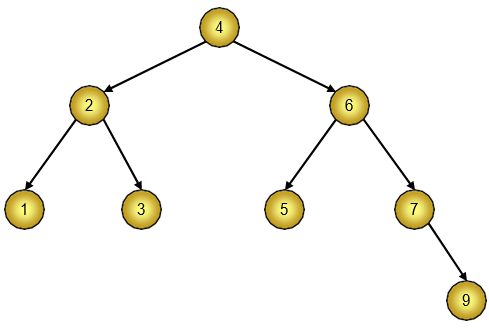
**9.4. CÂY NHỊ PHÂN TÌM KIẾM (BINARY SEARCH TREE - BST)**

Cho n khoá k1, k2, …, kn, trên các khoá có quan hệ thứ tự toàn phần. Cây nhị phân tìm kiếm ứng với dãy khoá đó là một cây nhị phân mà mỗi nút chứa giá trị một khoá trong n khoá đã cho, hai giá trị chứa trong hai nút bất kỳ là khác nhau. Đối với mọi nút trên cây, tính chất sau luôn được thoả mãn:

* Mọi khoá nằm trong cây con trái của nút đó đều nhỏ hơn khoá ứng với nút đó.
* Mọi khoá nằm trong cây con phải của nút đó đều lớn hơn khoá ứng với nút đó



**Cây nhị phân tìm kiếm**

Thuật toán tìm kiếm trên cây có thể mô tả chung như sau:

Trước hết, khoá tìm kiếm X được so sánh với khoá ở gốc cây, và 4 tình huống có thể xảy ra: Không có gốc (cây rỗng): X không có trên cây, phép tìm kiếm thất bại

X trùng với khoá ở gốc: Phép tìm kiếm thành công

X nhỏ hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con trái của gốc với cách làm tương tự

X lớn hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con phải của gốc với cách làm tương tự

Giả sử cấu trúc một nút của cây được mô tả như sau:

type

PNode = ^TNode; {Con trỏ chứa liên kết tới một nút}

TNode = record {Cấu trúc nút}

Info: TKey; {Trường chứa khoá}

Left, Right: PNode; {con trỏ tới nút con trái và phải, trỏ tới nil nếu không có nút con trái (phải)}

end;

Gốc của cây được lưu trong con trỏ Root. Cây rỗng thì Root = nil

Thuật toán tìm kiếm trên cây nhị phân tìm kiếm có thể viết như sau:

{Hàm tìm kiếm trên BST, nó trả về nút chứa khoá tìm kiếm X nếu tìm thấy, trả về nil nếu không tìm thấy}

function BSTSearch(X: TKey): PNode; var

p: PNode;

begin

p := Root; {Bắt đầu với nút gốc}

while p  nil do

if X = p^.Info then Break;

else

if X < p^.Info then p := p^.Left else p := p^.Right;

BSTSearch := p;

end;

Thuật toán dựng cây nhị phân tìm kiếm từ dãy khoá k1, k2, …, kn cũng được làm gần giống quá trình tìm kiếm. Ta chèn lần lượt các khoá vào cây, trước khi chèn, ta tìm xem khoá đó đã có trong cây hay chưa, nếu đã có rồi thì bỏ qua, nếu nó chưa có thì ta thêm nút mới chứa khoá cần chèn và nối nút đó vào cây nhị phân tìm kiếm.

{Thủ tục chèn khoá X vào BST}

procedure BSTInsert(X);

var

p, q: PNode;

begin

q := nil;

p := Root; {Bắt đầu với p = nút gốc; q là con trỏ chạy đuổi theo sau}

while p  nil do

begin

q := p;

if X = p^.Info then Break;

else {X  p^.Info thì cho p chạy sang nút con, q^ luôn giữ vai trò là cha của p^}

if X < p^.Info then p := p^.Left

else p := p^.Right;

end;

if p = nil then {Khoá X chưa có trong BST}

begin

New(p); {Tạo nút mới}

p^.Info := X; {Đưa giá trị X vào nút mới tạo ra}

p^.Left := nil;

p^.Right := nil; {Nút mới khi chèn vào BST sẽ trở thành nút lá}

if Root = nil then Root := NewNode {BST đang rỗng, đặt Root là nút mới tạo}

else {Móc NewNode^ vào nút cha q^}

if X < q^.Info then q^.Left := NewNode

else q^.Right := NewNode;

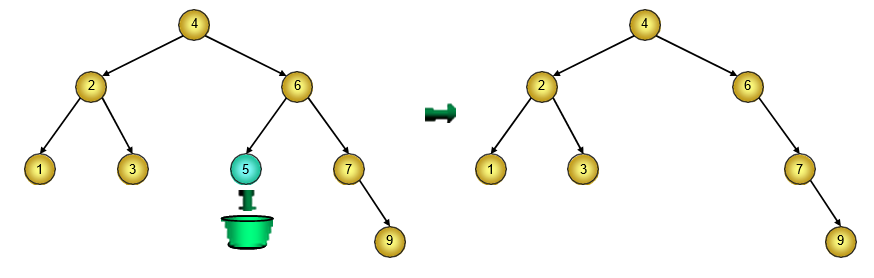
end;

end;

Phép loại bỏ trên cây nhị phân tìm kiếm không đơn giản như phép bổ sung hay phép tìm kiếm.

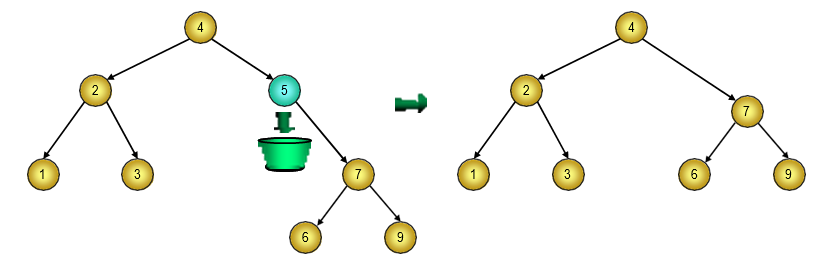
Muốn xoá một giá trị trong cây nhị phân tìm kiếm (Tức là dựng lại cây mới chứa tất cả những giá trị còn lại), trước hết ta tìm xem giá trị cần xoá nằm ở nút D nào, có ba khả năng xảy ra:

* Nút D là nút lá, trường hợp này ta chỉ việc đem mối nối cũ trỏ tới nút D (từ nút cha của D) thay bởi nil, và giải phóng bộ nhớ cấp cho nút D (Hình 38).



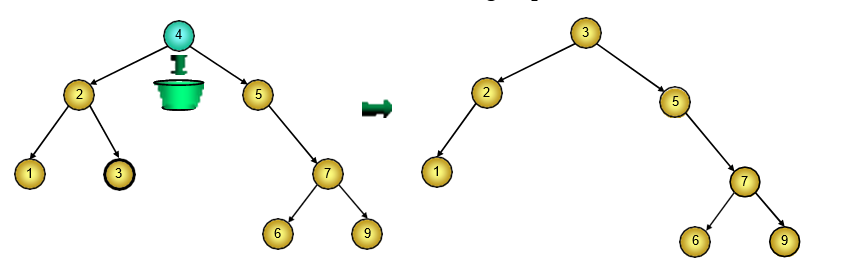
**Xóa nút lá ở cây BST**

* Nút D chỉ có một nhánh con, khi đó ta đem nút gốc của nhánh con đó thế vào chỗ nút D, tức là chỉnh lại mối nối: Từ nút cha của nút D không nối tới nút D nữa mà nối tới nhánh con duy nhất của nút D. Cuối cùng, ta giải phóng bộ nhớ đã cấp cho nút D



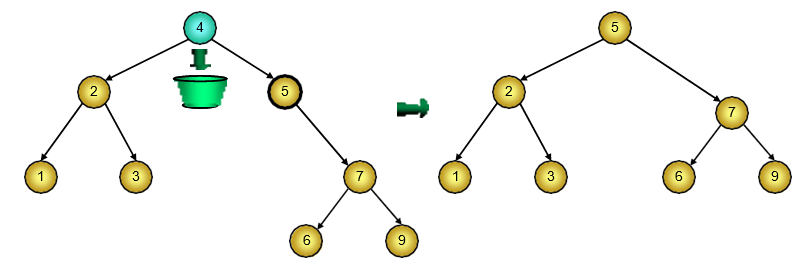
**Xóa nút chỉ có một nhánh con trên cây BST**

* Nút D có cả hai nhánh con trái và phải, khi đó có hai cách làm đều hợp lý cả:
* Hoặc tìm nút chứa khoá lớn nhất trong cây con trái, đưa giá trị chứa trong đó sang nút D, rồi xoá nút này. Do tính chất của cây BST, nút chứa khoá lớn nhất trong cây con trái chính là nút cực phải của cây con trái nên nó không thể có hai con được, việc xoá đưa về hai trường hợp trên (Hình 40)



**Xóa nút có cả hai nhánh con trên cây BST thay bằng nút cực phải của cây con trái**

o Hoặc tìm nút chứa khoá nhỏ nhất trong cây con phải, đưa giá trị chứa trong đó sang nút D, rồi xoá nút này. Do tính chất của cây BST, nút chứa khoá nhỏ nhất trong cây con phải chính là nút cực trái của cây con phải nên nó không thể có hai con được, việc xoá đưa về hai trường hợp trên.



**Xóa nút có cả hai nhánh con trên cây BST thay bằng nút cực trái của cây con phải**

{Thủ tục xoá khoá X khỏi BST}

procedure BSTDelete(X: TKey);

var

p, q, Node, Child: PNode;

begin

p := Root;

q := nil; {Về sau, khi p trỏ sang nút khác, ta luôn giữ cho q^ luôn là cha của p^}

while p  nil do {Tìm xem trong cây có khoá X không?}

begin

if p^.Info = X then Break; {Tìm thấy}

q := p;

if X < p^.Info then p := p^.Left

else p := p^.Right;

end;

if p = nil then Exit; {X không tồn tại trong BST nên không xoá được}

if (p^.Left  nil) and (p^.Right  nil) then {p^ có cả con trái và con phải}

begin

Node := p; {Giữ lại nút chứa khoá X}

q := p;

p := p^.Left; {Chuyển sang nhánh con trái để tìm nút cực phải}

while p^.Right  nil do

begin

q := p;

p := p^.Right;

end;

Node^.Info := p^.Info; {Chuyển giá trị từ nút cực phải trong nhánh con trái lên Node^}

end;

{Nút bị xoá giờ đây là nút p^, nó chỉ có nhiều nhất một con}

{Nếu p^ có một nút con thì đem Child trỏ tới nút con đó, nếu không có thì Child = nil }

if p^.Left  nil then Child := p^.Left

else Child := p^.Right;

if p = Root then Root := Child; {Nút p^ bị xoá là gốc cây}

else {Nút bị xoá p^ không phải gốc cây thì lấy mối nối từ cha của nó là q^ nối thẳng tới Child}

if q^.Left = p then q^.Left := Child

else q^.Right := Child;

Dispose(p);

end;

Trường hợp trung bình, thì các thao tác tìm kiếm, chèn, xoá trên BST có độ phức tạp là O(log2n). Còn trong trường hợp xấu nhất, cây nhị phân tìm kiếm bị suy biến thì các thao tác đó đều có độ phức tạp là O(n), với n là số nút trên cây BST.

Nếu ta mở rộng hơn khái niệm cây nhị phân tìm kiếm như sau: Giá trị lưu trong một nút lớn hơn **hoặc bằng** các giá trị lưu trong cây con trái và nhỏ hơn các giá trị lưu trong cây con phải. Thì chỉ cần sửa đổi thủ tục BSTInsert một chút, khi chèn lần lượt vào cây n giá trị, cây BST sẽ có n nút (có thể có hai nút chứa cùng một giá trị). Khi đó nếu ta duyệt các nút của cây theo kiểu trung thứ tự (inorder traversal), ta sẽ liệt kê được các giá trị lưu trong cây theo thứ tự tăng dần. Phương pháp sắp xếp này người ta gọi là Tree Sort. Độ phức tạp tính toán trung bình của Tree Sort là O(nlog2n).

Phép tìm kiếm trên cây BST sẽ kém hiệu quả nếu như cây bị suy biến, người ta có nhiều cách xoay xở để tránh trường hợp này. Đó là phép quay cây để dựng cây nhị phân cân đối AVL, hay kỹ thuật dựng cây nhị phân tìm kiếm tối ưu. Những kỹ thuật này ta có thể tham khảo trong các tài liệu khác về cấu trúc dữ liệu và giải thuật.