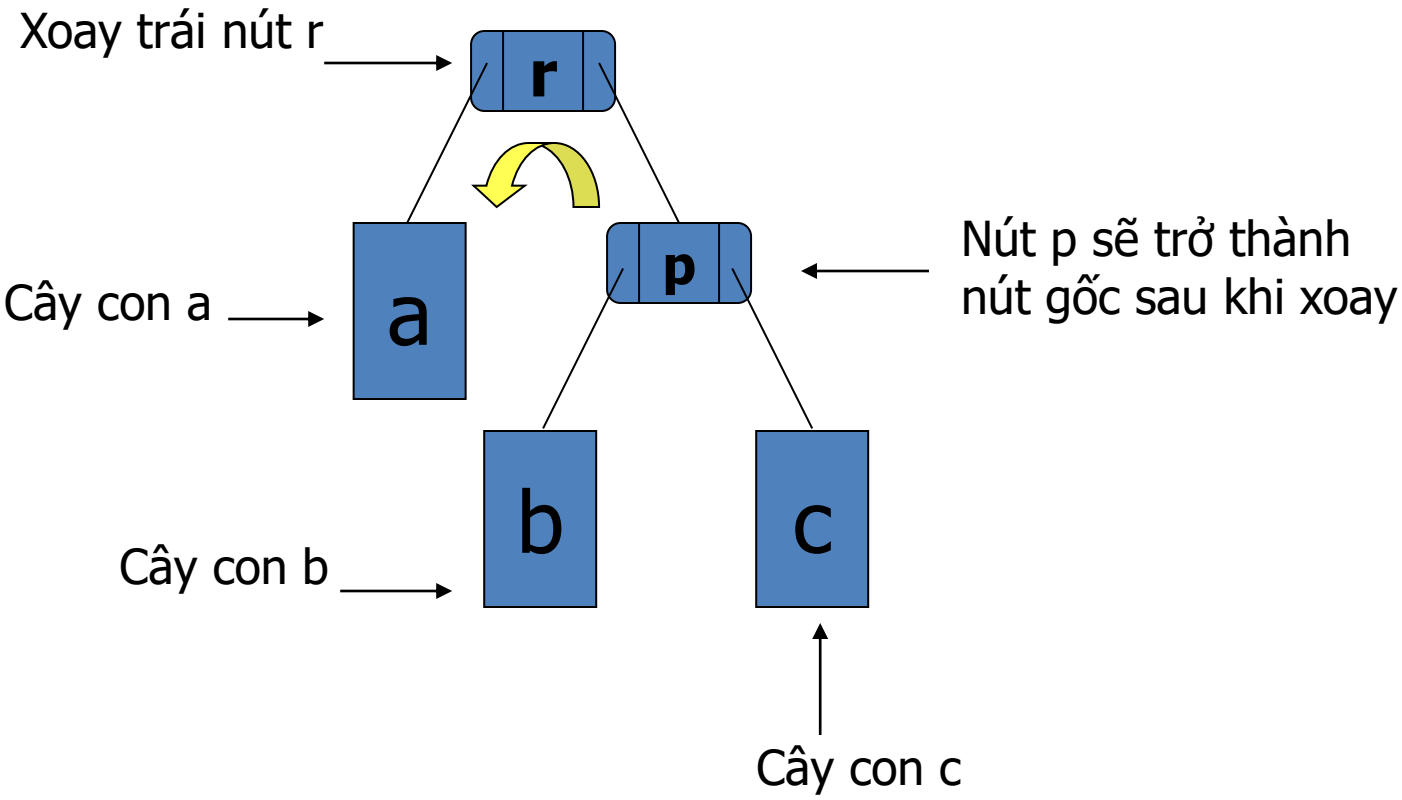
```
16.else { //trường hợp có 2 con chọn nút nn bên con phải
17.    f = p;                  //f để lưu cha của rp
18.    rp = p->right;          // rp bắt đầu từ p->right
19.    while ( rp->left != NULL)
20.    {
21.           f = rp;          // lưu cha của rp
22.           rp = rp->left;     //rp qua bên trái
23.    } //kết thúc khi rp là nút có nút con trái là null
24.    p->info = rp->info;      //đổi giá trị của p và rp
25.    if ( f == p)             // nếu cha của rp là p
26.           f->right = rp->right;
27.    else                     // f != p
28.           f->left = rp->right;
29.    p = rp;      //p trỏ đến phần tử thế mạng rp}
30.delete   p;                  // xoá nút p
31.return  TRUE;
32.}
```

Mở rộng BST

- Quá trình cập nhật cây nhị phân tìm kiếm thường làm cây mất cân bằng
- Thao tác:
  - Xoay trái RotateLeft
  - Xoay phải RotateRight

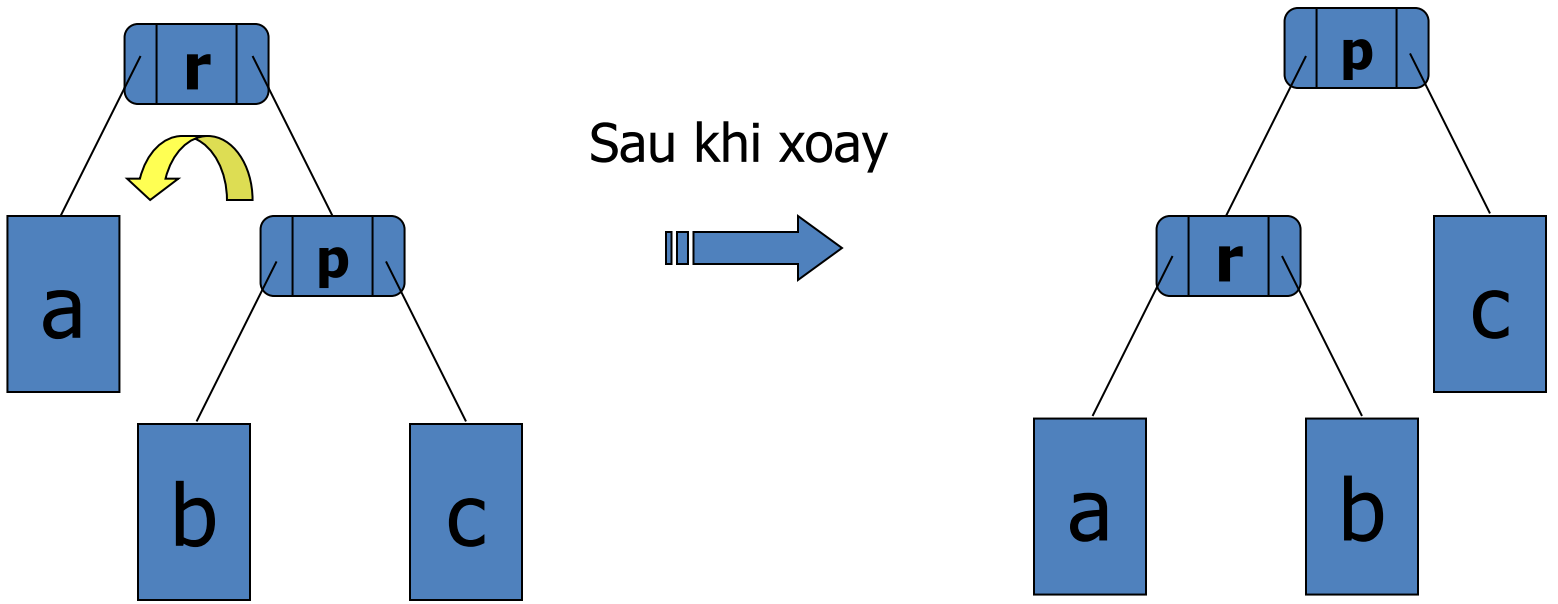
Mở rộng BST

- RotateLeft



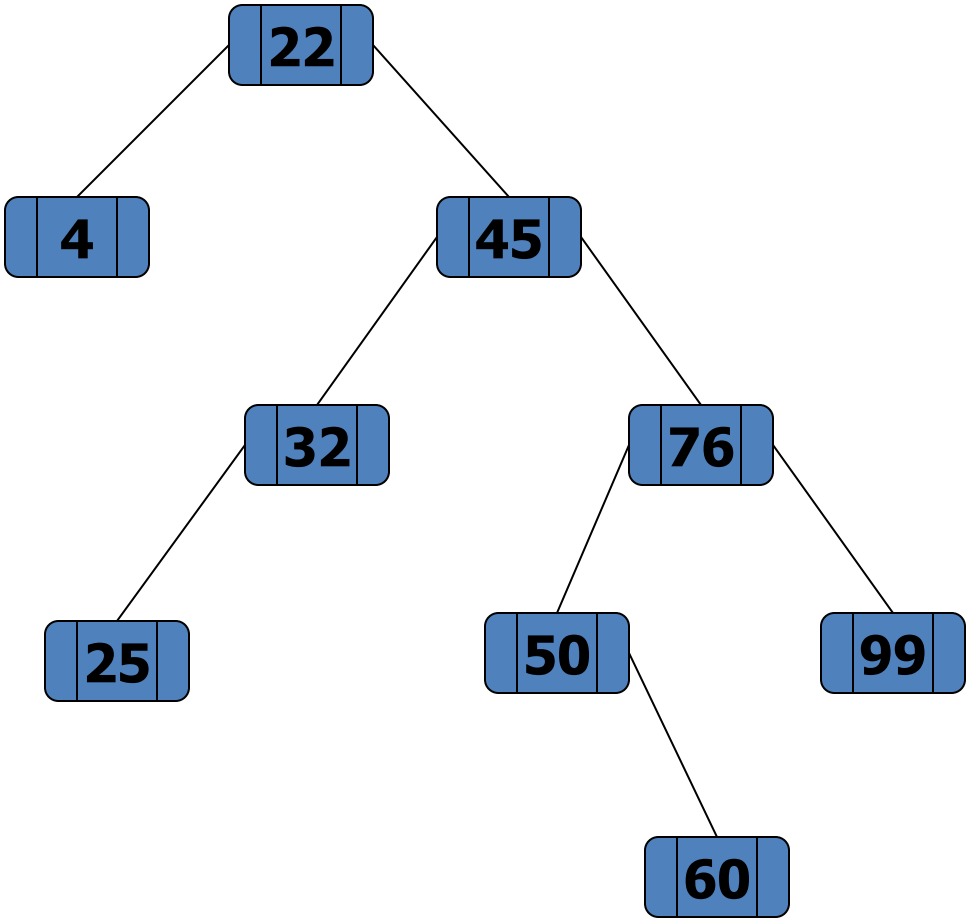
Mở rộng BST

- RotateLeft



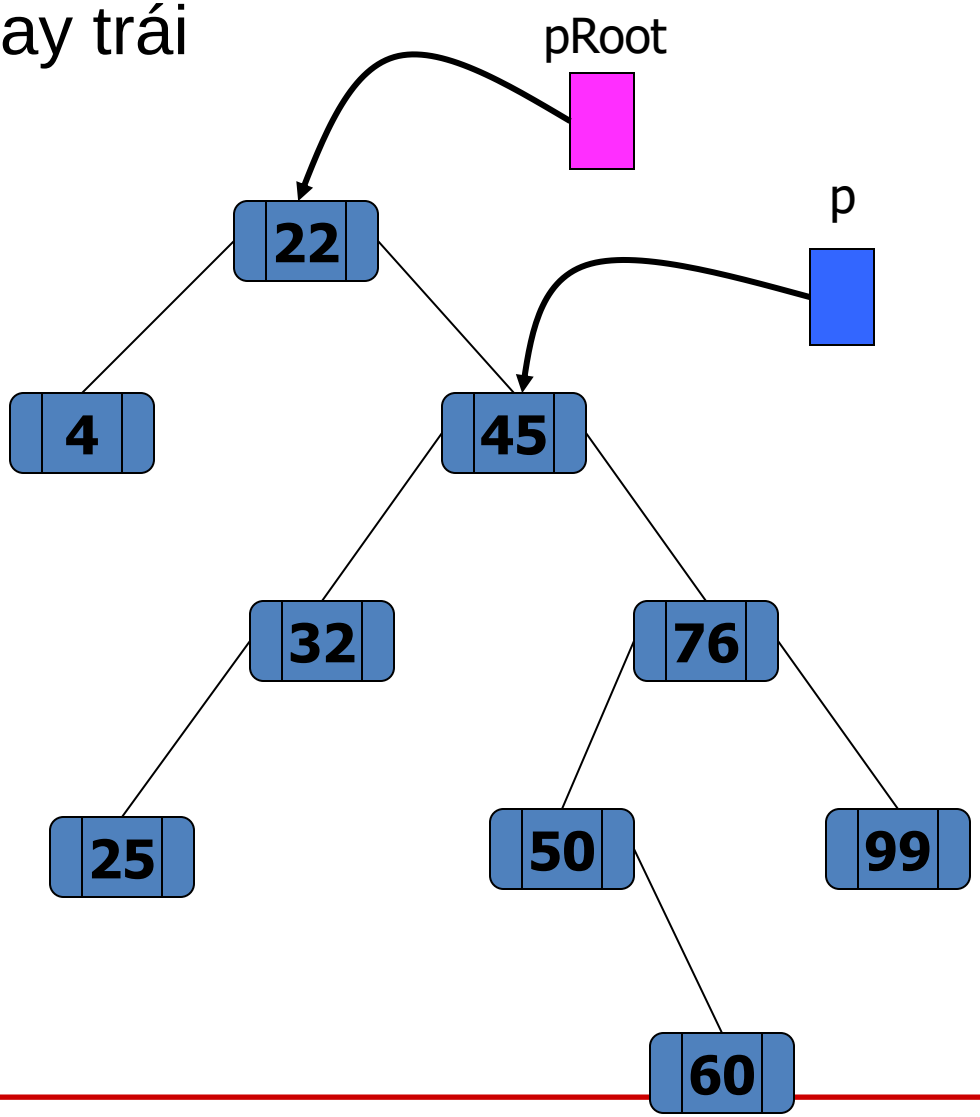
Mở rộng BST

- Minh họa xoay trái



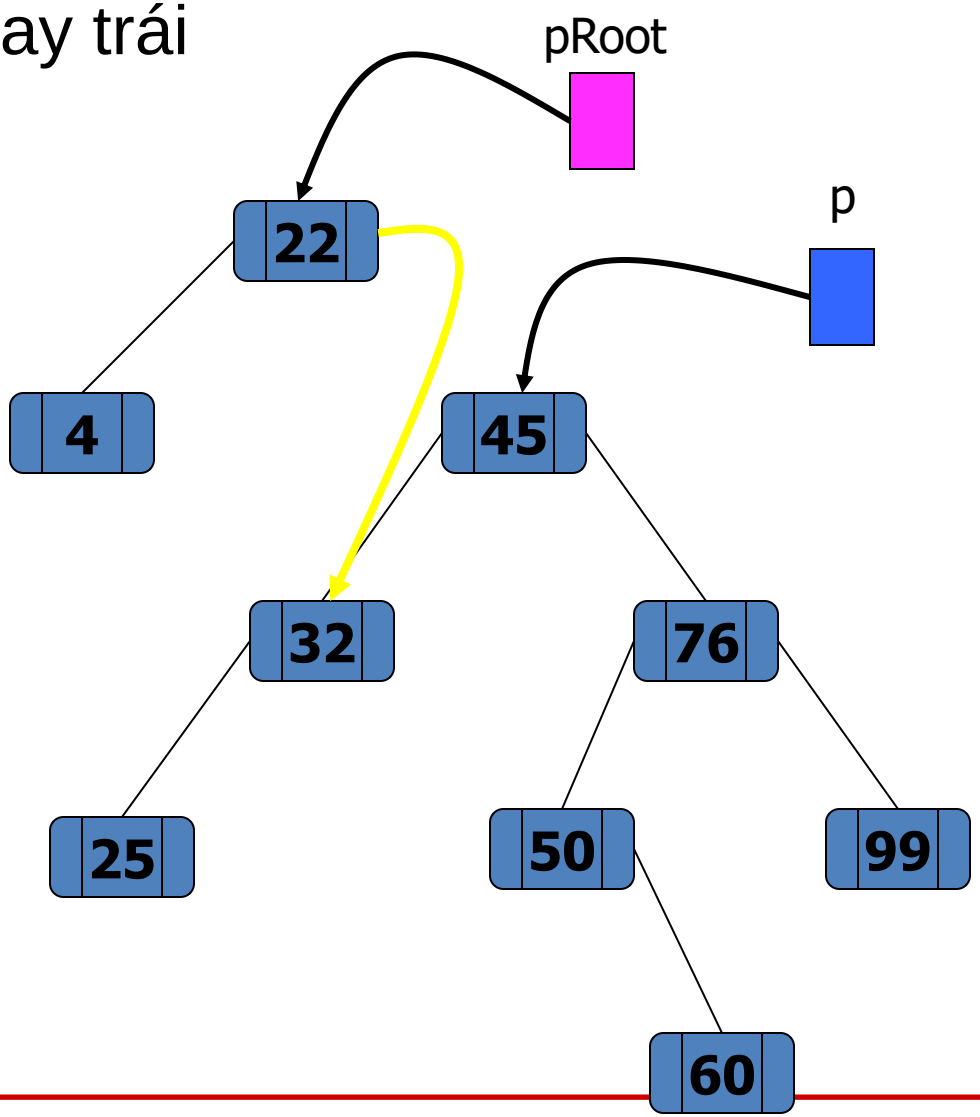
Mở rộng BST

- Minh họa xoay trái



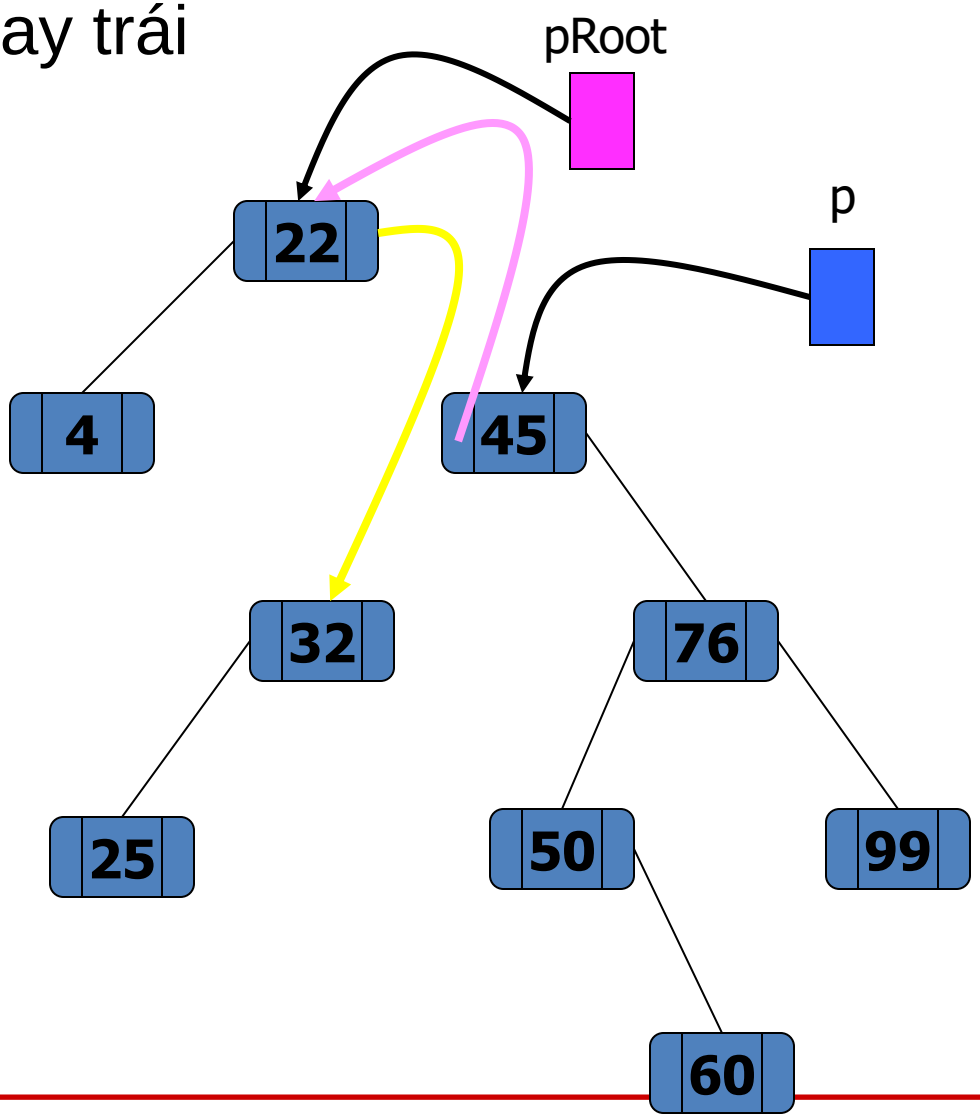
Mở rộng BST

- Minh họa xoay trái



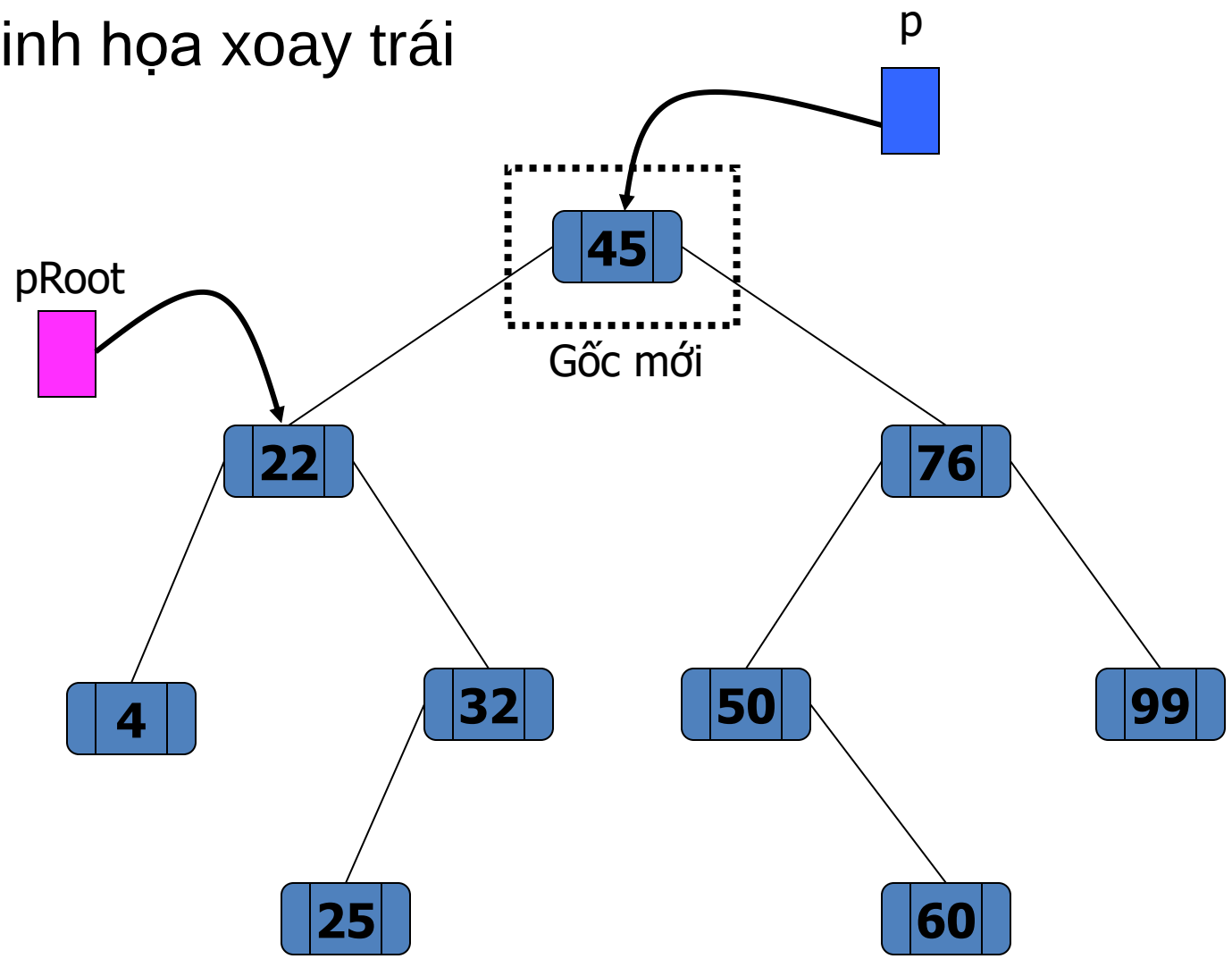
Mở rộng BST

- Minh họa xoay trái



Mở rộng BST

- Minh họa xoay trái



Mở rộng BST

- Thủ tục RotateLeft xoay nút root qua trái và trả về địa chỉ của nút gốc mới (thay cho root).

- NodePtr **RotateLeft**(NodePtr root)

  Nếu pRoot khác rỗng & có cây con phải

  p = root->right

  root->right = p->left

  p->left = root

  return p

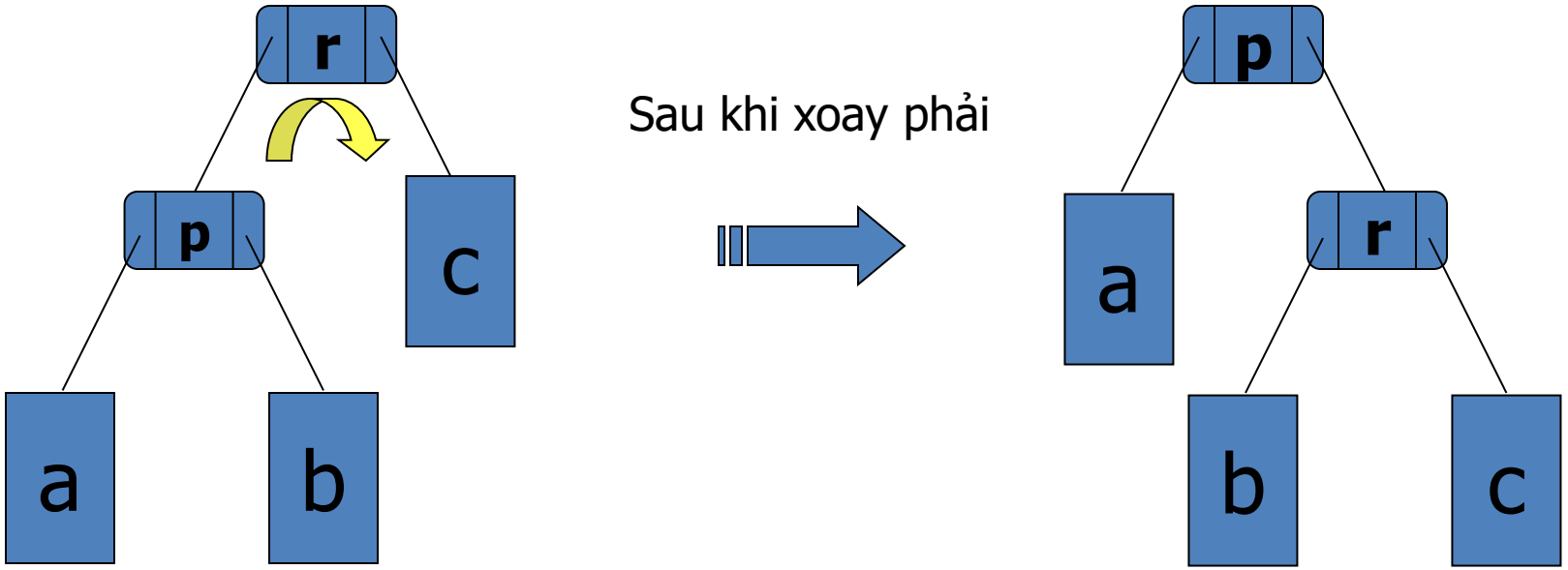
  return NULL

Mở rộng BST

- Sử dụng thủ tục RotateLeft
  - Xoay trái toàn cây nhị phân: trả về con trỏ là nút gốc mới của cây
    - pTree = RotateLeft(pTree)
  - Xoay trái nhánh cây con bên trái của p: trả về con trỏ đến nút gốc mới của nhánh này
    - p->left = RotateLeft(p->left)
  - Xoay trái nhánh cây con bên phải của p:
    - P->right = RotateLeft(p->right)
- Lưu ý: phải cập nhật link từ nút cha đến nút gốc mới
  - Ví dụ xoay nút p, trong đó p trỏ đến nút con trái của f
    - p = RotateLeft(p) // xoay trái nút p
    - f->left = p // cập nhật lại link từ nút cha đến nút gốc mới

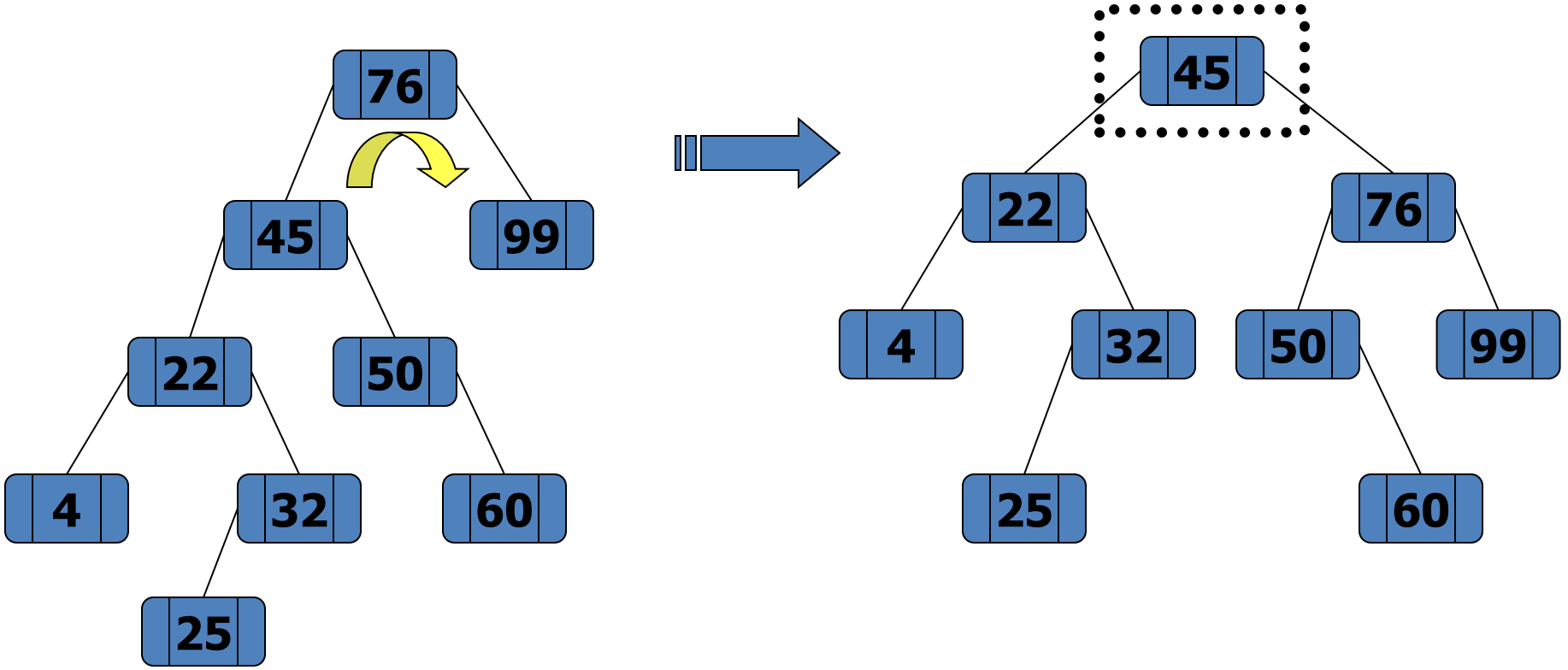
Mở rộng BST

- RotateRight



Mở rộng BST

- Minh họa xoay phải



Mở rộng BST

- Thủ tục **RotateRight** xoay nút root qua phải và trả về địa chỉ của nút gốc mới (thay cho root).

- NodePtr **RotateRight**(NodePtr root)

```
    Nếu pRoot khác rỗng & có cây con trái
        p = root->left
        root->left = p->right
        p->right = root
        return p
return NULL
```

Mở rộng BST

- Sử dụng thủ tục **RotateRight**
  - Xoay phải toàn cây nhị phân: trả về con trỏ là nút gốc mới của cây
    - pTree = RotateRight(pTree)
  - Xoay phải nhánh cây con bên trái của p: trả về con trỏ đến nút gốc mới của nhánh này
    - p->left = RotateRight(p->left)
  - Xoay phải nhánh cây con bên phải của p:
    - P->right = RotateRight(p->right)
- Lưu ý: phải cập nhật liên kết từ nút cha đến nút gốc mới

Nội dung

6.1 Các khái niệm về cây

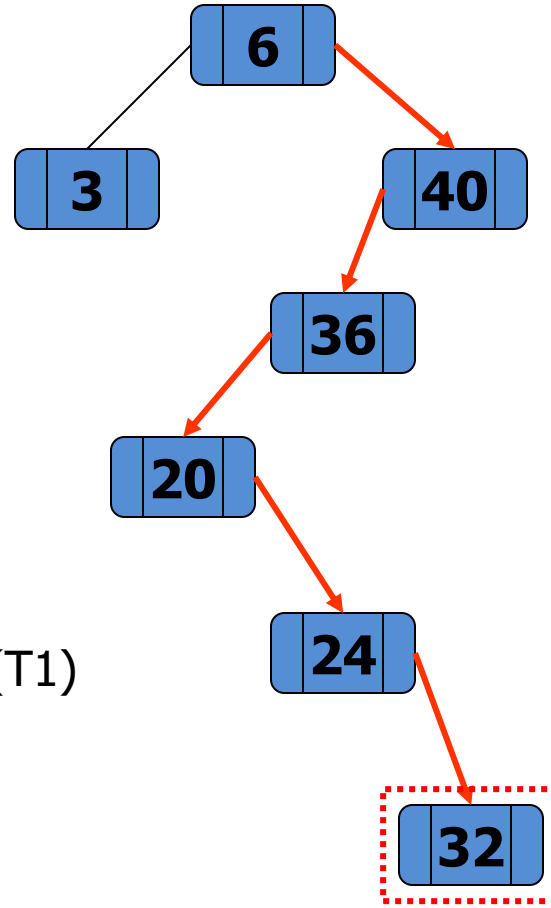
6.2 Cây nhị phân

6.3 Cây nhị phân tìm kiếm

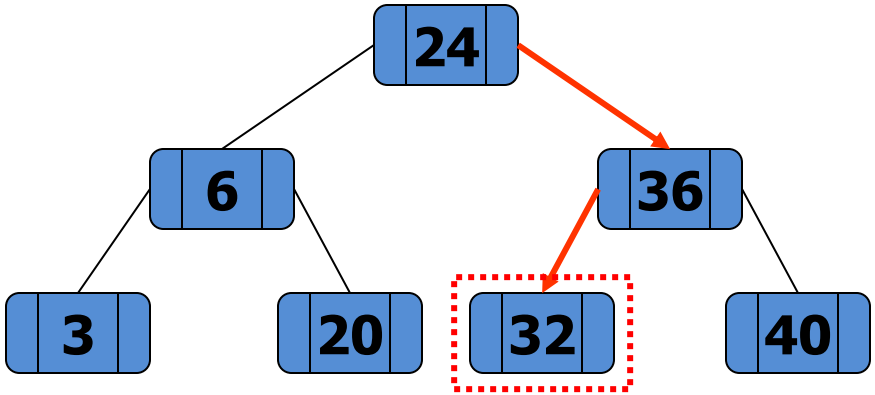
6.4 Cây nhị phân tìm kiếm cân bằng AVL

6.4 Cây AVL

- Đánh giá việc tìm kiếm



(T1)



(T2)

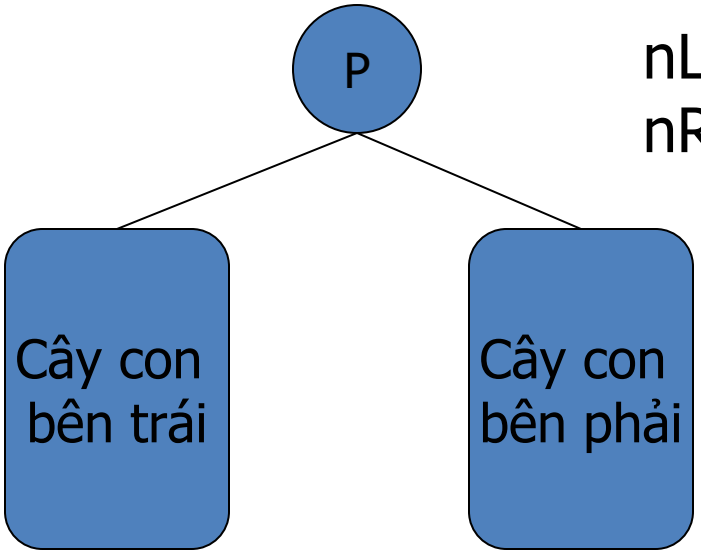
Tìm giá trị 32

6.4 Cây AVL

- Độ phức tạp của thao tác trên BST: chiều dài đường dẫn từ gốc đến nút thao tác
- Trong cây cân bằng tốt, chiều dài của đường đi dài nhất là logn
  - 1 triệu entry $\Rightarrow$ đường dẫn dài nhất là log2 1.048.576 = 20
- Trong trường hợp cây "cao", ko cân bằng, độ phức tạp là O(n)
  - Mỗi phần tử thêm vào theo thứ tự đã sắp
- Sự cân bằng cây là rất quan trọng

6.4 Cây AVL

- Cây NP hoàn toàn cân bằng
  - Là cây nhị phân
  - Tại mỗi nút: số nút trên nhánh trái và nhánh phải chênh lệch ko quá 1 nút!



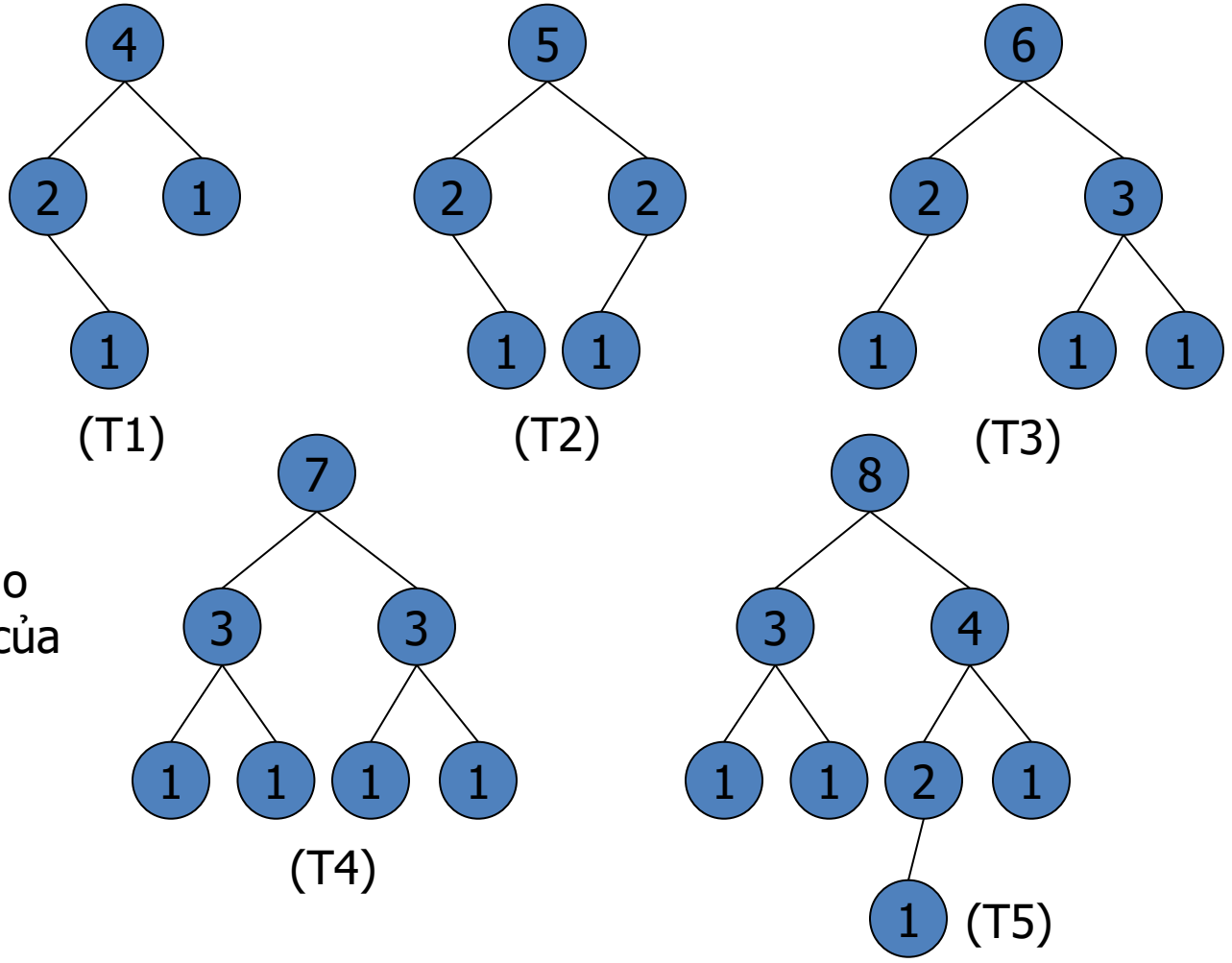
nL(p): số nút cây con trái nút p
nR(p): số nút cây con phải nút p

6.4 Cây AVL

- Trong cây NP hoàn toàn cân bằng
  - Có 3 trường hợp có thể có của một nút (p)
    - nL(p) = nR(p)
    - nL(p) = nR(p)+1
    - nR(p) = nL(p) +1
  - Chiều dài đường đi lớn nhất trong cây NP hoàn toàn cân bằng với n nút là log(n)
  - Quá trình thêm hay xóa một nút dễ làm mất trạng thái cân bằng

6.4 Cây AVL

- Minh họa



Nhãn của nút p cho biết số lượng nút của cây có gốc là p

6.4 Cây AVL

- Cây nhị phân tìm kiếm hoàn toàn cân bằng
  - Tối ưu nhất cho thao tác trên cây
  - Tốc độ tìm kiếm tỉ lệ với O(logn) với n là số nút
  - Thường bị mất cân bằng
    - Do thêm hay xoá
    - Chi phí để cân bằng rất lớn do thao tác trên toàn bộ cây
- Thực tế ít sử dụng cây NPTK hoàn toàn cân bằng!
- Xây dựng cây NPTK đạt trạng thái cân bằng yếu hơn (giảm chi phí)

6.4 Cây AVL

- Cây AVL: ra đời 1962
  - Do tác giả **A**delson-**V**elskii và **L**andis khám phá
  - Cây nhị phân cân bằng đầu tiên
- Tính chất
  - Là cây nhị phân tìm kiếm
  - Độ cao của cây con trái và cây con phải chênh lệch ko quá 1
  - Các cây con cũng là AVL

6.4 Cây AVL

- Mô tả
  - hl(p) là chiều cao của cây con trái nút p
  - hr(p) là chiều cao của cây con phải của p
  - Các trường hợp có thể xảy ra trên cây AVL
    - Nút p cân bằng hl(p) = hr(p)
    - Lệch về trái: hl(p) > hr(p) (lệch 1 đơn vị)
    - Lệch về phải: hl(p) < hr(p) (lệch 1 đơn vị)
  - Thao tác thêm hay xoá có thể làm AVL mất cân bằng
  - Thao tác cân bằng lại trên AVL xảy ra ở cục bộ

6.4 Cây AVL

- Tóm lại: thêm vào AVL
  - Khi thêm vào cây nếu cây bị mất cân bằng
    - Thực hiện cân bằng lại, có 2 trường hợp
      - Có thể chỉ dùng 1 phép xoay
      - Dùng 2 phép xoay như trường hợp 2
    - Trường hợp cây lệch về bên phải thì tương tự (đối xứng)

6.4 Cây AVL

- Cài đặt cây AVL:
  - Bổ sung thêm trường bf cho cấu trúc cây BST

```
typedef struct node
{
        DataType        info;
        int             bf;
        node *          left;
        node *          right;

} NODE;
typedef  NODE *    NodePtr;
NodePtr            pAVLTree;
```

Chỉ số cân bằng balance factor

Trỏ nút gốc của cây AVL

Cây AVL

- Các thao tác trên cây AVL tương tự như BST
- Khác biệt khi thêm/xoá sẽ làm mất cân bằng
  - Ảnh hưởng đến chỉ số cân bằng của nhánh cây liên quan
  - Sử dụng thao tác xoay phải, trái để cân bằng

Sinh viên tự tìm hiểu &
đọc thêm phần AVL

Bài tập

Bài 1: Cho cây nhị phân tìm kiếm có nút gốc được trỏ bởi pTree, trường info của các nút chứa giá trị nguyên.

- Cài đặt cấu trúc dữ liệu liên kết cho cây nhị phân tìm kiếm
- Cài đặt các thao tác xây dựng cây: Init, IsEmpty, CreateNode
- Cài đặt thao tác cập nhật: Insert, Remove, ClearTree
- Xuất danh sách tăng dần và giảm dần
- Kiểm tra xem cây có phải là cây nhị phân đúng
- Kiểm tra xem cây có phải là cây nhị phân đầy đủ
- Xác định nút cha của nút chứa khoá x
- Đếm số nút lá, nút giữa, kích thước của cây
- Xác định độ sâu/chiều cao của cây
- Tìm giá trị nhỏ nhất/lớn nhất trên cây
- Tính tổng các giá trị trên cây

Bài tập

Bài 2:

Viết chương trình hoàn chỉnh thực hiện các thao tác trên cây nhị phân tìm kiếm

Tài liệu tham khảo

[1]. Giáo trình Cấu trúc dữ liệu và giải thuật – Lê Văn Vinh, NXB Đại học quốc gia TP HCM, 2013

[2]. Cấu trúc dữ liệu & thuật toán, Đỗ Xuân Lôi, NXB Đại học quốc gia Hà Nội, 2010.

[3]. Trần Thông Quế, *Cấu trúc dữ liệu và thuật toán (phân tích và cài đặt trên C/C++)*, NXB Thông tin và truyền thông, 2018

[4]. Robert Sedgewick, *Cẩm nang thuật toán*, NXB Khoa học kỹ thuật, 2004 .

[5]. PGS.TS Hoàng Nghĩa Tý, *Cấu trúc dữ liệu và thuật toán*, NXB xây dựng, 2014