

Chương 4

CÂY

Trong chương này chúng ta sẽ nghiên cứu mô hình đồ thị cây (tree). Cây là một cấu trúc phân cấp trên một tập hợp nào đó các đối tượng. Một ví dụ quen thuộc về cây, đó là cây họ mạc hoặc các loài của cuốn sách cũng là một cây. Cây được sử dụng rộng rãi trong rất nhiều vấn đề khác nhau. Chẳng hạn nó được áp dụng để tìm kiếm thông tin trong các hệ cơ sở dữ liệu, để mô tả cấu trúc cú pháp của các chương trình nguồn khi xây dựng các chương trình dịch. Rất nhiều bài toán mà ta gặp trong các lĩnh vực khác nhau được quy về việc tìm kiếm các phép toán trên cây. Trong chương này chúng ta sẽ trình bày định nghĩa, các khái niệm cơ bản về cây. Chúng ta cũng sẽ xét các phương pháp cài đặt cây và tìm kiếm các phép toán cơ bản trên cây. Sau đó ta nghiên cứu kỹ một số dạng cây đặc biệt đó là cây nhị phân tìm kiếm và cây cân bằng.

1. CÂY VÀ CÁC KHÁI NIỆM LIÊN QUAN

1.1. Định nghĩa cây

Định nghĩa: Một cây là tập hợp hữu hạn các nút trong đó có một nút đặc biệt gọi là gốc (root). Giữa các nút có mối quan hệ phân cấp gọi là "quan hệ cha con".

Định nghĩa 2: Cây được định nghĩa theo như sau

- Một nút là một cây và nút này cũng là gốc của cây.
- Giả sử $T_1, T_2, \dots, T_n$ ($n \geq 1$) là các cây có gốc tương ứng $r_1, r_2, \dots, r_n$. Khi đó cây T với gốc r được hình thành bằng cách cho r trở thành nút cha của các nút $r_1, r_2, \dots, r_n$

1.2. Một số khái niệm cơ bản

- **Bậc của một nút:** là số con của nút đó
- **Bậc của một cây:** là bậc lớn nhất của các nút có trên cây đó (số cây con tối đa của một nút thuộc cây). Cây có bậc n thì gọi là cây n - phân
- **Nút gốc:** là nút có không có nút cha
- **Nút lá:** là nút có bậc bằng 0
- **Nút nhánh:** là nút có bậc khác 0 và không phải là nút gốc

- Mức của một nút

$$Mức(gốc(T_0)) = 1$$

Gọi $T_1, T_2, \dots, T_n$ là các cây con của T_0 .

Khi đó $Mức(T_1) = Mức(T_2) = \dots = Mức(T_n) = Mức(T_0) + 1$

- Chiều cao của cây: là số mức lớn nhất có trên cây đó

- Đường đi: Dãy các đỉnh $n_1, n_2, \dots, n_k$ được gọi là đường đi nếu n_i là cha của n_{i+1} ($1 \leq i \leq k-1$)

- Độ dài của đường đi: là số nút trên đường đi -1

- **Cây được sắp thứ tự**: Trong một cây, nếu các cây con của mỗi đỉnh được sắp theo một thứ nhất định, thì cây được gọi là cây được sắp (cây có thứ tự). Chẳng hạn, hình minh họa hai cây được sắp khác nhau



Hình 4.1. Hai cây được sắp khác nhau

- **Rừng**: là tập hợp hữu hạn các cây phân biệt



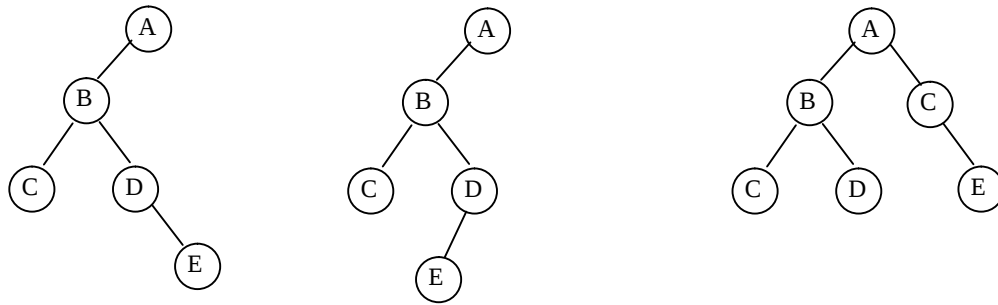
Hình 4.2. Rừng gồm ba cây

2. CÂY NHỊ PHÂN

2.1. Định nghĩa

Cây nhị phân là cây mà mỗi nút có tối đa hai cây con. Đối với cây con của một nút nào đó ta cũng phân biệt cây con trái và cây con phải.

Như vậy cây nhị phân là cây có thứ tự.



Hình 4.3. Một số cây nhị phân

2.2. Tính chất

Đối với cây nhị phân cần chú ý tới một số tính chất sau

1. Số lượng nút ở tầng các nút có mức i trên cây nhị phân là 2^{i-1} ($i \geq 1$)
2. Số lượng nút ở tầng trên một cây nhị phân có chiều cao h là $2^h - 1$ ($h \geq 1$)

Chứng minh

+ *Tính chất 1 sẽ được chứng minh bằng quy nạp*

Bước cơ sở: với $i=1$, cây nhị phân có tầng đầu $1=2^0$ nút. Vậy mệnh đề đúng với $i=1$

Bước quy nạp: Giả sử kết quả đúng với mức i , nghĩa là ở mức này cây nhị phân có tầng đầu 2^{i-1} nút, ta chứng minh mệnh đề đúng với mức $i+1$.

Theo định nghĩa cây nhị phân thì mỗi nút có tầng đầu hai cây con nên mỗi nút ở mức i có tầng đầu hai con. Do đó theo giả thiết quy nạp ta suy ra tầng mức $i+1$ ta có

$$2^{i-1} \times 2 = 2^i \text{ nút.}$$

+ *Tính chất 2 được chứng minh như sau:*

Ta đã biết rằng chiều cao của cây là số mức lớn nhất có trên cây đó. Theo i) ta suy ra số nút tầng đầu có trên cây nhị phân với chiều cao h là :

$$2^0 + 2^1 + \dots + 2^{h-1} = 2^h - 1.$$

Từ kết quả này có thể suy ra:

Nếu cây nhị phân có n nút thì chiều cao của nó là $h = \lceil \log_2(n + 1) \rceil$

(Ta qui định $\lceil x \rceil$ là số nguyên trên của x

$\lfloor x \rfloor$ là số nguyên dưới của x)

2.3. Biểu diễn cây nhị phân

2.3.1. Lưu trữ kiểu

Phương pháp tự nhiên nhất để biểu diễn cây nhị phân là chỉ ra đỉnh con trái và đỉnh con phải của mỗi đỉnh.

Ta có thể sử dụng mảng để lưu trữ các đỉnh của cây nhị phân. Mỗi đỉnh của cây được biểu diễn bởi vị trí ghi gồm ba trường:

INFOR: mô tả thông tin gắn với mỗi đỉnh

LEFT : chỉ đỉnh con trái

RIGHT: chỉ đỉnh con phải.

Giả sử các đỉnh của cây được đánh số từ 1 đến max. Khi đó cấu trúc dữ liệu biểu diễn cây nhị phân được khai báo như sau:

#define max 100. ; {số tối đa của nút trên cây}

Struct Node

Item infor;

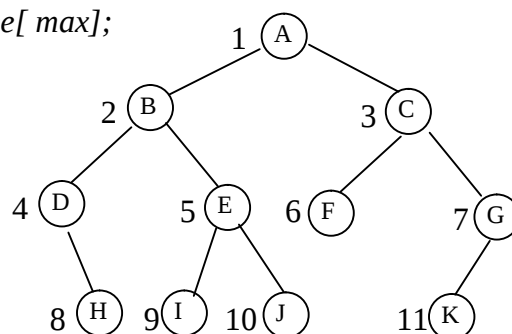
Int left;

Int right;

}

Node Tree[max];

Ví dụ:



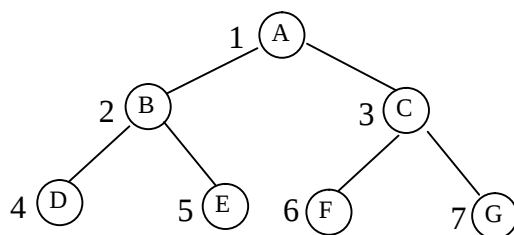
Hình 4.4. Mô tả cây nhị phân

Hình 4.5 minh họa cấu trúc dữ liệu biểu diễn cây nhị phân trong hình 4.4

	infor	left	right
1	A	2	3
2	B	4	5
3	C	6	7
4	D	0	8
5	E	9	10
6	F	0	0
7	G	11	9
8	H	0	0
9	I	0	0
10	J	0	0
11	K	0	0

Hình 4.5. Cấu trúc dữ liệu biểu diễn cây

Nếu có một cây nhị phân hoàn chỉnh đầy đủ, ta có thể dễ dàng đánh số cho các nút trên cây đó theo thứ tự lần lượt từ mức 1 trở lên, hết mức này đến mức khác và từ trái qua phải đi với các nút ở mức tiếp theo. Ví dụ với hình 4.6 có thể đánh số như sau:



Hình 4.6. Cây nhị phân được đánh số

Ta có nhận xét sau: con của nút thứ i là các nút thứ $2i$ và $2i + 1$ hoặc cha của nút thứ j là $\lfloor j/2 \rfloor$.

Như vậy, ta có thể lưu trữ cây này bằng một vectơ V , theo nguyên tắc: nút thứ i của cây được lưu trữ ở $V[i]$. Đó chính là cách lưu trữ kết thúc đối với cây nhị phân. Với

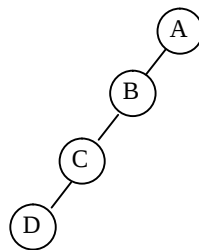
cách lưu trữ này nếu biết được địa chỉ của nút con sẽ tính được địa chỉ nút cha và ngược lại.

Với cây đầy đủ nêu trên thì hình ảnh lưu trữ sẽ như sau

A	B	C	D	E	F	G
v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]

Nhận xét:

Nếu cây nhị phân không đầy đủ thì cách lưu trữ này không thích hợp vì sẽ gây ra lãng phí bộ nhớ do có nhiều phần tử bỏ trống (như với cây con rỗng). Ta hãy xét cây như hình 4.7. Để lưu trữ cây này ta phải dùng mảng gồm 31 phần tử mà chỉ có 5 phần tử khác rỗng, hình ảnh lưu trữ minh họa của cây này như sau



Hình 4.7. Cây nhị phân đầy đủ

	B	∅	C	∅	∅	∅	D	∅	∅	∅	∅	∅	∅	∅	E	∅	...
--	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	-----

(∅: chỉ chỗ trống)

Nếu cây nhị phân luôn biến đổi ngẫu nhiên là có phép bổ sung, loại bỏ các nút thông qua xuyên tác động thì cách lưu trữ này gặp phải một số nhược điểm như tốn thời gian khi phải thực hiện các thao tác này, độ cao của cây phụ thuộc vào kích thước của mảng, ...

2.3.2. Lưu trữ móc nối

Cách lưu trữ này khác với cách thông thường ở điểm của cách lưu trữ kết tiếp động thời phản ánh được dòng tự nhiên của cây.

Trong cách lưu trữ móc nối, mỗi nút thông thường như với một phần tử như có qui cách như sau:

Left	Info	Right
------	------	-------

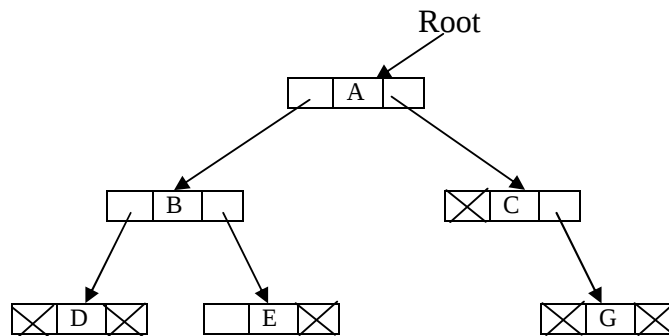
- Trỏ thông tin về thông tin (dữ liệu) của nút
- Trỏ left về con trái, trỏ phải về con phải của nút đó
- Trỏ right về con phải, trỏ trái về con trái của nút đó

Ta có thể khai báo như sau:

typedef Struct Node

```
{
    Int info;
    Struct Node *left, *right;
}tree;
tree *Cay;
```

Ví dụ: cây như phân hình 4.8 có dạng lưu trữ gốc như hình 4.9



Hình 4.9. Cấu trúc dữ liệu biểu diễn cây

Để truy nhập vào các nút trên cây cần có một con trỏ Root, trỏ tới nút gốc của cây.

2.4. Phép duyệt cây nhị phân

Phép xử lý các nút trên cây - mà ta gọi chung là phép thăm các nút một cách hệ thống, sao cho mỗi nút được thăm đúng một lần, gọi là phép duyệt cây. Chúng ta thường duyệt cây nhị phân theo một trong ba thứ tự: duyệt trước, duyệt giữa và duyệt sau, các phép duyệt này được định nghĩa như sau:

2.4.1. Duyệt theo thứ tự trước (preorder traversal)

- Thăm gốc
- Duyệt cây con trái theo thứ tự trước
- Duyệt cây con phải theo thứ tự trước

2.4.2. Duyệt theo thứ tự giữa (inorder traversal)

- Duyệt cây con trái theo thứ tự giữa
- Thăm gốc
- Duyệt cây con phải theo thứ tự giữa

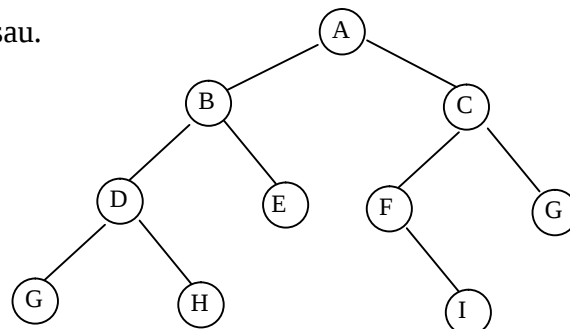
2.4.3. Duyệt theo thứ tự sau (postorder traversal)

- Duyệt cây con trái theo thứ tự sau
- Duyệt cây con phải theo thứ tự sau
- Thăm gốc

Tổng cộng với ba phép duyệt ta có ba thuật toán duyệt cây nhị phân. Sau đây là thuật toán để duyệt cây theo thứ tự trước

```
Void Preorder ( Tree *root);  
  
{  
    if (root != NULL)  
    {  
        printf("\n",root->info);  
        Preorder(root->left);  
        Preorder(root->right);  
    }  
}
```

Một cách thông thường, ta có thể viết để các thuật toán đi qua cây theo thứ tự giữa và theo thứ tự sau.



Hình 4.10

Với cây nh phân hình 4.10, dãy các nút được thăm trong các phép duyệt là:

1. Duyệt theo thứ tự trước

A B D G H E C F I G

2. Duyệt theo thứ tự giữa

G D H B E A F I C G

3. Duyệt theo thứ tự sau

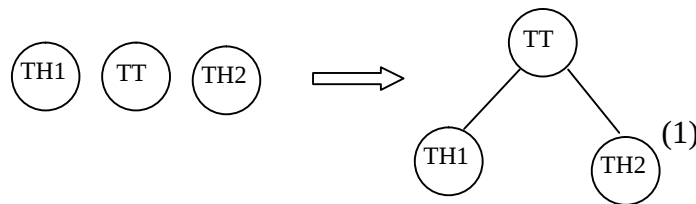
G H D E B I F G C A

2.5. Cây nh phân biểu diễn biểu thức

Cây biểu thức là cây nh phân mà nút gốc và các nút nhánh chứa các toán tử (phép toán) còn các nút lá thì chứa các toán hạng.

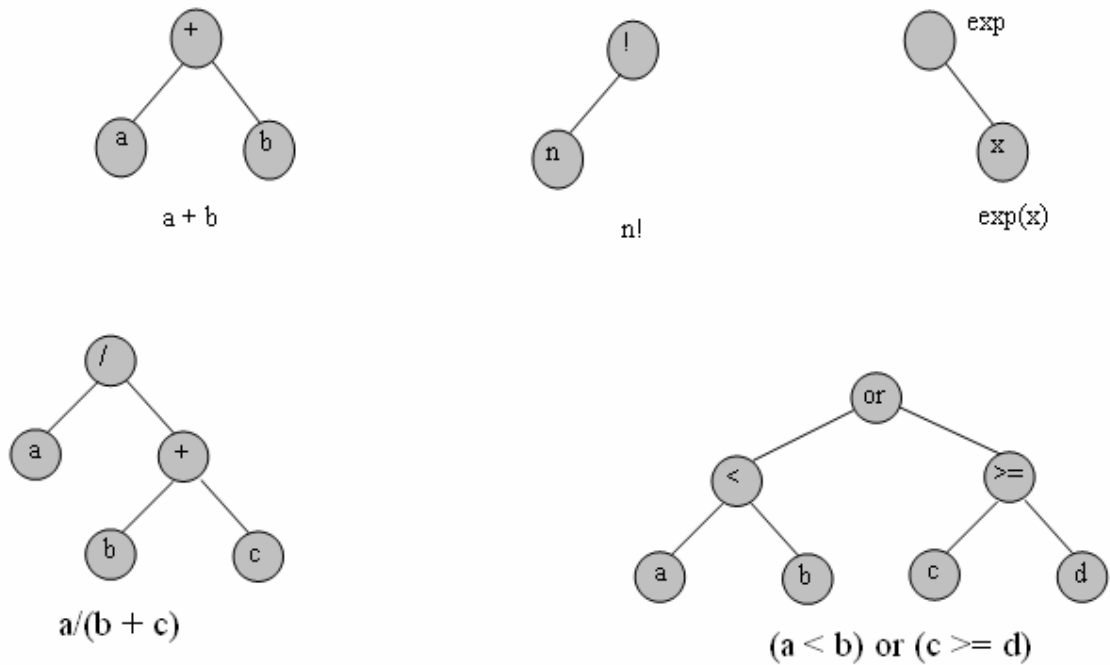
2.5.1. Cách dựng cây biểu thức

Đối với phép toán hai ngôi (chẳng hạn $+$, $-$, $*$, $/$) được biểu diễn bởi cây nh phân mà gốc của nó chứa toán tử, cây con trái biểu diễn toán hạng bên trái, còn cây con bên phải biểu diễn toán hạng bên phải



Hình 4.11. Biểu diễn phép toán hai ngôi

Đối với phép toán một ngôi như "phần trăm" hoặc "lấy giá trị đối", hoặc các hàm chuẩn như $\exp()$ hoặc $\cos()$... thì cây con bên trái rỗng. Còn với các phép toán một toán hạng như phép "lấy đạo hàm" $()'$ hoặc hàm "giai thừa" $()!$ thì cây con bên phải rỗng.



Hình 4.12. Một số cây biểu thức

Nhận xét

Nếu ta duyệt cây biểu thức theo thứ tự trước thì ta được biểu thức Balan đúng tiền tố (prefix). Nếu duyệt cây nhúng phân theo thứ tự sau thì ta có biểu thức Balan đúng hậu tố (postfix); còn theo thứ tự giữa thì ta nhận được cách viết thông thường của biểu thức (đúng trung tố).

2.5.2. Ví dụ

Đề minh cho nhận xét này ta lấy ví dụ sau:

Cho biểu thức $P = a*(b - c) + d/e$

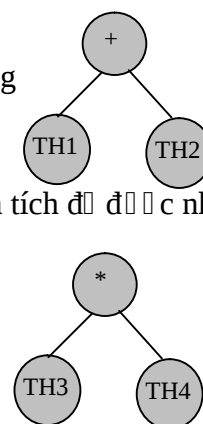
1. Hãy dùng cây biểu thức biểu diễn biểu thức trên
2. Đưa ra biểu thức ở đúng tiền tố và hậu tố

Giải

1. Ta có $TH1 = a*(b - c)$ } suy ra cây biểu thức có dạng
 $TH2 = d/e$ }

Xét $TH1 = a*(b - c)$, toán hạng chứa a ở dạng cuối bên ta phải phân tích để được như (1)

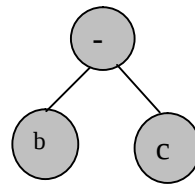
$TH3 = a$ } cây biểu thức của toán hạng này là



$$TH4 = b - c$$

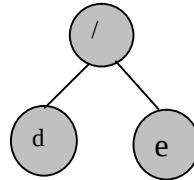
Toán hạng TH4 đã dùng các biến

nên ta có ngay cây biểu thức



Cũng tương tự như vậy đối với

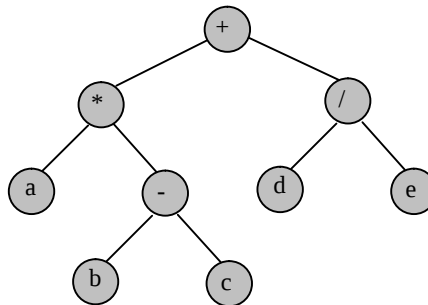
toán hạng TH2, cây biểu thức



tương ứng với toán hạng này là

Hình 4.13

Tổng hợp cây biểu thức của các toán hạng ta được cây biểu thức sau



Hình 4.14. Cây biểu thức

2. Bây giờ ta duyệt cây biểu thức ở hình 4.14

Duyệt theo thứ tự trước : $+ * a - b c / d e$

Duyệt theo thứ tự sau: $a b c - * d e / +$

3. CÂY TÍNH QUÁT

3.1. Biểu diễn cây tính quát

Đối với cây tính quát cấp m nào đó có thể sử dụng cách biểu diễn gốc nút tính toán để chia cây nhánh phân. Như vậy ứng với mỗi nút ta phải dành ra m trường để lưu trữ các con của nút đó và như vậy sẽ có m nút không sử dụng nữa: nếu cây có n nút sẽ có $n(m-1) + 1$ "m nút không" trong số m.n nút nút.

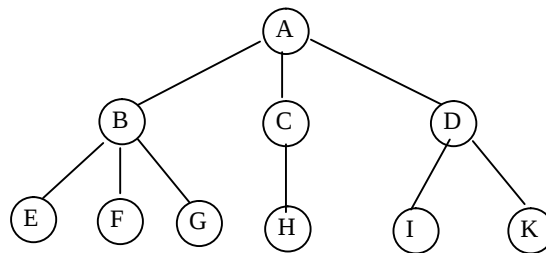
Nếu tùy theo số con của từng nút mà định ra mức độ nghĩa là dùng nút có kích thước của biên độ thì số tiết kiệm không gian như này sẽ phải trả giá bằng những phức tạp của quá trình xử lý trên cây.

Để khắc phục các nhược điểm trên là dùng cách biểu diễn cây như phân đồ biểu diễn cây tổng quát.

Ta có thể biến đổi một cây bất kỳ thành một cây như phân theo quy tắc sau

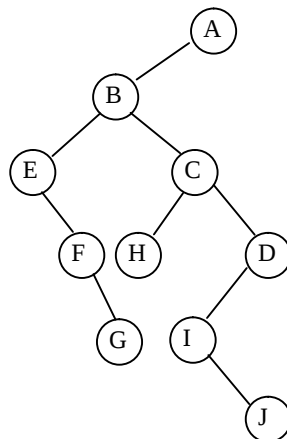
- Giữ lại nút con trái nhất làm nút con trái
- Các nút con còn lại chuyển thành các nút con phải
- Như vậy, trong cây như phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu. Khi đó cây như phân này được gọi là cây như phân tổng quát.

Ta có thể xem ví dụ dưới đây để thấy rõ quy trình. Giả sử có cây tổng quát như hình vẽ dưới đây



Hình 4.15. Cây tổng quát

Cây như phân tổng quát sẽ như sau



Hình 4.16. Cây như phân tổng quát

3.2. PHÉP DUYỆT CÂY TỪNG QUÁT

Phép duyệt cây từng quát cũng được đưa ra từng bước như đối với cây nhị phân. Tuy nhiên có một điều cần phải xem xét thêm, khi định nghĩa phép duyệt, đó là:

1. Sơ nhật quán với tất cả các nút được thăm giữa phép duyệt cây từng quát và phép duyệt cây nhị phân từng bước được gọi là nó

2. Sơ nhật quán giữa định nghĩa phép duyệt cây từng quát với định nghĩa phép duyệt cây nhị phân. Vì cây nhị phân cũng có thể coi là cây từng quát và ta có thể áp dụng định nghĩa phép duyệt cây từng quát cho cây nhị phân.

Ta có thể xây dựng được định nghĩa của phép duyệt cây từng quát T như sau:

Duyệt theo thứ tự trước

1. Nếu T rỗng thì không làm gì

2. Nếu T khác rỗng thì

Thăm gốc của T

Duyệt các cây con theo thứ tự trước của gốc cây T theo thứ tự trước

Duyệt các cây con còn lại $T_2, T_3, \dots, T_n$ của gốc T theo thứ tự trước

Duyệt theo thứ tự giữa

1. Nếu T rỗng thì không làm gì

2. Nếu T khác rỗng thì

Duyệt các cây con theo thứ tự giữa của gốc cây T theo thứ tự giữa

Thăm gốc của T

Duyệt các cây con còn lại $T_2, T_3, \dots, T_n$ của gốc T theo thứ tự giữa

Duyệt theo thứ tự sau

1. Nếu T rỗng thì không làm gì

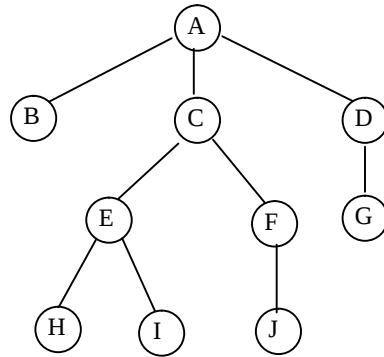
2. Nếu T khác rỗng thì

Duyệt các cây con theo thứ tự sau của gốc của cây T theo thứ tự sau

Duyệt các cây con còn lại $T_2, T_3, \dots, T_n$ của gốc T theo thứ tự sau

Thăm gốc của T

Ví dụ với cây ở hình vẽ 4.17



Hình 4.17

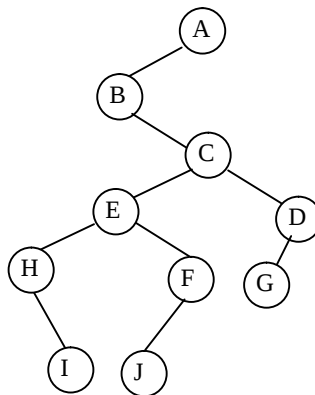
Thì dãy tên các nút được thăm sẽ là

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa: B A H E I C J F G D

Thứ tự sau: B H I E J F C G D A

Bây giờ ta dùng cây nhúng phân tán để vẽ cây tìm quát ở hình 4.17



Hình 4.18. Cây nhúng phân tán để vẽ

Dãy các nút được thăm khi duyệt nó theo phép duyệt của cây nhúng phân:

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa: B H I E J F C G D A

Thứ tự sau: I H J F E G D C B A

Nhận xét

Với thao tác truy cập phép duyệt cây tổng quát và phép duyệt cây nhị phân tổng quát của nó đều cho một dãy tên nhau. Phép duyệt cây tổng quát theo thao tác sau cho dãy tên giống như dãy tên các nút được duyệt theo thao tác giống trong phép duyệt cây nhị phân. Còn phép duyệt cây tổng quát theo thao tác giống thì cho dãy tên không giống bất kỳ dãy nào đối với cây nhị phân tổng quát. Do đó đối với cây tổng quát, nếu định nghĩa phép duyệt như trên ngược lại ta thường chỉ nêu hai phép duyệt theo thao tác truy cập và phép duyệt theo thao tác sau

4. Tìm kiếm trên cây nhị phân

Cây nhị phân được sử dụng vào nhiều mục đích khác nhau. Tuy nhiên việc sử dụng cây nhị phân để lưu giữ và tìm kiếm thông tin vẫn là một trong những ứng dụng quan trọng nhất của cây nhị phân. Trong phần này chúng ta sẽ nghiên cứu một lớp cây nhị phân đặc biệt phục vụ cho việc tìm kiếm thông tin, đó là cây nhị phân tìm kiếm.

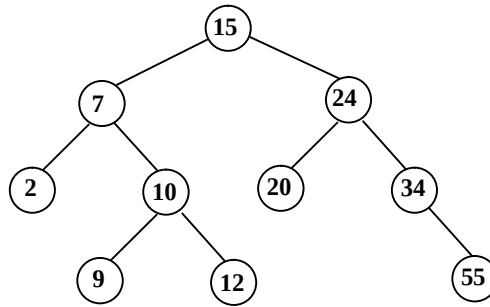
Trong thực tế, một lớp đối tượng nào đó có thể được mô tả bởi một kiểu dữ liệu, các trường của biến ghi biểu diễn các thuộc tính của đối tượng. Trong bài toán tìm kiếm thông tin, ta thường quan tâm đến một nhóm các thuộc tính nào đó của đối tượng mà thuộc tính này hoàn toàn xác định đối tượng. Chúng ta gọi các thuộc tính này là khoá. Như vậy, khoá là một nhóm các thuộc tính của một lớp đối tượng sao cho hai đối tượng khác nhau cần phải có giá trị khác nhau trên nhóm thuộc tính đó.

4.1. Định nghĩa

Cây nhị phân tìm kiếm (cnpkt) là cây nhị phân hoàn chỉnh hoặc thoả mãn định nghĩa các điều kiện sau:

- Khoá của các đỉnh thuộc cây con trái nhỏ hơn khoá nút gốc
- Khoá của nút gốc nhỏ hơn khoá của các đỉnh thuộc cây con phải của gốc
- Cây con trái và cây con phải của gốc cũng là cây nhị phân tìm kiếm

Hình 5.19 biểu diễn một cây nhị phân tìm kiếm, trong đó khoá của các đỉnh là các số nguyên.



Hình 5.19. Một cây nh phân tìm kiếm

4.2. Cài đặt cây nh phân tìm kiếm

Mỗi nút trên cây nh phân tìm kiếm có dạng

left	info	right
------	------	-------

Trong đó trường Left :con tr chỉ tới cây con trái

Right :con tr chỉ tới cây con phải

Info : chứa thông tin của nút

Type

keytype = ...; {kiểu dữ liệu của mỗi nút}

Tree = ^ node;

Node = **record**

Info : keytype;

Left, Right : Tree;

end;

Var T : Tree;

4.3. Các thao tác cơ bản trên cây nh phân tìm kiếm

4.3.1. Tìm kiếm

Tìm kiếm trên cây là một trong các phép toán quan trọng nhất đối với cây nh phân tìm kiếm. Ta xét bài toán sau

Bài toán: Giả sử mỗi đỉnh trên cây nhị phân tìm kiếm là một bản ghi, mỗi con trỏ Root chỉ tới gốc của cây và x là khoá cho tra cứu. Vấn đề đặt ra là, tìm xem trên cây có chứa đỉnh với khoá x hay không.

Giải thuật đệ qui

Procedure search (Root : Tree; x : keytype; var p : tree);

{ Nếu tìm thấy thì p chỉ tới nút có trữ giá khoá bằng x, ngược lại p=nil }

Begin

 p:=root;

 if p <> nil then

 if x < p^.info then search (p^.left, x, p)

 else

 if x > p^.info then search (p^.Right, x, p)

End;

Giải thuật lặp

Trong thuật toán này, ta sử dụng biến địa phương found có kiểu boolean để đi vào khi vào vòng lặp, nó có giá trị ban đầu là false. Nếu tìm kiếm thành công thì found nhận giá trị true, vòng lặp kết thúc và dùng thuật toán để tìm nút có trữ giá khoá bằng x. Còn nếu không tìm thấy thì giá trị của found vẫn là false và giá trị của p là nil

Procedure search (Root : Tree; x:keytype; var p : Tree);

Var Found : boolean;

Begin

 Found:=False;

 P:=Root;

 while (p<>nil) and(not Found) do

 if x > p^.info then p := p^.right

 else

 if x < p^.info then p := p^.left

 else Found = True;

End;

4.3.2. Đi qua CNPTK

Như ta đã biết CNPTK cũng là cây nhị phân nên các phép duyệt trên cây nhị phân cũng vẫn đúng trên CNPTK. Chỉ có lưu ý nhỏ là khi duyệt theo thứ tự giữa thì duyệt cả dãy khoá theo thứ tự tăng dần.

4.3.2. Chèn mới nút vào CNPTK

Việc thêm mới nút nút có trường khoá bằng x vào cây phải đảm bảo điều kiện ràng buộc của CNPTK. Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm vào nút lá sẽ là tiện lợi nhất do ta có thể thấy rõ quá trình từng bước thao tác tìm kiếm. Khi kết thúc việc tìm kiếm cũng chính là lúc tìm được chỗ cần chèn.

Giới thuật đệ quy

Procedure Insert (var Root :Tree; x: kkeytype);

Var p : tree;

Begin

New(p); {Tạo ra nút mới}

P^.info := x;

if root=nil then

begin

Root :=p;

P^.left:= nil;

P^.right:= nil;

End

Else

with Root^ do

if x> info then insert (Right, x)

else if x < info then insert (left, x);

End;

Giới thiệu thuật toán

Trong thuật toán này ta sẽ dùng biến con trỏ để lưu trữ vị trí của các đỉnh của cây nhị phân tìm kiếm. Khi đưa vào một đỉnh nào đó, sẽ xem xét vị trí của nó (phải) tùy theo khóa của đỉnh đó (nhỏ hay lớn) khóa x.

Để tìm vị trí của đỉnh nào đó khi đã biết vị trí của đỉnh con trái (phải) thì phải kiểm tra xem đỉnh này có đỉnh con trái (phải) không. Nếu có thì tiếp tục xem xét, ngược lại thì treo đỉnh mới vào bên trái (phải) của đỉnh đó. Điều kiện dừng là khi vị trí của đỉnh mới đã được chèn vào.

procedure Insert (var Root : Tree; x: keytype)

var p, q : tree;

begin

New(p);

p.info := x;

if Root = nil then

begin

Root := p;

p.left := nil;

p.right := nil;

end

else

begin

q := Root;

while q <> nil do

if x < q.info then

if q.left <> nil then q := q.left

else begin

q.left := p;

p := nil;

end

else if x > q.info then

if q.right <> nil then q := q.right

else begin

q.right := p;

p := nil;

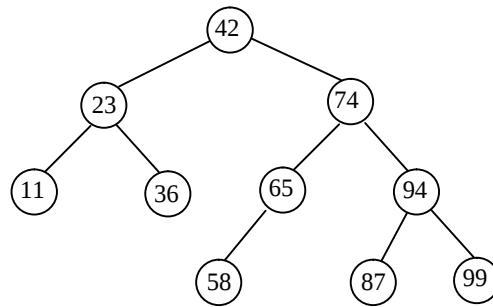
end;

end;
end;

Nhận xét:

Để dùng được CNPTK chúng ta phải đưa dãy khoá đã cho vào bảng cách liên tiếp bằng các nút và tiếp nối khoá, bắt đầu từ cây rỗng. Ban đầu phải đưa lên cây với nút gốc là khoá đầu tiên sau đó đưa vào các khoá tiếp theo, tìm trên cây không có thì bổ sung vào.

Ví dụ với dãy khoá: 42 23 74 11 65 58 94 36 99 87 thì cây nh phân tìm kiếm được có dạng như hình 4.20



Hình 4.20. Một cây nh phân tìm kiếm

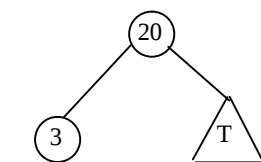
4.3.4. Loại bỏ nút trên cây nh phân tìm kiếm

Để tiếp với phép toán chèn vào là phép toán loại bỏ. Chúng ta cần phải loại bỏ khỏi CNPTK một đỉnh có khoá x (ta gọi tắt là nút x) cho trống, sao cho việc huỷ một nút ra khỏi cây cũng phải bảo đảm điều kiện ràng buộc của CNPTK.

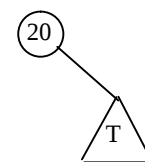
Có ba trường hợp khi huỷ một nút x có thể xảy ra:

- X là nút lá
- X là nút nội lá (chỉ có một con trái hoặc con phải)
- X có đủ hai con (trường hợp tổng quát)

Trường hợp tổng quát: chỉ đơn giản huỷ nút x vì nó không liên quan đến phần tử nào khác.



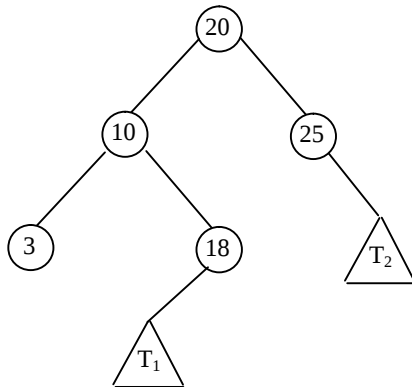
Cây trước khi xoá



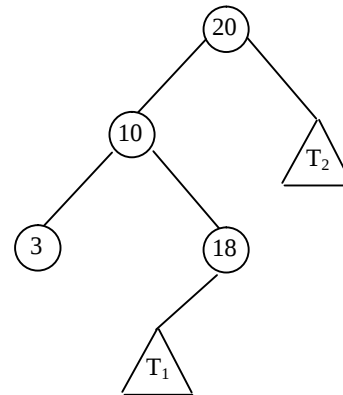
Cây sau khi xoá

Hình 4.21

Trình hợp nhất hai: Trình thực hiện khi xóa nút x có nóc là cha của x và có nút con (nút con trái hoặc nút con phải) của nó



a. Cây trước khi xóa

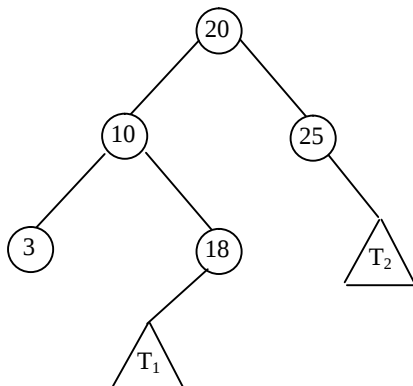


b. Cây sau khi xóa đỉnh (25)

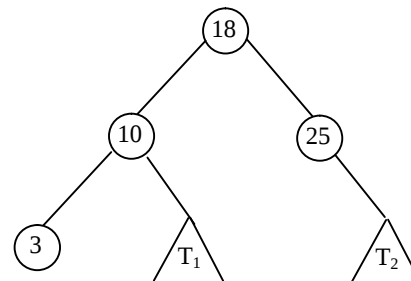
Hình 4.22

Trình hợp nhất ngược: khi nút bị xóa có cả cây con trái và cây con phải, thì nút thay thế nó hoặc là nút đang giữ khóa hiện tại ngay sát trước nó (nút có phải của cây con trái nó) hoặc nút đang giữ khóa hiện tại ngay sát sau nó (nút có trái của cây con phải nó).nhưng vậy ta sẽ phải thay đổi một số mối liên hệ các nút:

- Nút cha của nút bị xóa
- Nút để chọn làm nút thay thế
- Nút cha của nút để chọn làm nút thay thế



1. Cây trước khi xóa đỉnh 20

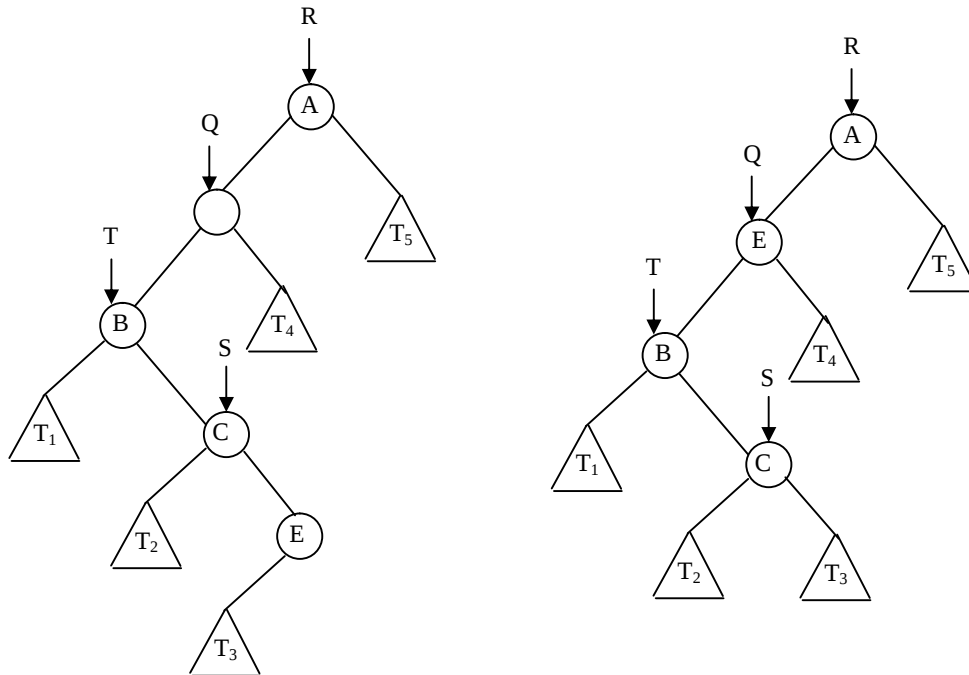


2. Cây sau khi xóa đỉnh 20

Hình 4.23

Trong ví dụ này ta chọn nút thay thế nút bị xóa là nút con phải của cây con trái (nút 18).

Sau đây là giới thuật thực hiện việc loại bỏ một nút từ cây B. Ban đầu Q chính là nút trái hoặc nút phải của một nút R trên cây nhờ phân tìm kiếm, mà ta gọi sẽ đã biết rồi.



1. Cây trước khi xóa nút từ cây B

2. Cây sau khi xóa nút từ cây B

Hình 4.24

```
procedure Del (var Q: Tree); {xóa nút từ cây B}
```

```
var T, S : Tree;
```

```
begin
```

```
P := Q; {X là lý do tìm kiếm nút lá và nút không lá}
```

```
if P^.left = nil then
```

```
begin
```

```
Q := P^.right ; {R^.left := P^.right}
```

```
Dispose(P);
```

```
end
```

```
else
```

```
if P^.right = nil then
```

```

begin
    Q := P^.left;
    Dispose (P);
end
else { X là lý tưởng không phải là ideal }
begin
    T := P^.left;
    if T^.right = nil then
        begin
            T^.right := P^.right;
            Q := T;
            Dispose (P);
        end
    else
        begin
            S := T^.right; {Tìm nút thay thế, là nút con phải của cây }
            while S^.right <> nil do
                begin
                    T := S;
                    S := T^.right;
                end;
            S^.right := P^.right;
            T^.right := S^.left;
            S^.left := P^.left;
            Q:=S;
            Dispose(p);
        end;
    end;
end;

```

end;

Thủ tục xoá trình độ lưu bảng X

- Tìm đến nút có trình độ lưu bảng X

- áp dụng thủ tục Del để xoá

Sau đây chúng ta sẽ viết thủ tục loại khỏi cây gốc Root đỉnh có khoá x cho trình độ c. Đó là thủ tục để qui, nó tìm ra đỉnh có khoá x, sau đó áp dụng thủ tục Del để loại đỉnh đó ra khỏi cây.

```
procedure Delete (var Root :Tree ; x : keytype);
```

```
begin
```

```
  if Root <> nil then
```

```
    if x < Root^.info then Delete (Root^.left, x)
```

```
    else if x > Root^.info then Delete (Root^.right, x)
```

```
    else Del(Root);
```

```
end;
```

Nhận xét

Việc huỷ toàn bộ cây có thể thực hiện thông qua thao tác duyệt cây theo thứ sau. Nghĩa là ta sẽ huỷ cây con trái, cây con phải rồi mới huỷ nút gốc.

```
procedure RemoveTree (var Root: Tree);
```

```
begin
```

```
  if Root <> nil then
```

```
    begin
```

```
      RemoveTree(Root^.left);
```

```
      RemoveTree(Root^.right);
```

```
      Dispose (Root);
```

```
    end;
```

```
end;
```

4.4. Thời gian thực hiện các phép toán trên CNPTK

Trong mục này ta sẽ đánh giá thời gian để thực hiện các phép toán trên CNPTK. Ta có nhận xét rằng, thời gian thực hiện các phép tìm kiếm là số phép so sánh giá trị khoá x cho trước với khoá của các đỉnh nằm trên đường đi từ gốc tới đỉnh nào đó trên cây. Do đó thời gian thực hiện các phép tìm kiếm, bổ sung và loại bỏ là độ dài đường đi từ gốc tới một đỉnh nào đó trên cây.

Với giới thuật tìm kiếm nêu trên, ta thấy dòng cây như phân tìm kiếm dòng để hoàn toàn phụ thuộc vào dãy khoá đưa vào. Như vậy nghĩa là trong quá trình xử lý dòng ta không thể biết trước được cây sẽ phát triển ra sao, hình dạng của nó sẽ như thế nào.

Trong trường hợp nó là một cây như phân hoàn chỉnh (ta gọi là cân để ngay cả khi nó chưa đầy đủ) thì chiều cao của nó là $\lceil \log_2(n+1) \rceil$, nên chi phí tìm kiếm có cấp độ là $O(\log_2 n)$.

Trong trường hợp nó là một cây như phân suy biến thành một danh sách móc nối (khi mà mỗi nút chỉ có một con trẻ nút lá). Lúc đó các thao tác trên cây sẽ có độ phức tạp là $O(n)$.

Ngay cả khi chứng minh được rằng số lượng trung bình các phép so sánh trong tìm kiếm trên cây như phân tìm kiếm là

$$C_{tb} \approx 1,386 \log_2 n$$

Do đó cấp độ lớn của thời gian thực hiện trung bình các phép toán cũng chỉ là $O(\log_2 n)$.

5. Cây cân bằng (avl tree)

Giả sử ta có một tập hợp dữ liệu nào đó. Vấn đề đặt ra là, ta phải tổ chức các dữ liệu đó như thế nào sao cho việc cập nhật thông tin (tìm kiếm, bổ sung và loại bỏ) được hiệu quả nhất. Với các kiểu dữ liệu này ngay cả có tổ chức trên cây như phân tìm kiếm. Khi đó thời gian trung bình thực hiện các phép toán là $O(\log_2 n)$. Trong nhu cầu áp dụng chúng ta cần thêm

xuân thực hiện các phép toán bổ sung và loại bỏ khi CNPTK. Điều này có thể được đảm bảo bằng cách suy biến thành danh sách móc nối. Để với những cây này,

việc thực hiện các phép toán kém hiệu quả do chi phí thời gian thực hiện là $O(n)$. Để khắc phục nhược điểm này ta tổ chức dữ liệu trên một lớp cây để biết thời gian thực hiện các phép toán luôn là $O(\log_2 n)$.

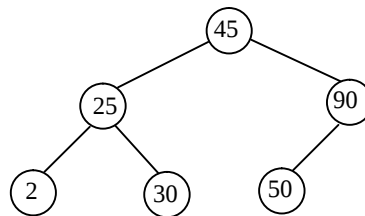
5.1. Cây cân bằng hoàn toàn (CCBHT)

Định nghĩa: Cây cân bằng hoàn toàn là cây nhị phân tìm kiếm mà tất cả các nút của nó, số nút của cây con trái và cây con phải không chênh lệch nhau quá một.

Một cây rất khó đạt được trạng thái cân bằng hoàn toàn và cũng rất dễ mất cân bằng vì khi thêm hay huỷ các nút trên cây có thể làm mất cân bằng (xác suất rất lớn), chi phí cân bằng lại cây là rất lớn vì phải thao tác trên toàn bộ cây.

Tuy nhiên nếu cây cân đối thì việc tìm kiếm sẽ rất nhanh. Đối với CCBHT, trong trường hợp xấu nhất ta cũng chỉ phải tìm qua $\log_2 n$ phần tử (n là số nút trên cây)

Sau đây là ví dụ về một CCBHT



Hình 4.25. Cây cân bằng hoàn toàn

CCBHT có n nút, có chiều cao $h = \log_2 n$. Đây chính là lý do cho phép tìm kiếm hiệu năng tìm kiếm nhanh trên cấu trúc dữ liệu này.

Do CCBHT là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng. Nhưng người tìm kiếm của nó lại rất quan trọng. Vì vậy, cần đưa ra một cấu trúc dữ liệu khác có đặc tính giống CCBHT nhưng ổn định hơn.

Như vậy, cần tìm cách tổ chức một cây để trạng thái cân bằng yếu hơn và việc cân bằng lại chỉ xảy ra ở phạm vi cục bộ nhưng vẫn đảm bảo chi phí cho thao tác tìm kiếm đạt mức $O(\log_2 n)$.

5.2. Cây cân bằng

Năm 1962, P.M. ADELSON - VELSKI và E.LANDIS đã mở đầu phương pháp gọi quy tắc này bằng cách đưa ra dạng cây cân đối mà sau này mang tên của họ, đó

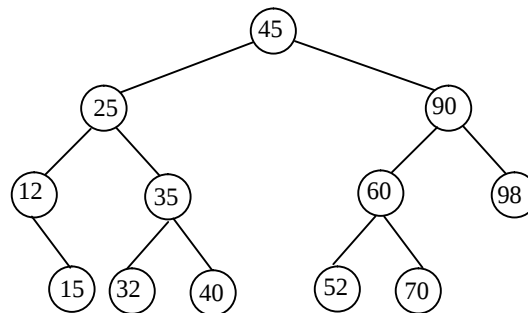
là cây nhị phân cân bằng AVL. Tiếp này vẽ sau, chúng ta sẽ dùng thuật ngữ cây AVL thay cho cây cân bằng.

Thời khi được giới thiệu, cây AVL đã nhanh chóng tìm thấy ứng dụng trong nhiều bài toán khác nhau. Vì vậy, nó mau chóng trở nên thịnh hành và thu hút nhiều nghiên cứu. Trên cây AVL, người ta đã phát triển thêm nhiều loại cấu trúc dữ liệu hữu dụng khác như cây đỏ - đen, B - Tree,...

5.2.1. Định nghĩa

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

Dưới đây là ví dụ cây cân bằng



Hình 4.26. Cây AVL

Dòng họ CCBHT là cây cân bằng. Điều này ngược lại không đúng. Tuy nhiên cây AVL là cấu trúc dữ liệu ổn định hơn CCBHT.

5.2.2. Chiều cao của cây AVL

Một vấn đề quan trọng, nếu đã được đề cập đến phần trước, là ta phải khẳng định cây AVL n nút phải có chiều cao khoảng $\log_2 n$.

Để đánh giá chính xác về chiều cao của cây AVL, ta xét bài toán: cây AVL có chiều cao h sẽ có tối thiểu bao nhiêu nút?

Gọi $N(h)$ số nút tối thiểu của cây AVL có chiều cao h.

Ta có $N(0) = 0$, $N(1) = 1$ và $N(2) = 2$.

Cây AVL tối thiểu có chiều cao h sẽ có cây con AVL tối thiểu chiều cao h-1 và cây con AVL tối thiểu chiều cao h-2. Như vậy:

$$N(h) = 1 + N(h - 1) + N(h - 2) \quad (1)$$

Ta lấy có: $N(h - 1) > N(h - 2)$

Nên từ (1) suy ra

$$N(h) > 2N(h - 2)$$

$$N(h) > 2^2 N(h - 4)$$

...

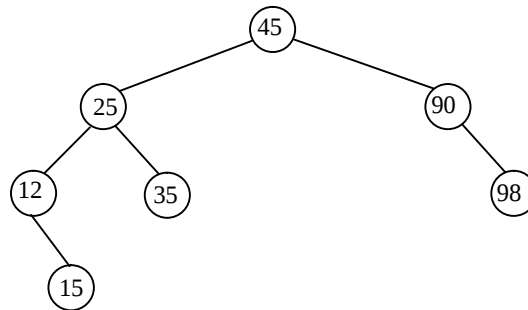
$$N(h) > 2^i N(h - 2i)$$

$$\Rightarrow N(h) > 2^{h/2-1}$$

$$\Rightarrow h < 2\log_2(N(h)) + 2$$

Như vậy, cây AVL có chiều cao $O(\log_2 n)$.

Ví dụ: cây AVL tối thiểu có chiều cao $h = 4$



Hình 4.27. Cây AVL tối thiểu

5.2.3. Chức năng cân bằng của nút

Định nghĩa: chức năng cân bằng của nút là hiệu của chiều cao cây con phải và cây con trái của nó.

Đối với cây AVL chức năng cân bằng của mỗi nút có thể nhận một trong ba giá trị sau đây:

00: độ cao cây con trái bằng độ cao cây con phải

01: cây con phải có độ cao lớn hơn độ cao cây con trái là 1 (lệch phải)

10: cây con trái có độ cao lớn hơn độ cao cây con phải là 1 (lệch trái)

5.2.4. Biện pháp cân bằng AVL

Để khảo sát cây cân bằng, ta cần lưu thêm thông tin về chiều cân bằng tại mỗi nút. Khi đó cây cân bằng được khai báo như sau:

Type

```
keytype = ...;
Tree = ^ Node;
Node = Record
    info : keytype;
    left, right : Tree;
    bal : (LH, EH, RH);
end;
```

var Root : Tree;

Trong khai báo trên ta bổ sung thêm trường bal (balance: cân bằng), EH (Equal Hight: hai cây con cao bằng nhau) có giá trị 00, RH (Right Hight: cao bên phải) có giá trị 01, LH (Left Hight: cao bên trái) có giá trị 10.

Sau đây chúng ta sẽ xét các phép toán bổ sung và loại bỏ trên cây cân bằng

5.2.5. Bổ sung trên cây AVL

Khi bổ sung nút mới vào khoá x cho trước để thực hiện bằng cách sau:

- áp dụng thuật toán bổ sung vào cây như phân tìm kiếm
- Cân bằng lại các đỉnh mà tại đó tính cân bằng bị phá vỡ (độ cao của hai cây con khác nhau 2).

Xét các trường hợp sau

Trường hợp 1: Cây lệch trái (lệch phải) sau khi bổ sung vào cây con phải (trái) cây cân bằng.

Trường hợp 2: Hai cây con có độ cao bằng nhau sau khi bổ sung thì cây lệch trái hoặc lệch phải.

Trường hợp 3: Cây con trái (phải) cao hơn cây con phải (trái)1, sau khi bổ sung vào cây con trái (phải) thì hai cây này có độ cao chênh nhau là 2. Như vậy, tính cân bằng bị phá vỡ.

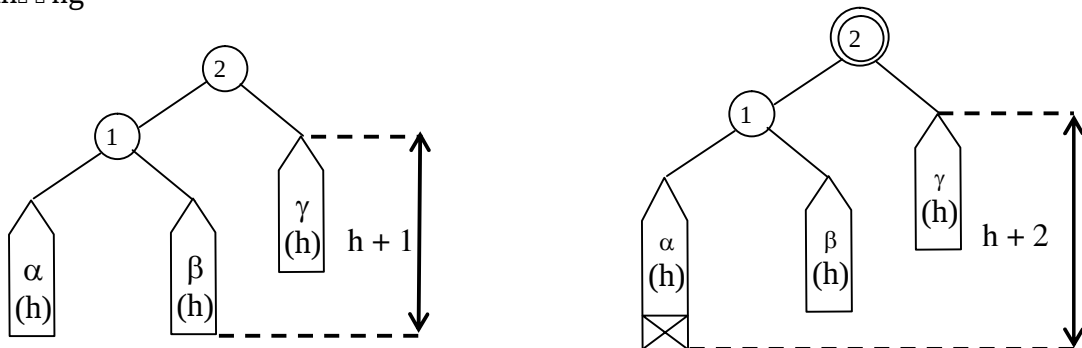
Đối với các trường hợp TH1 và TH2 khi bổ sung nút mới thì tính cân bằng không bị phá vỡ nên ta chỉ cần chỉnh lại các hệ số cân bằng ở nút đã xét và ở các nút tiền bố của nó.

Đối với trường hợp TH3 ta phải sửa lại cây con mà ta đã xét là nút gốc (ta sẽ gọi là nút bất thường).

Quy ước:

-  chèn nút bất thường vào vị trí khoá bằng 2
-  chèn nút mới bổ sung
- $\alpha(h)$ chèn cây con α có chiều cao h

- *Trường hợp 3.1*: Nút bổ sung làm tăng chiều cao cây con trái của nút con trái nút bất thường

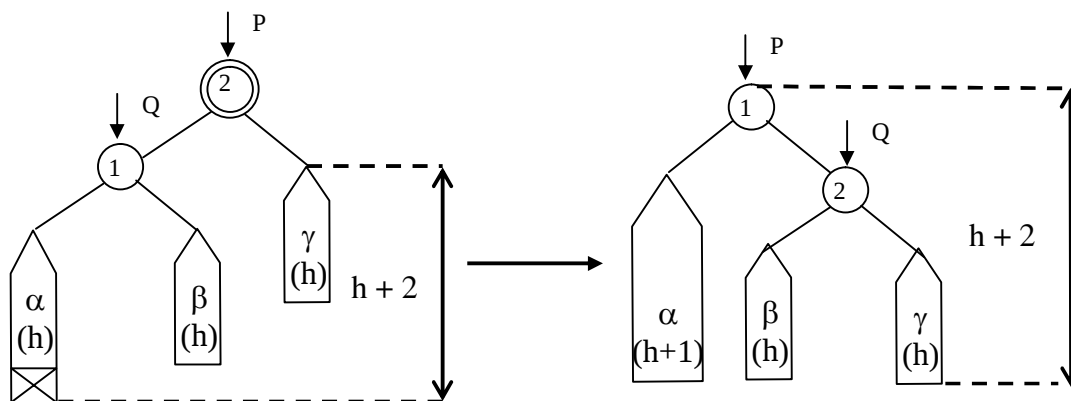


a) Trước khi bổ sung
đi

b) Sau khi bổ sung cây mất cân

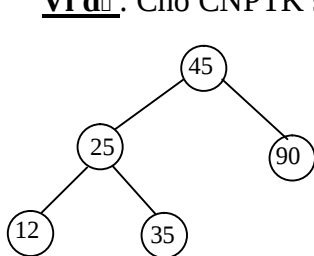
Hình 4.28

Đối với trường hợp này để tái cân bằng ta phải thực hiện phép quay từ trái sang phải để đưa nút (1) lên vị trí gốc cây con, nút (2) sẽ trở thành con phải của nó và β được gán vào thành con trái của (2). Nghĩa là phép quay này là phép quay đơn (single rotation) hay ta còn gọi là phép quay phải.

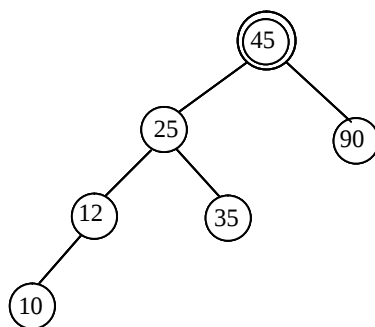


Hình 4.29. Quay phải cây P

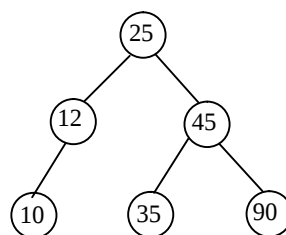
Ví dụ: Cho CNPTK sau



a) Cây ban đầu



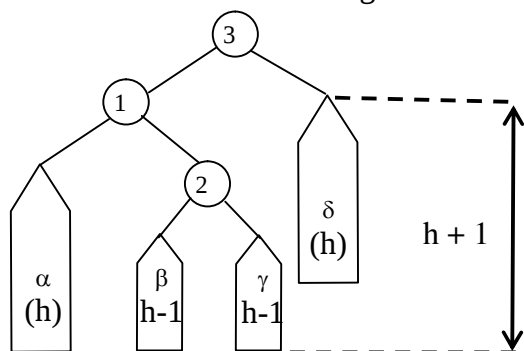
b) Bổ sung thêm nút (10)



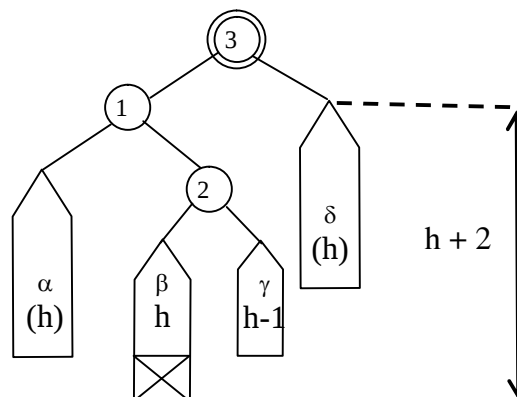
c) Trái cân đối bằng phép quay trái

Hình 4.30

- Trường hợp 3.2: Nút mới bổ sung làm tăng chiều cao cây con phải của nút con trái nút bắt đầu từ gốc.



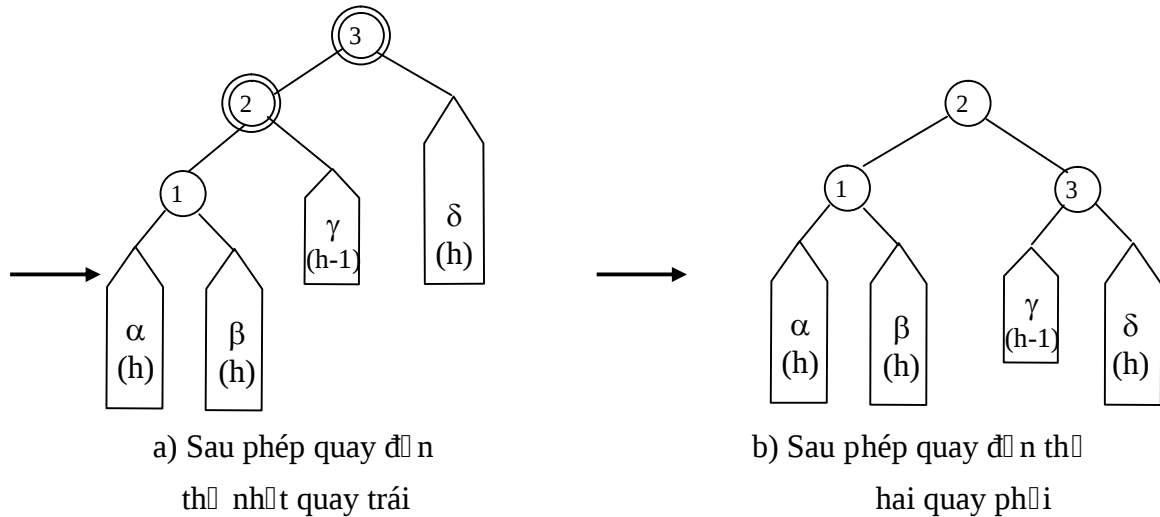
a) Trường hợp khi bổ sung



b) Sau khi bổ sung cây mất cân đối

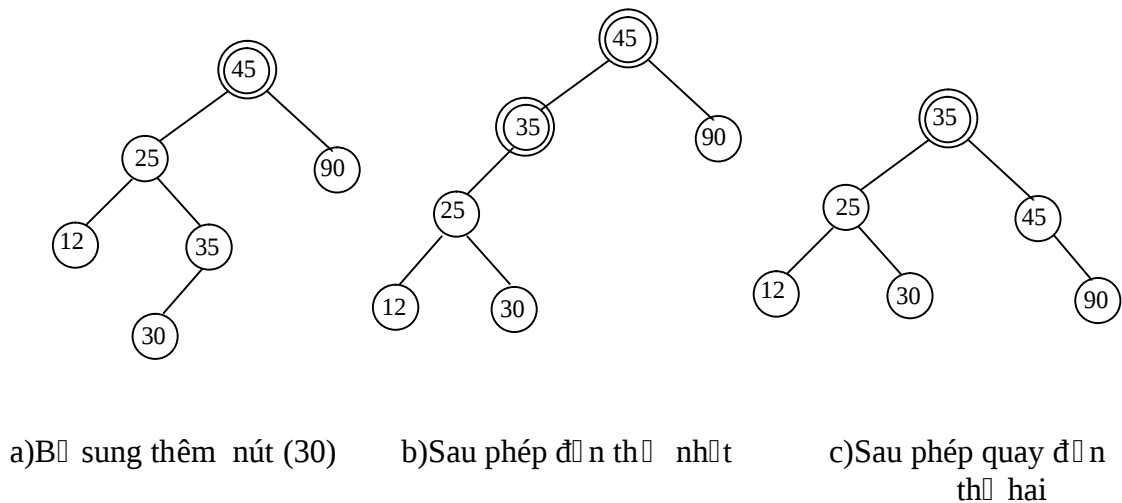
Hình 4.31

Đối với trường hợp này để tái cân bằng ta sẽ dùng phép quay kép (double rotation), đó là việc phải thực hiện hai phép quay để: quay trái để với cây con trái ((1), (2)) và quay phải để với cây ((3), (2)) như hình 5.32



Hình 4.32

Ví dụ: cho cây như hình vẽ



Hình 4.33

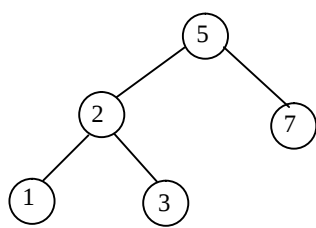
Nhận xét:

Sau khi thực hiện phép quay để tái cân bằng cây con mà nút gốc là nút bất thường không làm ảnh hưởng đến chiều cao của các cây con đó vẫn giữ nguyên như trước lúc bổ sung, nghĩa là phép quay không làm ảnh hưởng tới chiều cao các cây có liên quan tới cây con này.

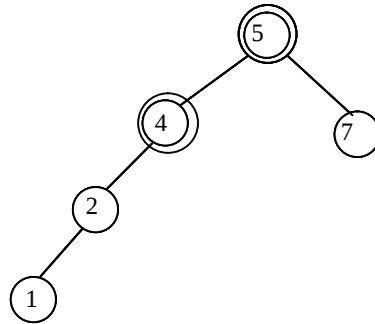
Trong quá trình thực hiện các phép quay, tính chất của cây nhị phân tìm kiếm luôn luôn được đảm bảo.

Đôi khi các trường hợp khi bổ sung thêm nút mới làm tăng chiều cao cây con phải của nút con phải và nút mới bổ sung làm chiều cao cây con trái của nút con phải nút tiếp theo, thì ta lần lượt thực hiện theo thuật toán gốc để tiếp theo với trường hợp TH3.1 và TH3.2.

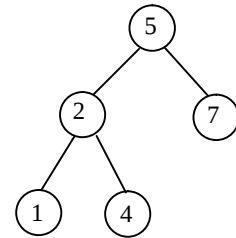
Ví dụ sau đây minh họa các trường hợp và các phép xử lý tiếp theo



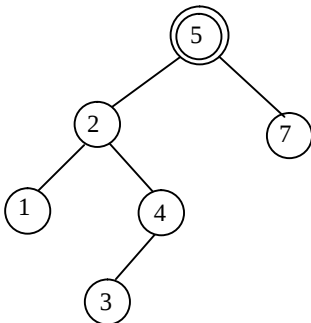
a) Cây ban đầu



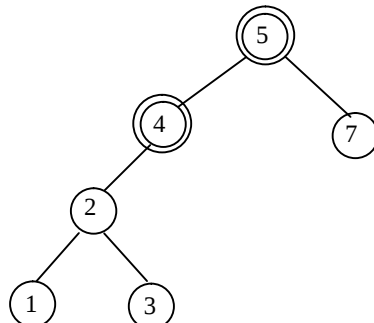
b) Bổ sung thêm nút (1): một cây nhị phân tìm kiếm



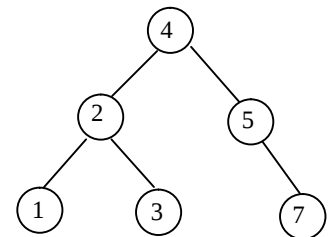
c) Tái cân bằng bằng phép quay trái



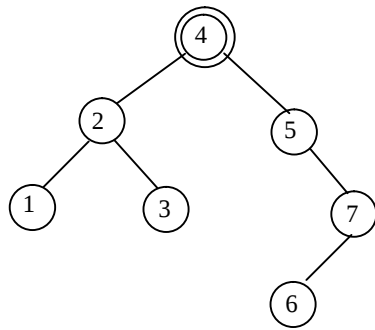
d) Bổ sung thêm nút (3): một cây nhị phân tìm kiếm



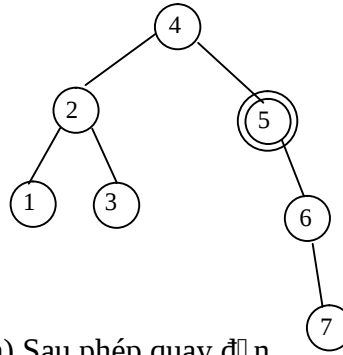
e) Tái cân bằng bằng phép quay kép Sau phép đổi mới



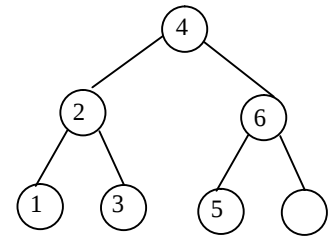
f) Sau phép quay thứ hai



g) Bổ sung thêm nút (6):
mất cân đối trọng hợp TH3.2



h) Sau phép quay đổi
thế nút (quay phải)



i) Sau phép quay đổi
thế hai (quay trái)

Hình 4.34

5.2.6. Huỷ nút trên cây AVL

Cũng giống như thao tác thêm nút, việc huỷ nút x ra khỏi cây AVL cũng cần phải thực hiện trên CNPTK. Chỉ sau khi huỷ, nếu tính cân bằng bị phá vỡ thì ta sẽ thực hiện việc cân bằng lại.

Tuy nhiên việc cân bằng lại trong thao tác huỷ sẽ phức tạp hơn nhiều so với việc cân bằng khi thêm nút do có thể xảy ra phần ngắt dây chuyền.

Nhận xét

- Thao tác thêm nút có độ phức tạp $O(1)$
- Thao tác huỷ nút có độ phức tạp $O(h)$
- Với cây cân bằng trung bình hai lần thêm vào cây thì cần mất lần cân bằng lại; 5 lần huỷ thì cần mất lần cân bằng lại.
- Việc huỷ nút có thể phá vỡ cân bằng dây chuyền các nút tiếp theo cho đến phần tử bị huỷ trong khi thêm vào chỉ cần mất lần cân bằng cục bộ
- Để đảm bảo việc tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn, việc cân bằng lại để giảm hơn nhiều
- Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tổng số nút trên cây là bao nhiêu.