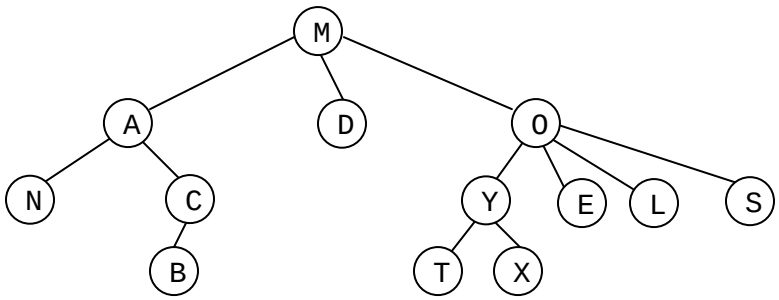
Chương 9 – CÂY NHỊ PHÂN

So với hiện thực liên tục của các cấu trúc dữ liệu, các danh sách liên kết có những ưu điểm lớn về tính mềm dẻo. Nhưng chúng cũng có một điểm yếu, đó là sự tuần tự, chúng được tổ chức theo cách mà việc di chuyển trên chúng chỉ có thể qua từng phần tử một. Trong chương này chúng ta khắc phục nhược điểm này bằng cách sử dụng các cấu trúc dữ liệu cây chứa con trỏ. Cây được dùng trong rất nhiều ứng dụng, đặc biệt trong việc truy xuất dữ liệu.

9.1. Các khái niệm cơ bản về cây

Một **cây** (*tree*) - hình 9.1- gồm một tập hữu hạn các **nút** (*node*) và một tập hữu hạn các **cành** (*branch*) nối giữa các nút. Cành đi vào nút gọi là **cành vào** (*indegree*), cành đi ra khỏi nút gọi là **cành ra** (*outdegree*). Số cành ra từ một nút gọi là **bậc** (*degree*) của nút đó. Nếu cây không rỗng thì phải có một nút gọi là **nút gốc** (*root*), **nút này không có cành vào**. Cây trong hình 9.1 có M là nút gốc. Các nút còn lại, mỗi nút phải có **chính xác một cành vào**. Tất cả các nút đều có thể có 0, 1, hoặc nhiều hơn số cành ra.



(a)

M
- A
- - N
- - C
- - - B
- D

- O
- - Y
- - - T
- - - X
- - E
- - L
- - S

(b)

M (A (N C (B)) D O (Y (T X) E L S))

(c)

Hình 9.1 – Các cách biểu diễn của cây

Nút lá (*leaf*) được định nghĩa như là nút của cây mà số cành ra bằng 0. Các nút không phải nút gốc hoặc nút lá thì được gọi là **nút trung gian** hay **nút trong** (*internal node*). Nút có số cành ra khác 0 có thể gọi là **nút cha** (*parent*) của các nút mà cành ra của nó đi vào, các nút này cũng được gọi là các **nút con** (*child*) của nó. Các nút cùng cha được gọi là các **nút anh em** (*sibling*) với nhau. Nút trên nút cha có thể gọi là **nút ông** (*grandparent,* trong một số bài toán chúng ta cũng cần gọi tên như vậy để trình bày giải thuật).

Theo hình 9.1, các nút lá gồm: N, B, D, T, X, E, L, S; các nút trung gian gồm: A, C, O, Y. Nút Y là cha của hai nút T và X. T và X là con của Y, và là nút anh em với nhau.

Đường đi (*path*) từ nút n_1 đến nút n_k được định nghĩa là một dãy các nút n_1, n_2, ..., n_k sao cho n_i là nút cha của nút n_{i+1} với $1 \leq i < k$. **Chiều dài** (*length*) **đường đi** này là số cành trên nó, đó là k-1. Mỗi nút có đường đi chiều dài bằng 0 đến chính nó. Trong một cây, từ nút gốc đến mỗi nút còn lại chỉ có duy nhất một đường đi.

Đối với mỗi nút n_i, **độ sâu** (*depth*) hay còn gọi là **mức** (*level*) của nó chính là chiều dài đường đi duy nhất từ nút gốc đến nó cộng 1. Nút gốc có mức bằng 1. **Chiều cao** (*height*) **của nút** n_i là chiều dài của đường đi dài nhất từ nó đến một nút lá. Mọi nút lá có chiều cao bằng 1. **Chiều cao của cây** bằng chiều cao của nút gốc. **Độ sâu của cây** bằng độ sâu của nút lá sâu nhất, nó luôn bằng chiều cao của cây.

Nếu giữa nút n_1 và nút n_2 có một đường đi, thì n_1 đươc gọi là **nút trước** (*ancestor*) của n_2 và n_2 là **nút sau** (*descendant*) của n_1.

M là nút trước của nút B. M là nút gốc, có mức là 1. Đường đi từ M đến B là: M, A, C, B, có chiều dài là 3. B có mức là 4.

B là nút lá, có chiều cao là 1. Chiều cao của C là 2, của A là 3, và của M là 4 chính bằng chiều cao của cây.

Một cây có thể được chia thành nhiều cây con (*subtree*). Một cây con là bất kỳ một cấu trúc cây bên dưới của nút gốc. Nút đầu tiên của cây con là nút gốc của nó và đôi khi người ta dùng tên của nút này để gọi cho cây con. Cây con gốc A (hay gọi tắt là cây con A) gồm các nút A, N, C, B. Một cây con cũng có thể chia thành nhiều cây con khác. Khái niệm cây con dẫn đến định nghĩa đệ quy cho cây như sau:

Định nghĩa: Một cây là tập các nút mà
- là tập rỗng, hoặc
- có một nút gọi là nút gốc có không hoặc nhiều cây con, các cây con cũng là cây

Các cách biểu diễn cây

Thông thường có 3 cách biểu diễn cây: biểu diễn bằng đồ thị – hình 9.1a, biểu diễn bằng cách canh lề – hình 9.1b, và biểu diễn bằng biểu thức có dấu ngoặc – hình 9.1c.

9.2. Cây nhị phân

9.2.1. Các định nghĩa

Định nghĩa: Một cây nhị phân hoặc là một cây rỗng, hoặc bao gồm một nút gọi là nút gốc (*root*) và hai cây nhị phân được gọi là cây con bên trái và cây con bên phải của nút gốc.

Lưu ý rằng định nghĩa này là định nghĩa toán học cho một cấu trúc cây. Để đặc tả cây nhị phân như một kiểu dữ liệu trừu tượng, chúng ta cần chỉ ra các tác vụ có thể thực hiện trên cây nhị phân. Các phương thức cơ bản của một cây nhị phân tổng quát chúng ta bàn đến có thể là tạo cây, giải phóng cây, kiểm tra cây rỗng, duyệt cây,...

Định nghĩa này không quan tâm đến cách hiện thực của cây nhị phân trong bộ nhớ. Chúng ta sẽ thấy ngay rằng một biểu diễn liên kết là tự nhiên và dễ sử dụng, nhưng các hiện thực khác như mảng liên tục cũng có thể thích hợp. Định nghĩa này cũng không quan tâm đến các khóa hoặc cách mà chúng được sắp thứ tự. Cây nhị phân được dùng cho nhiều mục đích khác hơn là chỉ có tìm kiếm truy xuất, do đó chúng ta cần giữ một định nghĩa tổng quát.

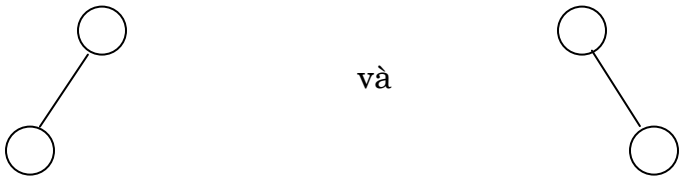
Trước khi xem xét xa hơn về các đặc tính chung của cây nhị phân, chúng ta hãy quay về định nghĩa tổng quát và nhìn xem bản chất đệ quy của nó thể hiện như thế nào trong cấu trúc của một cây nhị phân nhỏ.

Trường hợp thứ nhất, một trường hợp cơ bản không liên quan đến đệ quy, đó là một cây nhị phân rỗng.

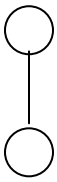
Cách duy nhất để xây dựng một cây nhị phân có một nút là cho nút đó là gốc và cho hai cây con trái và phải là hai cây rỗng.

Với cây có hai nút, một trong hai sẽ là gốc và nút còn lại sẽ thuộc cây con. Hoặc cây con trái hoặc cây con phải là cây rỗng, và cây còn lại chứa chính xác chỉ

một nút. Như vậy có hai cây nhị phân khác nhau có hai nút. Hai cây nhị phân có hai nút có thể được vẽ như sau:



và đây là hai cây khác nhau. Chúng ta sẽ không bao giờ vẽ bất kỳ một phần nào của một cây nhị phân như sau:

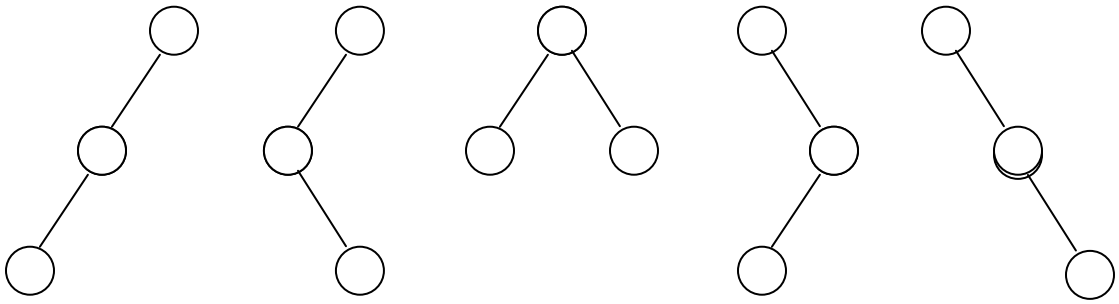


do chúng ta sẽ không thể nói được nút bên dưới là con trái hay con phải của nút trên.

Đối với trường hợp cây nhị phân có ba nút, một trong chúng sẽ là gốc, và hai nút còn lại có thể được chia giữa cây con trái và cây con phải theo một trong các cách sau:

2 + 0 1 + 1 0 + 2

Do có thể có hai cây nhị phân có hai nút và chỉ có một cây rỗng, trường hợp thứ nhất trên cho ra hai cây nhị phân. Trường hợp thứ ba, tương tự, cho thêm hai cây khác. Trường hợp giữa, cây con trái và cây con phải mỗi cây chỉ có một nút, và chỉ có duy nhất một cây nhị phân có một nút nên trường hợp này chỉ có một cây nhị phân. Tất cả chúng ta có năm cây nhị phân có ba nút:



Hình 9.2- Các cây nhị phân có ba nút

Các bước để xây dựng cây này là một điển hình cho các trường hợp lớn hơn. Chúng ta bắt đầu từ gốc của cây và xem các nút còn lại như là các cách phân chia giữa cây con trái và cây con phải. Cây con trái và cây con phải lúc này sẽ là các trường hợp nhỏ hơn mà chúng ta đã biết.

Gọi N là số nút của cây nhị phân, H là chiều cao của cây thì,

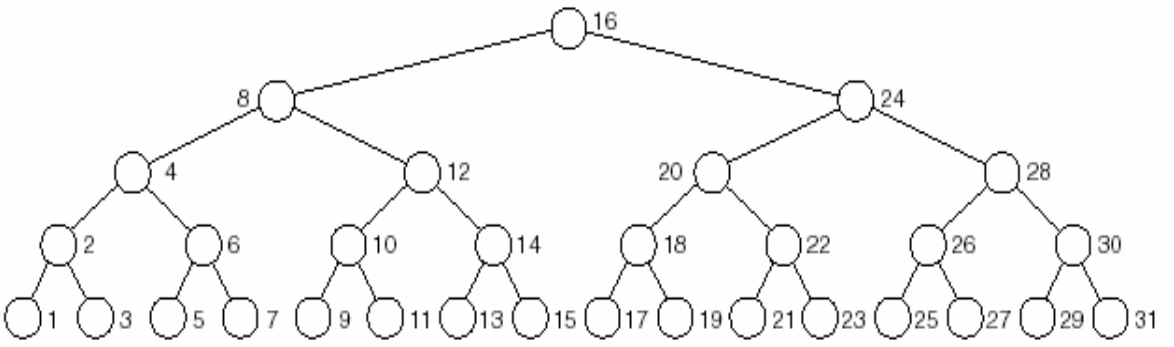
$$H_{max} = N,\ H_{min} = \lfloor \log_2 N \rfloor + 1$$
$$N_{min} = H,\ N_{max} = 2^H - 1$$

Khoảng cách từ một nút đến nút gốc xác định chi phí cần để định vị nó. Chẳng hạn một nút có độ sâu là 5 thì chúng ta phải đi từ nút gốc và qua 5 cành trên đường đi từ gốc đến nó để tìm đến nó. Do đó, nếu cây càng thấp thì việc tìm đến các nút sẽ càng nhanh. Điều này dẫn đến tính chất cân bằng của cây nhị phân. Hệ số cân bằng của cây (*balance factor*) là sự chênh lệch giữa chiều cao của hai cây con trái và phải của nó:

$$B = H_L - H_R$$

Một **cây cân bằng** khi hệ số này bằng 0 và các cây con của nó cũng cân bằng. Một cây nhị phân cân bằng với chiều cao cho trước sẽ có số nút là lớn nhất có thể. Ngược lại, với số nút cho trước cây nhị phân cân bằng có chiều cao nhỏ nhất. Thông thường điều này rất khó xảy ra nên định nghĩa có thể nới lỏng hơn với các trị B = –1, 0, hoặc 1 thay vì chỉ là 0. Chúng ta sẽ học kỹ hơn về cây cân bằng AVL trong phần sau.

Một **cây nhị phân đầy đủ** (*complete tree*) là cây có được số nút tối đa với chiều cao của nó. Đó cũng chính là cây có B=0 với mọi nút. Thuật ngữ **cây nhị phân gần như đầy đủ** cũng được dùng cho trường hợp cây có được chiều cao tối thiểu của nó và mọi nút ở mức lớn nhất dồn hết về bên trái.
Hình 9.3 biểu diễn cây nhị phân đầy đủ có 31 nút. Giả sử loại đi các nút 19, 21, 23, 25, 27, 29, 31 ta có một cây nhị phân gần như đầy đủ.



Hình 9.3 – Cây nhị phân đầy đủ với 31 nút.

9.2.2. Duyệt cây nhị phân

Một trong các tác vụ quan trọng nhất được thực hiện trên cây nhị phân là duyệt cây (*traversal*). **Một phép duyệt cây là một sự di chuyển qua khắp các nút của cây theo một thứ tự định trước, mỗi nút chỉ được xử lý một**

lần duy nhất. Cũng như phép duyệt trên các cấu trúc dữ liệu khác, hành động mà chúng ta cần làm khi ghé qua một nút sẽ phụ thuộc vào ứng dụng.

Đối với các danh sách, các nút nằm theo một thứ tự tự nhiên từ nút đầu đến nút cuối, và phép duyệt cũng theo thứ tự này. Tuy nhiên, đối với các cây, có rất nhiều thứ tự khác nhau để duyệt qua các nút.

Có 2 cách tiếp cận chính khi duyệt cây: duyệt theo chiều sâu và duyệt theo chiều rộng.

Duyệt theo chiều sâu (*defth-first traversal*): mọi nút sau của một nút con được duyệt trước khi sang một nút con khác.

Duyệt theo chiều rộng (*breadth-first traversal*): mọi nút trong cùng một mức được duyệt trước khi sang mức khác.

9.2.2.1. Duyệt theo chiều sâu

Tại một nút cho trước, có ba việc mà chúng ta muốn làm: ghé nút này, duyệt cây con bên trái, duyệt cây con bên phải. Sự khác nhau giữa các phương án duyệt là chúng ta quyết định ghé nút đó trước hoặc sau khi duyệt hai cây con, hoặc giữa khi duyệt hai cây con.

Nếu chúng ta gọi công việc ghé một nút là V, duyệt cây con trái là L, duyệt cây con phải là R, thì có đến sáu cách kết hợp giữa chúng:

VLR LVR LRV VRL RVL RLV.

Các thứ tự duyệt cây chuẩn

Theo quy ước chuẩn, sáu cách duyệt trên giảm xuống chỉ còn ba bởi chúng ta chỉ xem xét các cách mà trong đó cây con trái được duyệt trước cây con phải. Ba cách còn lại rõ ràng là tương tự vì chúng chính là những thứ tự ngược của ba cách chuẩn. Các cách chuẩn này được đặt tên như sau:

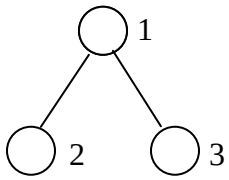
VLR	LVR	LRV
preorder	*inorder*	*postorder*

Các tên này được chọn tương ứng với bước mà nút đã cho được ghé đến. Trong phép duyệt *preorder*, nút được ghé trước các cây con; trong phép duyệt *inorder*, nó được ghé đến giữa khi duyệt hai cây con; và trong phép duyệt *postorder*, gốc của cây được ghé sau hai cây con của nó.

Phép duyệt *inorder* đôi khi còn được gọi là phép duyệt đối xứng (*symmetric order*), và *postorder* được gọi là *endorder*.

Các ví dụ đơn giản

Trong ví dụ thứ nhất, chúng ta hãy xét cây nhị phân sau:

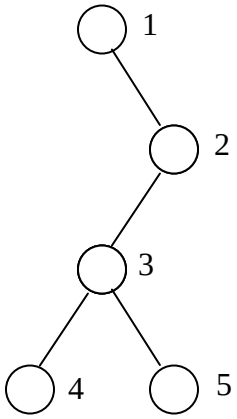


Với phép duyệt *preorder*, gốc cây mang nhãn 1 được ghé đầu tiên, sau đó phép duyệt di chuyển sang cây con trái. Cây con trái chỉ chứa một nút có nhãn là 2, nút này được duyệt thứ hai. Sau đó phép duyệt chuyển sang cây con phải của nút gốc, cuối cùng là nút mang nhãn 3 được ghé. Vậy phép duyệt preorder sẽ ghé các nút theo thứ tự 1, 2, 3.

Trước khi gốc của cây được ghé theo thứ tự *inorder*, chúng ta phải duyệt cây con trái của nó trước. Do đó nút mang nhãn 2 được ghé đầu tiên. Đó là nút duy nhất trong cây con trái. Sau đó phép duyệt chuyển đến nút gốc mang nhãn 1, và cuối cùng duyệt qua cây con phải. Vậy phép duyệt inorder sẽ ghé các nút theo thứ tự 2, 1, 3.

Với phép duyệt *postorder*, chúng ta phải duyệt các hai cây con trái và phải trước khi ghé nút gốc. Trước tiên chúng ta đi đến cây con bên trái chỉ có một nút mang nhãn 2, và nó được ghé đầu tiên. Tiếp theo, chúng ta duyệt qua cây con phải, ghé nút 3, và cuối cùng chúng ta ghé nút 1. Phép duyệt *postorder* duyệt các nút theo thứ tự 2, 3, 1.

Ví dụ thứ hai phức tạp hơn, chúng ta hãy xem xét cây nhị phân dưới đây:

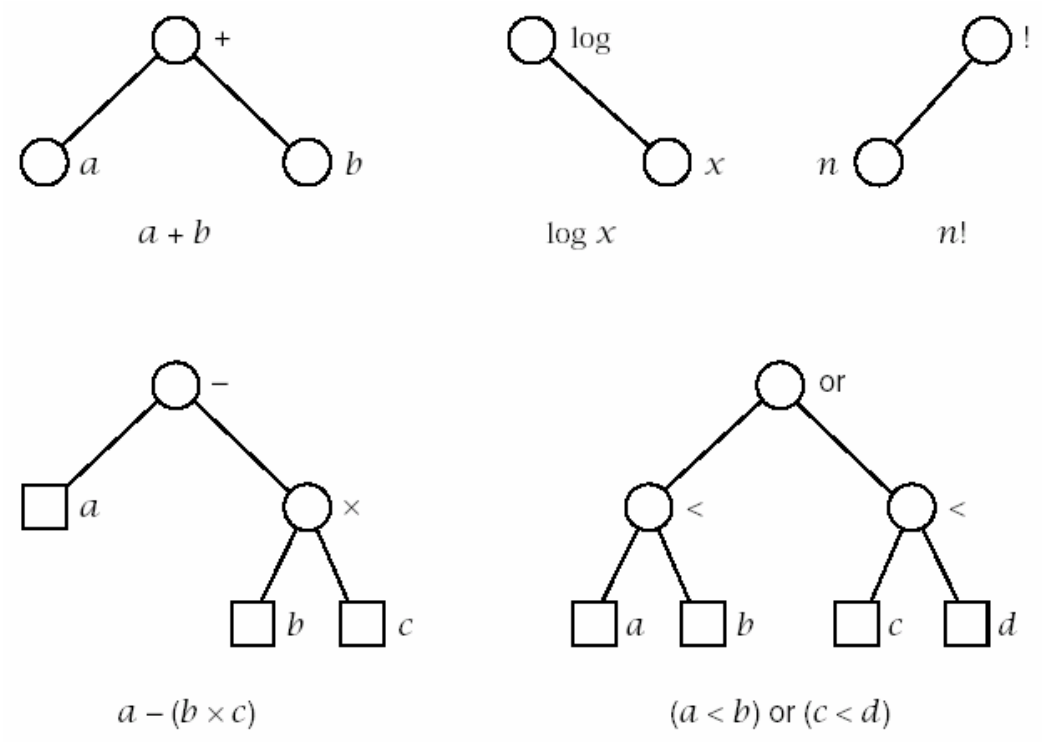


Tương tự cách làm trên chúng ta có phép duyệt *preorder* sẽ ghé các nút theo thứ tự 1, 2, 3, 4, 5. Phép duyệt *inorder* sẽ ghé các nút theo thứ tự 1, 4, 3, 5, 2. Phép duyệt *postorder* sẽ ghé các nút theo thứ tự 4, 5, 3, 2, 1.

Cây biểu thức

Cách chọn các tên *preorder*, *inorder*, và *postorder* cho ba phép duyệt cây trên không phải là tình cờ, nó liên quan chặt chẽ đến một trong những ứng dụng, đó là các cây biểu thức.

Một cây biểu thức (*expression tree*) được tạo nên từ các toán hạng đơn giản và các toán tử (số học hoặc luận lý) của biểu thức bằng cách thay thế các toán hạng đơn giản bằng các nút lá của một cây nhị phân và các toán tử bằng các nút bên trong cây. Đối với mỗi toán tử hai ngôi, cây con trái chứa mọi toán hạng và mọi toán tử thuộc toán hạng bên trái của toán tử đó, và cây con phải chứa mọi toán hạng và mọi toán tử thuộc toán hạng bên phải của nó.

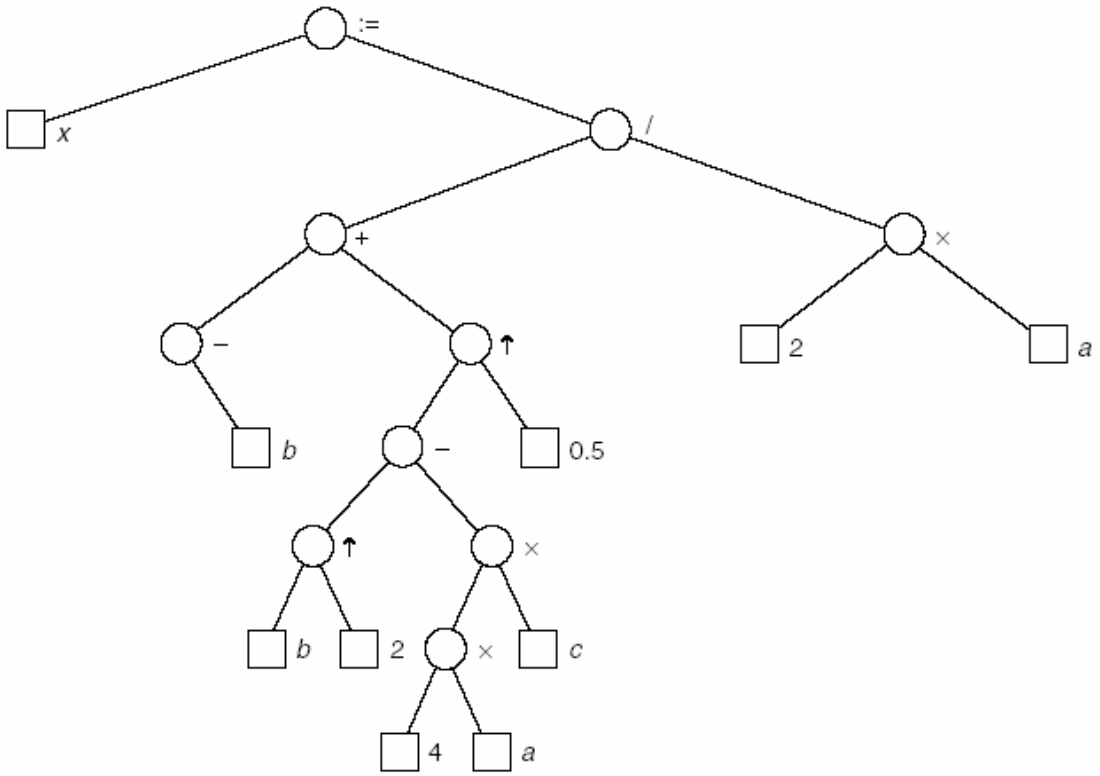


Hình 9.4 – Cây biểu thức

Đối với toán tử một ngôi, một trong hai cây con sẽ rỗng. Chúng ta thường viết một vài toán tử một ngôi phía bên trái của toán hạng của chúng, chẳng hạn dấu trừ (phép lấy số âm) hoặc các hàm chuẩn như log() và cos(). Các toán tử một ngôi khác được viết bên phải của toán hạng, chẳng hạn hàm giai thừa ()! hoặc hàm bình phương $()^2$. Đôi khi cả hai phía đều hợp lệ, như phép lấy đạo hàm có thể viết d/dx phía bên trái, hoặc ()' phía bên phải, hoặc toán tử tăng ++ có ảnh hưởng

khác nhau khi nằm bên trái hoặc nằm bên phải. Nếu toán tử được ghi bên trái, thì trong cây biểu thức nó sẽ có cây con trái rỗng, như vậy toán hạng sẽ xuất hiện bên phải của nó trong cây. Ngược lại, nếu toán tử xuất hiện bên phải, thì cây con phải của nó sẽ rỗng, và toán hạng sẽ là cây con trái của nó.

Một số cây biểu thức của một vài biểu thức đơn giản được minh họa trong hình 9.4. Hình 9.5 biểu diễn một công thức bậc hai phức tạp hơn. Ba thứ tự duyệt cây chuẩn cho cây biểu thức này liệt kê trong hình 9.6.



$x := (-b + (b \uparrow 2 - 4 \times a \times c) \uparrow 0.5)/(2 \times a)$

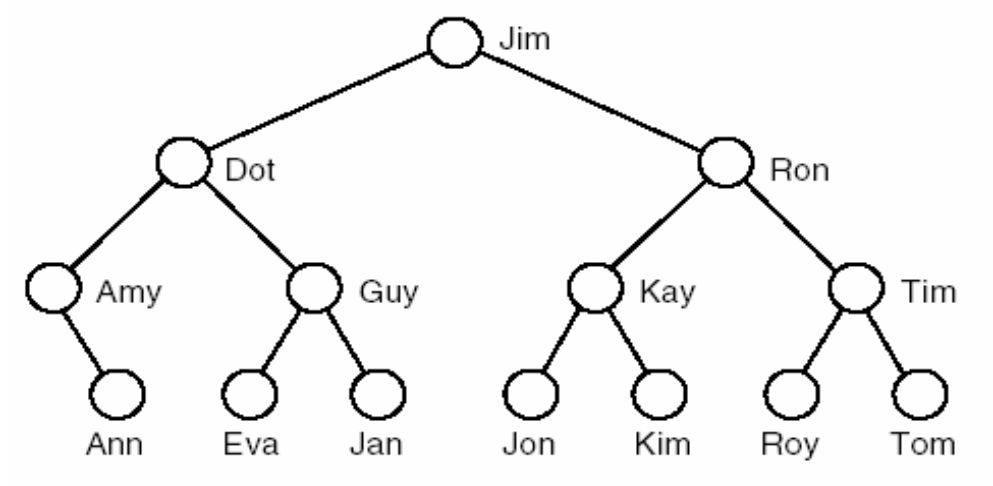
Hình 9.5 – Cây biểu thức cho công thức bậc hai.

Các tên của các phép duyệt liên quan đến các dạng Balan của biểu thức: duyệt cây biểu thức theo *preorder* là dạng *prefix*, trong đó mỗi toán tử nằm trước các toán hạng của nó; duyệt cây biểu thức theo *inorder* là dạng *infix* (cách viết biểu thức quen thuộc của chúng ta); duyệt cây biểu thức theo *postorder* là dạng *postfix*, mọi toán hạng nằm trước toán tử của chúng. Như vậy các cây con trái và cây con phải của mỗi nút luôn là các toán hạng của nó, và vị trí tương đối của một toán tử so với các toán hạng của nó trong ba dạng Balan hoàn toàn giống với thứ tự tương đối của các lần ghé các thành phần này theo một trong ba phép duyệt cây biểu thức.

Expression:	$a + b$	$\log x$	$n!$	$a - (b \times c)$	$(a < b)$ or $(c < d)$
Preorder :	$+\ a\ b$	$\log\ x$	$!\ n$	$-\ a \times b\ c$	or $<\ a\ b <\ c\ d$
Inorder :	$a + b$	$\log x$	$n\ !$	$a - b \times c$	$a < b$ or $c < d$
Postorder :	$a\ b\ +$	$x\ \log$	$n\ !$	$a\ b\ c \times -$	$a\ b <\ c\ d <$ or

Hình 9.6 – Các thứ tư duyệt cho cây biểu thức

Cây so sánh



Hình 9.7 – Cây so sánh để tìm nhị phân

Chúng ta hãy xem lại ví dụ trong hình 9.7 và ghi lại kết quả của ba phép duyệt cây chuẩn như sau:

preorder: Jim Dot Amy Ann Guy Eva Jan Ron Kay Jon Kim Tim Roy Tom
inorder: Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom
postorder:Ann Amy Eva Jan Guy Dot Jon Kim Kay Roy Tom Tim Ron Jim

Phép duyệt *inorder* cho các tên có thứ tự theo alphabet. Cách tạo một cây so sánh như hình 9.7 như sau: di chuyển sang trái khi khóa của nút cần thêm nhỏ hơn khóa của nút đang xét, ngược lại thì di chuyển sang phải. Như vậy cây nhị phân trên đã được xây dựng sao cho mọi nút trong cây con trái của mỗi nút có thứ tự nhỏ hơn thứ tự của nó, và mọi nút trong cây con phải có thứ tự lớn hơn nó. Do đối với mỗi nút, phép duyệt *inorder* sẽ duyệt qua các nút trong cây con trái trước, rồi đến chính nó, và cuối cùng là các nút trong cây con phải, nên chúng ta có được các nút theo thứ tự.

Trong phần sau chúng ta sẽ tìm hiểu các cây nhị phân với đặc tính trên, chúng còn được gọi là các cây nhị phân tìm kiếm (*binary search tree*), do chúng rất có ích và hiệu quả cho yêu cầu tìm kiếm.

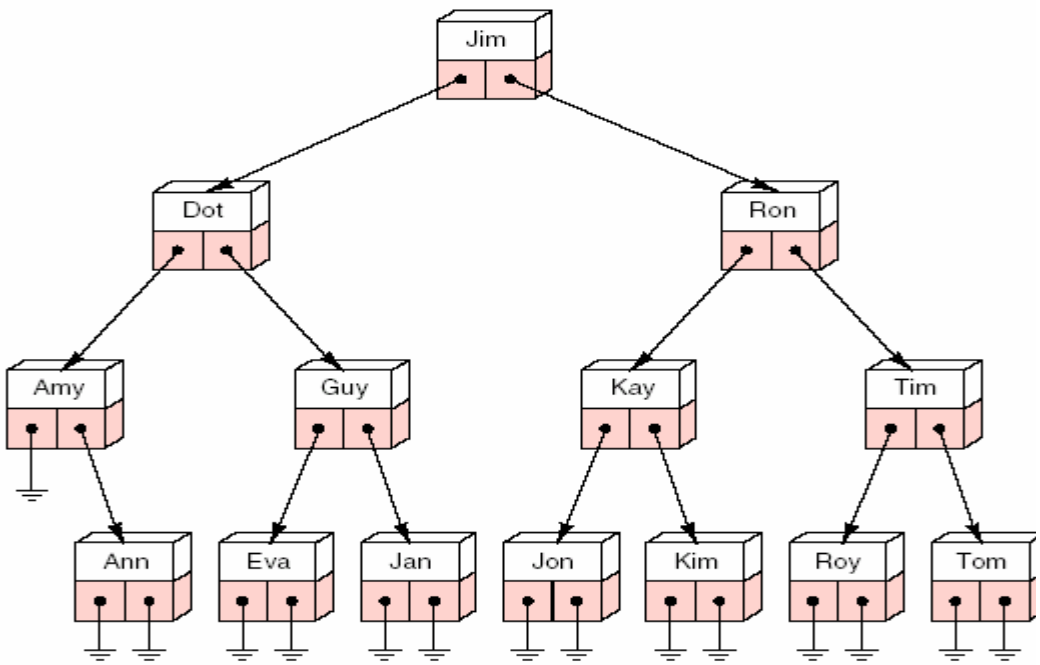
9.2.2.2. Duyệt theo chiều rộng

Thứ tự duyệt cây theo chiều rộng là thứ tự duyệt hết mức này đến mức kia, có thể từ mức cao đến mức thấp hoặc ngược lại. Trong mỗi mức có thể duyệt từ trái sang phải hoặc từ phải sang trái. Ví dụ cây trong hình 9.7 nếu duyệt theo chiều rộng từ mức thấp đến mức cao, trong mỗi mức duyệt từ trái sang phải, ta có: Jim, Dot, Ron, Amy, Guy, Kay, Tim, Ann, Eva, Jan, Jon, Kim, Roy, Tom.

9.2.3. Hiện thực liên kết của cây nhị phân

Chúng ta hãy xem xét cách biểu diễn của các nút để xây dựng nên cây.

9.2.3.1. Cấu trúc cơ bản cho một nút trong cây nhị phân



Hình 9.8 – Cây nhị phân liên kết

Mỗi nút của một cây nhị phân (cũng là gốc của một cây con nào đó) có hai cây con trái và phải. Các cây con này có thể được xác định thông qua các con trỏ chỉ đến các nút gốc của nó. Chúng ta có đặc tả sau:

```
template <class Entry>
struct Binary_node {
// Các thành phần.
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
```

```
// constructors:
   Binary_node();
   Binary_node(const Entry &x);
};
```

Binary_node chứa hai *constructor* đều khởi gán các thuộc tính con trỏ là NULL mỗi khi đối tượng được tạo ra.

Trong hình 9.8, chúng ta thấy những tham chiếu NULL, tuy nhiên chúng ta có thể quy ước rằng các cây con rỗng và các cành đến nó có thể bỏ qua không cần hiển thị khi vẽ cây.

9.2.3.2. Đặc tả cây nhị phân

Một cây nhị phân có một hiện thực tự nhiên trong vùng nhớ liên kết. Cũng như các cấu trúc liên kết, chúng ta sẽ cấp phát động các nút, nối kết chúng lại với nhau. Chúng ta chỉ cần một con trỏ chỉ đến nút gốc của cây.

```
template <class Entry>
class Binary_tree {
public:
   Binary_tree();
   bool empty() const;
   void preorder(void (*visit)(Entry &));
   void inorder(void (*visit)(Entry &));
   void postorder(void (*visit)(Entry &));

   int size() const;
   void clear();
   int height() const;
   void insert(const Entry &);

   Binary_tree (const Binary_tree<Entry> &original);
   Binary_tree & operator =(const Binary_tree<Entry> &original);
   ~Binary_tree();

protected:
   //  Các hàm đệ quy phụ trợ:
   void recursive_inorder(Binary_node<Entry>*sub_root,
                                          void (*visit)(Entry &))
   void recursive_preorder(Binary_node<Entry>*sub_root,
                                          void (*visit)(Entry &))
   void recursive_postorder(Binary_node<Entry>*sub_root,
                                          void (*visit)(Entry &))

   Binary_node<Entry> *root;
};
```

Với con trỏ root, có thể dễ dàng nhận ra một cây nhị phân rỗng bởi biểu thức

```
root == NULL;
```

và khi tạo một cây nhị phân mới chúng ta chỉ cần gán root bằng NULL.

```
template <class Entry>
Binary_tree<Entry>::Binary_tree()
/*
post: Cây nhị phân rỗng được tạo ra.
*/
{
   root = NULL;
}
```

Phương thức empty kiểm tra xem một cây nhị phân có rỗng hay không:

```
template <class Entry>
bool Binary_tree<Entry>::empty() const
/*
post: Trả về true nếu cậy rỗng, ngược lại trả về false.
*/
{
   return root == NULL;
}
```

9.2.3.3. Duyệt cây

Bây giờ chúng ta sẽ xây dựng các phương thức duyệt một cây nhị phân liên kết theo cả ba phép duyệt cơ bản. Cũng như trước kia, chúng ta sẽ giả sử như chúng ta đã có hàm visit để thực hiện một công việc mong muốn nào đó cho mỗi nút của cây. Và như các hàm duyệt cho những cấu trúc dữ liệu khác, con trỏ hàm visit sẽ là một thông số hình thức của các hàm duyệt cây.

Trong các hàm duyệt cây, chúng ta cần ghé đến nút gốc và duyệt các cây con của nó. Đệ quy sẽ làm cho việc duyệt các cây con trở nên hết sức dễ dàng. Các cây con được tìm thấy nhờ các con trỏ trong nút gốc, do đó các con trỏ này cần được chuyển cho các lần gọi đệ quy. Mỗi phương thức duyệt cần gọi hàm đệ quy có một thông số con trỏ. Chẳng hạn, phương thức duyệt *inorder* được viết như sau:

```
template <class Entry>
void Binary_tree<Entry>::inorder(void (*visit)(Entry &))
/*
post: Cây được duyệt theo thứ tự inorder
uses: Hàm recursive_inorder
*/
{
   recursive_inorder(root, visit);
}
```

Một cách tổng quát, chúng ta nhận thấy một cách tổng quát rằng bất kỳ phương thức nào của Binary_tree mà bản chất là một quá trình đệ quy cũng được hiện thực bằng cách gọi một hàm đệ quy phụ trợ có thông số là gốc của cây. Hàm duyệt *inorder* phụ trợ được hiện thực bằng cách gọi đệ quy đơn giản như sau:

```
template <class Entry>
void Binary_tree<Entry>::recursive_inorder(Binary_node<Entry>*sub_root, void
                                           (*visit)(Entry &))
/*
pre:  sub_root hoặc là NULL hoặc chỉ đến gốc của một cây con.
post: Cây con được duyệt theo thứ tự inorder.
uses: Hàm recursive_inorder được gọi đệ quy.
*/
{
   if (sub_root != NULL) {
      recursive_inorder(sub_root->left, visit);
      (*visit)(sub_root->data);
      recursive_inorder(sub_root->right, visit);
   }
}
```

Các phương thức duyệt khác cũng được xây dựng một cách tương tự bằng cách gọi các hàm đệ quy phụ trợ. các hàm đệ quy phụ trợ có hiện thực như sau:

```
template <class Entry>
void Binary_tree<Entry>::recursive_preorder(Binary_node<Entry> *sub_root,
                                    void (*visit)(Entry &))
/*
pre:  sub_root hoặc là NULL hoặc chỉ đến gốc của một cây con.
post: Cây con được duyệt theo thứ tự preorder.
uses: Hàm recursive_inorder được gọi đệ quy.
*/
{
   if (sub_root != NULL) {
      (*visit)(sub_root->data);
      recursive_preorder(sub_root->left, visit);
      recursive_preorder(sub_root->right, visit);
   }
}
```

```
template <class Entry>
void Binary_tree<Entry>::recursive_postorder(Binary_node<Entry> *sub_root,
                                   void (*visit)(Entry &))
/*
pre:  sub_root hoặc là NULL hoặc chỉ đến gốc của một cây con.
post: Cây con được duyệt theo thứ tự postorder.
uses: Hàm recursive_inorder được gọi đệ quy.
*/
{
   if (sub_root != NULL) {
      recursive_postorder(sub_root->left, visit);
      recursive_postorder(sub_root->right, visit);
      (*visit)(sub_root->data);
   }
}
```

Chương trình duyệt cây theo chiều rộng luôn phải sử dụng đến CTDL hàng đợi. Nếu duyệt theo thứ tự từ mức thấp đến mức cao, mỗi mức duyệt từ trái sang phải, trước tiên nút gốc được đưa vào hàng đợi. Công việc được lặp cho đến khi

hàng đợi rỗng: lấy một nút ra khỏi hàng đợi, xử lý cho nó, đưa các nút con của nó vào hàng đợi (theo đúng thứ tự từ trái sang phải). Các biến thể khác của phép duyệt cây theo chiều rộng cũng vô cùng đơn giản, sinh viên có thể tự suy nghĩ thêm.

Chúng ta để phần hiện thực các phương thức của cây nhị phân như **`height`**, **`size`**, và **`clear`** như là bài tập. Các phương thức này cũng được hiện thực dễ dàng bằng cách gọi các hàm đệ quy phụ trợ. Trong phần bài tập chúng ta cũng sẽ viết phương thức **`insert`** để thêm các phần tử vào cây nhị phân, phương thức này cần để tạo một cây nhị phân, sau đó, kết hợp với các phương thức nêu trên, chúng ta sẽ kiểm tra lớp `Binary_tree` mà chúng ta xây dựng được.

Trong phần sau của chương này, chúng ta sẽ xây dựng các lớp dẫn xuất từ cây nhị phân có nhiều đặc tính và hữu ích hơn (các lớp dẫn xuất này sẽ có các phương thức thêm hoặc loại phần tử trong cây thích hợp với đặc tính của từng loại cây). Còn hiện tại thì chúng ta không nên thêm những phương thức như vậy vào cây nhị phân cơ bản.

Mặc dù lớp `Binary_tree` của chúng ta xuất hiện chỉ như là một lớp vỏ mà các phương thức của nó đều đẩy các công việc cần làm đến cho các hàm phụ trợ, bản thân nó lại mang một ý nghĩa quan trọng. Lớp này tập trung vào nó nhiều hàm khác nhau và cung cấp một giao diện thuận tiện tương tự các kiểu dữ liệu trừu tượng khác. Hơn nữa, chính lớp mới có thể cung cấp tính đóng kín: không có nó thì các dữ liệu trong cây không được bảo vệ một cách an toàn và dễ dàng bị thâm nhập và sửa đổi ngoài ý muốn. Cuối cùng, chúng ta có thể thấy lớp `Binary_tree` còn làm một lớp cơ sở cho các lớp khác dẫn xuất từ nó hữu ích hơn.

9.3. Cây nhị phân tìm kiếm

Chúng ta hãy xem xét vấn đề tìm kiếm một khóa trong một danh sách liên kết. Không có cách nào khác ngoài cách di chuyển trên danh sách mỗi lần một phần tử, và do đó việc tìm kiếm trên danh sách liên kết luôn là tìm tuần tự. Việc tìm kiếm sẽ trở nên nhanh hơn nhiều nếu chúng ta sử dụng danh sách liên tục và tìm nhị phân. Tuy nhiên, danh sách liên tục lại không phù hợp với sự biến động dữ liệu. Giả sử chúng ta cũng cần thay đổi danh sách thường xuyên, thêm các phần tử mới hoặc loại các phần tử hiện có. Như vậy danh sách liên tục sẽ chậm hơn nhiều so với danh sách liên kết, do việc thêm và loại phần tử trong danh sách liên tục mỗi lần đều đòi hỏi phải di chuyển nhiều phần tử sang các vị trí khác. Trong danh sách liên kết chỉ cần thay đổi một vài con trỏ mà thôi.

Vấn đề chủ chốt trong phần này chính là:

Liệu chúng ta có thể tìm một hiện thực cho các danh sách có thứ tự mà trong đó chúng ta có thể tìm kiếm, hoặc thêm bớt phần tử đều rất nhanh?

Cây nhị phân cho một lời giải tốt cho vấn đề này. Bằng cách đặt các `entry` của một danh sách có thứ tự vào trong các nút của một cây nhị phân, chúng ta sẽ thấy rằng chúng ta có thể tìm một khóa cho trước qua O(log n) bước, giống như tìm nhị phân, đồng thời chúng ta cũng có giải thuật thêm và loại phần tử trong O(log n) thời gian.

Định nghĩa: Một cây nhị phân tìm kiếm (*binary search tree* -BST) là một cây hoặc rỗng hoặc trong đó mỗi nút có một khóa (nằm trong phần dữ liệu của nó) và thỏa các điều kiện sau:
1. Khóa của nút gốc lớn hơn khóa của bất kỳ nút nào trong cây con trái của nó.
2. Khóa của nút gốc nhỏ hơn khóa của bất kỳ nút nào trong cây con phải của nó.
3. Cây con trái và cây con phải của gốc cũng là các cây nhị phân tìm kiếm.

Hai đặc tính đầu tiên mô tả thứ tự liên quan đến khóa của nút gốc, đặc tính thứ ba mở rộng chúng đến mọi nút trong cây; do đó chúng ta có thể tiếp tục sử dụng cấu trúc đệ quy của cây nhị phân. Chúng ta đã viết định nghĩa này theo cách mà nó bảo đảm rằng không có hai phần tử trong một cây nhị phân tìm kiếm có cùng khóa, do các khóa trong cây con trái chính xác là nhỏ hơn khóa của gốc, và các khóa của cây con phải cũng chính xác là lớn hơn khóa của gốc. Chúng ta có thể thay đổi định nghĩa để cho phép các phần tử trùng khóa. Tuy nhiên trong phần này chúng ta có thể giả sử rằng:

Không có hai phần tử trong một cây nhị phân tìm kiếm có trùng khóa.

Các cây nhị phân trong hình 9.7 và 9.8 là các cây nhị phân tìm kiếm, do quyết định di chuyển sang trái hoặc phải tại mỗi nút dựa trên cách so sánh các khóa trong định nghĩa của một cây tìm kiếm.

9.3.1. Các danh sách có thứ tự và các cách hiện thực

Đã đến lúc bắt đầu xây dựng các phương thức C++ để xử lý cho cây nhị phân tìm kiếm, chúng ta nên lưu ý rằng có ít nhất là ba quan điểm khác nhau dưới đây:

- Chúng ta có thể xem cây nhị phân tìm kiếm như một kiểu dữ liệu trừu tượng mới với định nghĩa và các phương thức của nó;
- Do cây nhị phân tìm kiếm là một dạng đặc biệt của cây nhị phân, chúng ta có thể xem các phương thức của nó như các dạng đặc biệt của các phương thức của cây nhị phân;
- Do các phần tử trong cây nhị phân tìm kiếm có chứa các khóa, và do chúng được gán dữ liệu để truy xuất thông tin theo cách tương tự như các danh sách có thứ tự, chúng ta có thể nghiên cứu cây nhị phân tìm kiếm như là một hiện thực mới của kiểu dữ liệu trừu tượng danh sách có thứ tự (*ordered list ADT*).

Trong thực tế, đôi khi các lập trình viên chỉ tập trung vào một trong ba quan điểm trên, và chúng ta cũng sẽ như thế. Chúng ta sẽ đặc tả lớp cây nhị phân tìm kiếm dẫn xuất từ cây nhị phân. Như vậy, lớp cây nhị phân của chúng ta lại biểu diễn cho một kiểu dữ liệu trừu tượng khác. Tuy nhiên, lớp mới sẽ thừa kế các phương thức của lớp cây nhị phân trước kia. Bằng cách này, sự sử dụng lớp thừa kế nhấn mạnh vào hai quan điểm trên. Quan điểm thứ ba thường được nhìn thấy trong các ứng dụng của cây nhị phân tìm kiếm. Chương trình của người sử dụng có thể dùng lớp của chúng ta để giải quyết các bài toán sắp thứ tự và tìm kiếm liên quan đến danh sách có thứ tự.

Chúng ta đã đưa ra những khai báo C++ cho phép xử lý cho cây nhị phân. Chúng ta sẽ sử dụng hiện thực này của cây nhị phân làm cơ sở cho lớp cây nhị phân tìm kiếm.

```
template <class Record>
class Search_tree: public Binary_tree<Record> {
public:
   Error_code insert(const Record &new_data);
   Error_code remove(const Record &old_data);
   Error_code tree_search(Record &target) const;
 private: //  Các hàm đệ quy phụ trợ.
};
```

Do lớp cây nhị phân tìm kiếm thừa kế từ lớp nhị phân, chúng ta có thể dùng lại các phương thức đã định nghĩa trên cây nhị phân tổng quát cho cây nhị phân tìm kiếm. Các phương thức này là `constructor`, `destructor`, `clear`, `empty`, `size`, `height`, và các phương thức duyệt `preorder`, `inorder`, `postorder`. Để thêm vào các phương thức này, một cây nhị phân tìm kiếm cần thêm các phương thức chuyên biệt hóa như `insert`, `remove`, và `tree_search`.

9.3.2. Tìm kiếm trên cây

Phương thức mới quan trọng đầu tiên của cây nhị phân tìm kiếm là: tìm một phần tử với một khóa cho trước trong cây nhị phân tìm kiếm liên kết. Đặc tả của phương thức như sau:

```
Error_code Search_tree<Record> :: tree_search (Record &target) const;
```
post: Nếu có một phần tử có khóa trùng với khóa trong **target**, thì **target** được chép đè bởi phần tử này, phương thức trả về `success`; ngược lại phương thức trả về `not_present`.

Ở đây chúng ta dùng lớp `Record` như đã mô tả trong chương **7.** Ngoài thuộc tính thuộc lớp `Key` dành cho khóa, trong `Record` có thể còn nhiều thành phần dữ liệu khác. Trong các ứng dụng, phương thức này thường được gọi với thông số `target` chỉ chứa trị của thành phần khóa. Nếu tìm thấy khóa cần tìm, phương thức sẽ bổ sung các dữ liệu đầy đủ vào các thành phần khác còn lại của `Record`.

9.3.2.1. Chiến lược

Để tìm một khóa, trước tiên chúng ta so sánh nó với khóa của nút gốc trong cây. Nếu so trùng, giải thuật dừng. Ngược lại, chúng ta đi sang cây con trái hoặc cây con phải và lặp lại việc tìm kiếm trong cây con này.

Ví dụ, chúng ta cần tìm tên Kim trong cây nhị phân tìm kiếm hình 9.7 và 9.8. Chúng ta so sánh Kim với phần tử tại nút gốc, Jim. Do Kim lớn hơn Jim theo thứ tự *alphabet*, chúng ta đi sang phải và tiếp tục so sánh Kim với Ron. Do Kim nhỏ hơn Jon, chúng ta di chuyển sang trái, so sánh Kim với Kay. Chúng ta lại di chuyển sang phải và gặp được phần tử cần tìm.

Đây rõ ràng là một quá trình đệ quy, cho nên chúng ta sẽ hiện thực phương thức này bằng cách gọi một hàm đệ quy phụ trợ. Liệu điều kiện dừng của việc tìm kiếm đệ quy là gì? Rõ ràng là, nếu chúng ta tìm thấy phần tử cần tìm, hàm sẽ kết thúc thành công. Nếu không, chúng ta sẽ cứ tiếp tục tìm cho đến khi gặp một cây rỗng, trong trường hợp này việc tìm kiếm thất bại.

Hàm đệ quy tìm kiếm phụ trợ sẽ trả về một con trỏ chỉ đến phần tử được tìm thấy. Mặc dù con trỏ này có thể được sử dụng để truy xuất đến dữ liệu lưu trong đối tượng cây, nhưng chỉ có các hàm là những phương thức của cây mới có thể gọi hàm tìm kiếm phụ trợ này (vì chỉ có chúng mới có thể gởi thuộc tính `root` của cây làm thông số). Như vậy, việc trả về con trỏ đến một nút sẽ không vi phạm đến tính đóng kín của cây khi nhìn từ ứng dụng bên ngoài. Chúng ta có đặc tả sau đây của hàm tìm kiếm phụ trợ.

```
Binary_node<Record> *Search_tree<Record> :: search_for_node
             (Binary_node<Record> *sub_root, const Record &target) const;
```
pre: `sub_root` hoặc là `NULL` hoặc chỉ đến một cây con của lớp `Search_tree`.
post:Nếu khóa của `target` không có trong cây con `sub_tree`, hàm trả về `NULL`; ngược lại, hàm trả về con trỏ đến nút chứa `target`.

9.3.2.2. Phiên bản đệ quy

Cách đơn giản nhất để viết hàm tìm kiếm trên là dùng đệ quy:

```
template <class Record>
Binary_node<Record> *Search_tree<Record>::search_for_node(
            Binary_node<Record>* sub_root, const Record &target) const
{
   if (sub_root == NULL || sub_root->data == target) return sub_root;
   else if (sub_root->data < target)
             return search_for_node(sub_root->right, target);
        else return search_for_node(sub_root->left, target);
}
```

9.3.2.3. Khử đệ quy

Đệ quy xuất hiện trong hàm trên chỉ là đệ quy đuôi, đó là lệnh cuối cùng được thực hiện trong hàm. Bằng cách sử dụng vòng lặp, đệ quy đuôi luôn có thể được thay thế bởi sự lặp lại nhiều lần. Trong trường hợp này chúng ta cần viết vòng lặp thế cho lệnh `if` đầu tiên, và thay đổi thông số `sub_root` để nó di chuyển xuống các cành của cây.

```
template <class Record>
Binary_node<Record> *Search_tree<Record>::search_for_node(
            Binary_node<Record> *sub_root, const Record &target) const
{  while (sub_root != NULL && sub_root->data != target)
      if (sub_root->data < target)
           sub_root = sub_root->right;
      else sub_root = sub_root->left;
   return sub_root;
}
```

9.3.2.4. Phương thức `tree_search`

Phương thức `tree_search` đơn giản chỉ gọi hàm phụ trợ `search_for_node` để tìm nút chứa khóa trùng với khóa cần tìm trong cây tìm kiếm nhị phân. Sau đó nó trích dữ liệu cần thiết và trả về `Error_code` tương ứng.

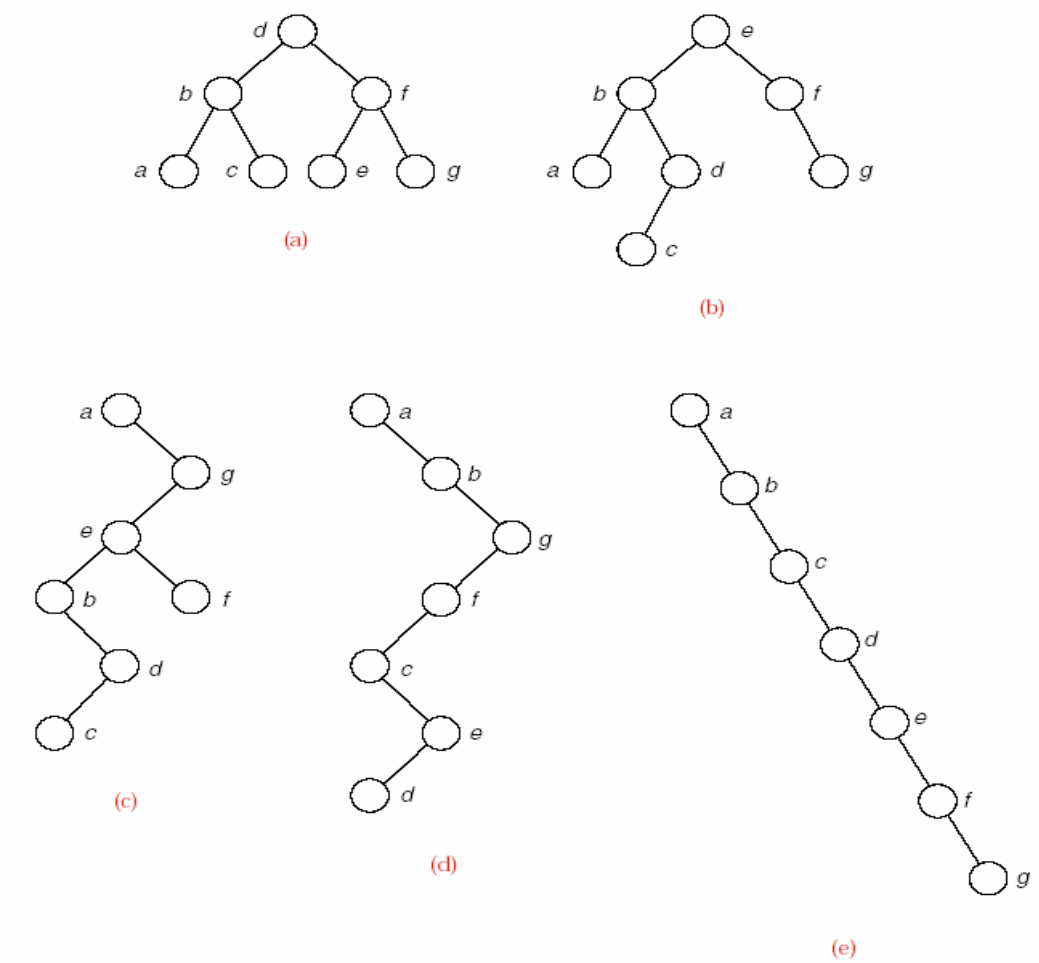
```
template <class Record>
Error_code Search_tree<Record>::tree_search(Record &target) const
/*
post: Nếu tìm thấy khóa cần tìm trong target, phương thức sẽ bổ sung các dữ liệu đầy đủ vào
      các thành phần khác còn lại của target và trả về success. Ngược lại trả về
      not_present. Cả hai trường hợp cây đều không thay đổi.
Uses: Hàm search_for_node
*/
{
   Error_code result = success;
   Binary_node<Record> *found = search_for_node(root, target);
   if (found == NULL)
      result = not_present;
   else
      target = found->data;
   return result;
}
```

9.3.2.5. Hành vi của giải thuật

Chúng ta thấy rằng `tree_search` dựa trên cơ sở của tìm nhị phân. Nếu chúng ta thực hiện tìm nhị phân trên một danh sách có thứ tự, chúng ta thấy rằng tìm nhị phân thực hiện các phép so sánh hoàn toàn giống như `tree_search`. Chúng ta cũng đã biết tìm nhị phân thực hiện O(log n) lần so sánh đối với danh sách có chiều dài n. Điều này thực sự tốt so với các phương pháp tìm kiếm khác, do log n tăng rất chậm khi n tăng.

Cây trong hình 9.9a là cây tốt nhất đối với việc tìm kiếm. Cây càng "rậm rạp" càng tốt: nó có chiều cao nhỏ nhất đối với số nút cho trước. Số nút nằm giữa nút gốc và nút cần tìm, kể cả nút cần tìm, là số lần so sánh cần thực hiện khi tìm kiếm. Vì vậy, cây càng rậm rạp thì số lần so sánh này càng nhỏ.

Không phải chúng ta luôn có thể dự đoán trước hình dạng của một cây nhị phân tìm kiếm trước khi cây được tạo ra, và cây ở hình (b) là một cây điển hình thường có nhất so với cây ở hình (a). Trong cây này, việc tìm phần tử c cần bốn lần so sánh, còn hình (a) chỉ cần ba lần so sánh. Tuy nhiên, cây ở hình (b) vẫn còn tương đối rậm rạp và việc tìm kiếm trên nó chỉ dở hơn một ít so với cây tối ưu trong hình (a).



Hình 9.9 – Một vài cây nhị phân tìm kiếm có các khóa giống nhau

Trong hình (c), cây đã trở nên suy thoái, và việc tìm phần tử c cần đến 6 lần so sánh. Hình (d) và (e) các cây đã trở thành chuỗi các mắc xích. Khi tìm trên các chuỗi mắc xích như vậy, `tree_search` không thể làm được gì khác hơn là duyệt từ phần tử này sang phần tử kia. Nói cách khác, `tree_search` khi thực hiện trên chuỗi các mắc xích như vậy đã suy thoái thành tìm tuần tự. Trong trường hợp xấu

nhất này, với một cây có n nút, tree_search có thể cần đến n lần so sánh để tìm một phần tử.

Trong thực tế, nếu các nút được thêm vào một cây nhị phân tìm kiếm treo một thứ tự ngẫu nhiên, thì rất hiếm khi cây trở nên suy thoái thành các dạng như ở hình (d) hoặc (e). Thay vào đó, cây sẽ có hình dạng gần giống với hình (a) hoặc (b). Do đó, hầu như là tree_search luôn thực hiện gần giống với tìm nhị phân. Đối với cây nhị phân tìm kiếm ngẫu nhiên, sự thực hiện tree_search chỉ chậm hơn 39% so với sự tìm kiếm tối ưu với lg n lần so sánh các khóa, và như vậy nó cũng tốt hơn rất nhiều so với tìm tuần tự có n lần so sánh.

9.3.3. Thêm phần tử vào cây nhị phân tìm kiếm

9.3.3.1. Đặt vấn đề

Tác vụ quan trọng tiếp theo đối với chúng ta là thêm một phần tử mới vào cây nhị phân tìm kiếm sao cho các khóa trong cây vẫn giữ đúng thứ tự; có nghĩa là, cây kết quả vẫn thỏa định nghĩa của một cây nhị phân tìm kiếm. Đặc tả tác vụ này như sau:

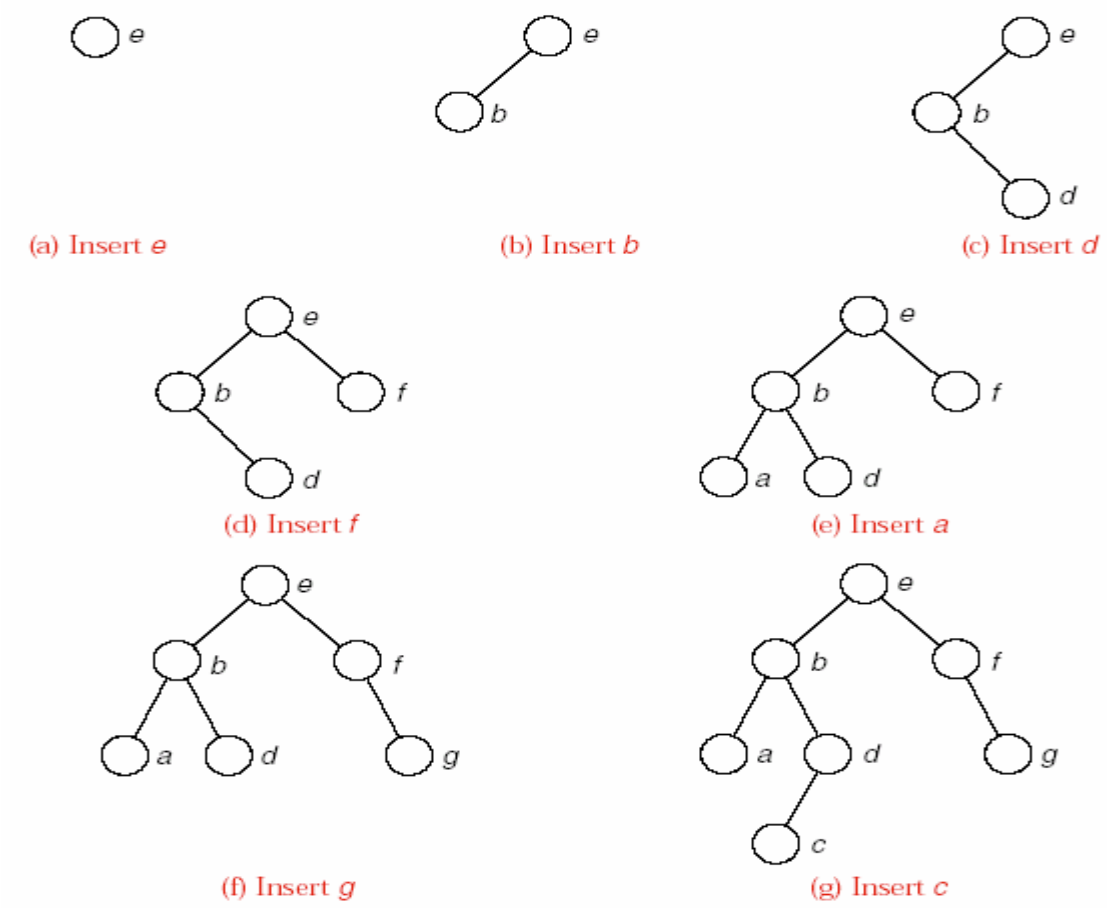
```
Error_code Search_tree<Record>::insert(const Record &new_data);
```
post: Nếu bản ghi có khóa trùng với khóa của new_data đã có trong cây thì Search_tree trả về duplicate_error. Ngược lại, new_data được thêm vào cây sao cho cây vẫn giữ được các đặc tính của một cây nhị phân tìm kiếm, phương thức trả về success.

9.3.3.2. Các ví dụ

Trước khi viết phương thức này, chúng ta hãy xem một vài ví dụ. Hình 9.10 minh họa những gì xảy ra khi chúng ta thêm các khóa e, b, d, f, a, g, c vào một cây rỗng theo đúng thứ tự này.

Khi phần tử đầu tiên e được thêm vào, nó trở thành gốc của cây như hình 9.10a. Khi thêm b, do b nhỏ hơn e, b được thêm vào cây con bên trái của e như hình (b). Tiếp theo, chúng ta thêm d, do d nhỏ hơn e, chúng ta đi qua trái, so sánh d với b, chúng ta đi qua phải. Khi thêm f, chúng ta qua phải của e như hình (d). Để thêm a, chúng ta qua trái của e, rồi qua trái của b, do a là khóa nhỏ nhất trong các khóa cần thêm vào. Tương tự, khóa g là khóa lớn nhất trong các khóa cần thêm, chúng ta đi sang phải liên tục trong khi còn có thể, như hình (f). Cuối cùng, việc thêm c, so sánh với e, rẽ sang trái, so sánh với b, rẽ phải, và so sánh với d, rẽ trái. Chúng ta có được cây ở hình (g).

Hoàn toàn có thể có một thứ tự thêm vào khác cũng tạo ra một cây nhị phân tìm kiếm tương tự. Chẳng hạn, cây ở hình 9.10 có thể được tạo ra khi các khóa



Hình 9.10 – Thêm phần tử vào cây nhị phân tìm kiếm

được thêm theo thứ tự e, f, g, b, a, d, c hoặc e, b, d, c, a, f, g hoặc một số thứ tự khác.

Có một trường hợp thật đặc biệt. Giả sử các khóa được thêm vào một cây rỗng theo đúng thứ tự tự nhiên a, b, ..., g, thì cây nhị phân tìm kiếm được tạo ra sẽ là một chuỗi các mắc xích, như hình 9.9e. Chuỗi mắc xích như vậy rất kém hiệu quả đối với việc tìm kiếm. Chúng ta có kết luận sau:

Nếu các khóa được thêm vào một cây nhị phân tìm kiếm rỗng theo thứ tự tự nhiên của chúng, thì phương thức `insert` *sẽ sinh ra một cây suy thoái về một chuỗi mắc xích kém hiệu quả. Phương thức* `insert` *không nên dùng với các khóa đã có thứ tự.*

Kết quả trên cũng đúng trong trường hợp các khóa có thứ tự ngược hoặc gần như có thứ tự.

9.3.3.3. Phương pháp

Từ ví dụ trên đến phương thức `insert` tổng quát của chúng ta chỉ có một bước nhỏ.

Trong trường hợp thứ nhất, thêm một nút vào một cây rỗng rất dễ. Chúng ta chỉ cần cho con trỏ `root` chỉ đến nút này. Nếu cây không rỗng, chúng ta cần so sánh khóa của nút cần thêm với khóa của nút gốc. Nếu nhỏ hơn, nút mới cần thêm vào cây con trái, nếu lớn hơn, nút mới cần thêm vào cây con phải. Nếu hai khóa bằng nhau thì phương thức trả về `duplicate_error`.

Lưu ý rằng chúng ta vừa mô tả việc thêm vào bằng cách sử dụng đệ quy. Sau khi chúng ta so sánh khóa, chúng ta sẽ thêm nút mới vào cho cây con trái hoặc cây con phải theo đúng phương pháp mà chúng ta sử dụng cho nút gốc.

9.3.3.4. Hàm đệ quy

Giờ chúng ta đã có thể viết phương thức `insert`, phương thức này sẽ gọi hàm đệ quy phụ trợ với thông số `root`.

```
template <class Record>
Error_code Search_tree<Record>::insert(const Record &new_data)
{
   return search_and_insert(root, new_data);
}
```

Lưu ý rằng hàm phụ trợ cần thay đổi `sub_root`, đó là trường hợp việc thêm nút mới thành công. Do đó, thông số `sub_root` phải là tham chiếu.

```
template <class Record>
Error_code Search_tree<Record>::search_and_insert(
           Binary_node<Record> *&sub_root, const Record &new_data)
{
   if (sub_root == NULL) {
      sub_root = new Binary_node<Record>(new_data);
      return success;
   }
   else if (new_data < sub_root->data)
      return search_and_insert(sub_root->left, new_data);
   else if (new_data > sub_root->data)
      return search_and_insert(sub_root->right, new_data);
   else return duplicate_error;
}
```

Chúng ta đã quy ước cây nhị phân tìm kiếm sẽ không có hai phần tử trùng khóa, do đó hàm `search_and_insert` từ chối mọi phần tử có trùng khóa.

Sự sử dụng đệ quy trong phương thức `insert` thật ra không phải là bản chất, vì đây là đệ quy đuôi. Cách hiện thực không đệ quy được xem như bài tập.

Xét về tính hiệu quả, `insert` cũng thực hiện cùng một số lần so sánh các khóa như `tree_search` đã làm khi tìm một khóa đã thêm vào trước đó. Phương thức `insert` còn làm thêm một việc là thay đổi một con trỏ, nhưng không hề thực hiện việc di chuyển các phần tử hoặc bất cứ việc gì khác chiếm nhiều thời gian. Vì thế, hiệu quả của `insert` cũng giống như `tree_search`:

Phương thức `insert` có thể thêm một nút mới vào một cây nhị phân tìm kiếm ngẫu nhiên có n nút trong O(log n) bước. Có thể xảy ra, nhưng cực kỳ hiếm, một cây ngẫu nhiên trở nên suy thoái và làm cho việc thêm vào cần đến n bước. Nếu các khóa được thêm vào một cây rỗng mà đã có thứ tự thì trường hợp suy thoái này sẽ xảy ra.

9.3.4. Sắp thứ tự theo cây

Khi duyệt một cây nhị phân tìm kiếm theo *inorder* chúng ta sẽ có được các khóa theo đúng thứ tự của chúng. Lý do là vì tất cả các khóa bên trái của một khóa đều nhỏ hơn chính nó, và các khóa bên phải của nó đều lớn hơn nó. Bằng đệ quy, điều này cũng tiếp tục đúng với các cây con cho đến khi cây con chỉ còn là một nút. Vậy phép duyệt *inorder* luôn cho các khóa có thứ tự.

9.3.4.1. Thủ tục sắp thứ tự

Điều quan sát được trên là cơ sở cho một thủ tục sắp thứ tự thú vị được gọi là *treesort*. Chúng ta chỉ cần dùng phương thức `insert` để xây dựng một cây nhị phân tìm kiếm từ các phần tử cần sắp thứ tự, sau đó dùng phép duyệt *inorder* chúng ta sẽ có các phần tử có thứ tự.

9.3.4.2. So sánh với *quicksort*

Chúng ta sẽ xem thử số lần so sánh khóa của *treesort* là bao nhiêu. Nút đầu tiên là gốc của cây, không cần phải so sánh khóa. Với hai nút tiếp theo, khóa của chúng trước tiên cần so sánh với khóa của gốc để sau đó rẽ trái hoặc phải. *Quicksort* cũng tương tự, trong đó, ở bước thứ nhất mỗi khóa cần so sánh với phần tử *pivot* để được đặt vào danh sách con bên trái hoặc bên phải. Trong *treesort*, khi mỗi nút được thêm, nó sẽ dần đi tới vị trí cuối cùng của nó trong cấu trúc liên kết. Khi nút thứ hai trở thành nút gốc của cây con trái hoặc cây con phải, mọi nút thuộc một trong hai cây con này sẽ được so sánh với nút gốc của nó. Tương tự, trong *quicksort* mọi khóa trong một danh sách con được so sánh với phần tử *pivot* của nó. Tiếp tục theo cách tương tự, chúng ta có được nhận xét sau:

Treesort có cùng số lần so sánh các khóa với quicksort.

Như chúng ta đã biết, *quicksort* là một phương pháp rất tốt. Xét trung bình, trong các phương pháp mà chúng ta đã học, chỉ có *mergesort* là có số lần so sánh

các khóa ít nhất. Do đó chúng ta có thể hy vọng rằng *treesort* cũng là một phương pháp tốt nếu xét về số lần so sánh khóa. Từ phần 8.8.4 chúng ta có thể kết luận:

Trong trường hợp trung bình, trong một danh sách có thứ tự ngẫu nhiên có n phần tử, *treesort* thực hiện

$$2n \ln n + O(n) \approx 1.39 \lg n + O(n)$$

số lần so sánh.

Treesort còn có một ưu điểm so với *quicksort*. *Quicksort* cần truy xuất mọi phần tử trong suốt quá trình sắp thứ tự. Với *treesort*, khi bắt đầu quá trình, các phần tử không cần phải có sẵn một lúc, mà chúng được thêm vào cây từng phần tử một. Do đó *treesort* thích hợp với các ứng dụng mà trong đó các phần tử được nhận vào mỗi lúc một phần tử. **Ưu điểm lớn của *treesort* là cây nhị phân tìm kiếm vừa cho phép thêm hoặc loại phần tử đi sau đó, vừa cho phép tìm kiếm theo thời gian *logarit***. Trong khi tất cả các phương pháp sắp thứ tự trước kia của chúng ta, với hiện thực danh sách liên tục thì việc thêm hoặc loại phần tử rất khó, còn với danh sách liên kết, thì việc tìm kiếm chỉ có thể là tuần tự.

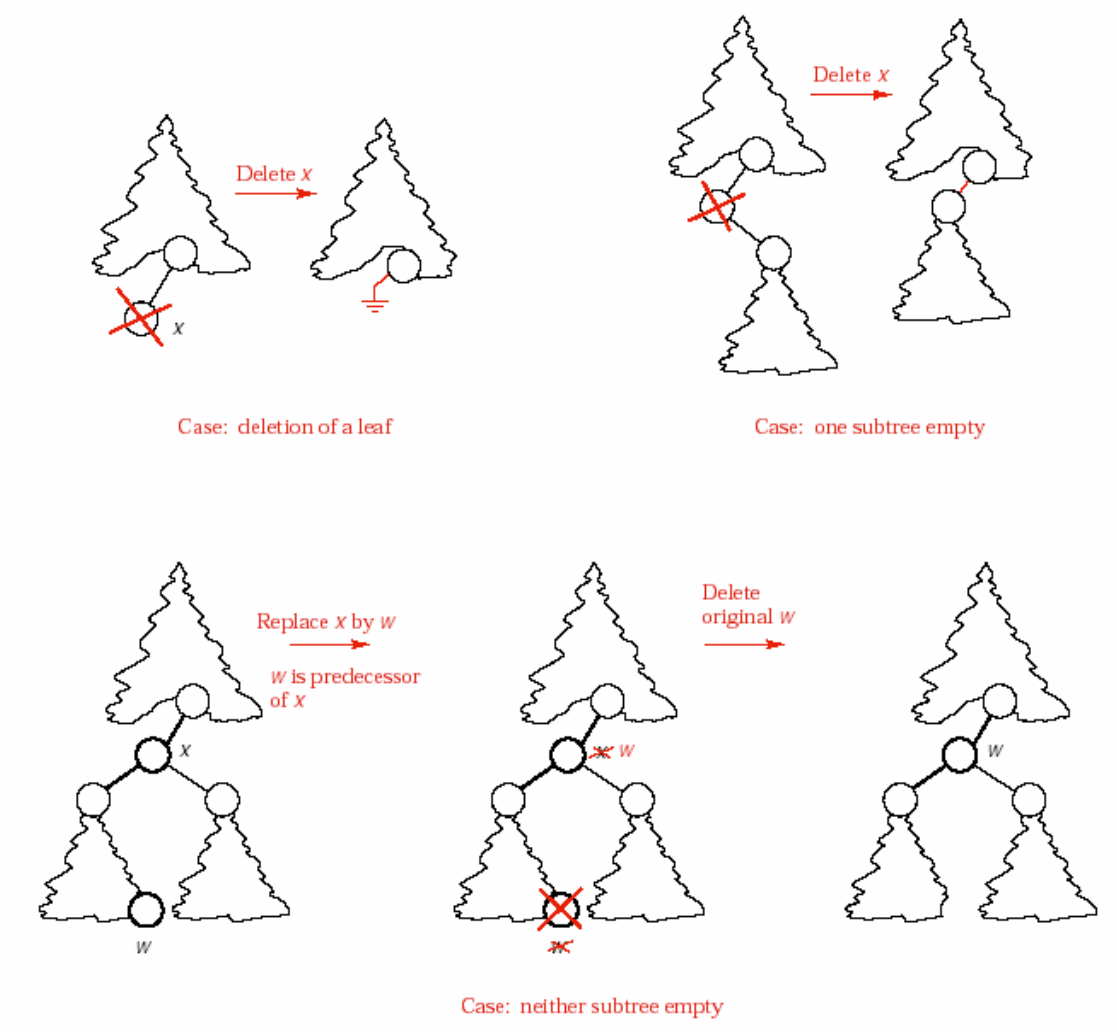
Nhược điểm chính của *treesort* được xem xét như sau. Chúng ta biết rằng *quicksort* có hiệu quả rất thấp trong trường hợp xấu nhất của nó, nhưng nếu phần tử *pivot* được chọn tốt thì trường hợp này cũng rất hiếm khi xảy ra. Khi chúng ta chọn phần tử đầu của mỗi danh sách con làm *pivot,* trường hợp xấu nhất là khi các khóa đã có thứ tự. Tương tự, nếu các khóa đã có thứ tự thì *treesort* sẽ trở nên rất dở, cây tìm kiếm sẽ suy thoái về một chuỗi các mắc xích. *Treesort* không bao giờ nên dùng với các khóa đã có thứ tự, hoặc gần như có thứ tự.

9.3.5. Loại phần tử trong cây nhị phân tìm kiếm

Khi xem xét về *treesort*, chúng ta đã nhắc đến khả năng thay đổi trong cây nhị phần tìm kiếm là một ưu điểm. Chúng ta cũng đã có một giải thuật thêm một nút vào một cây nhị phân tìm kiếm, và nó có thể được sử dụng cho cả trường hợp cập nhật lại cây cũng như trường hợp xây dựng cây từ đầu. Nhưng chúng ta chưa đề cập đến cách loại một phần tử ra khỏi cây. Nếu nút cần loại là một nút lá, thì công việc rất dễ: chỉ cần sửa tham chiếu đến nút cần loại thành `NULL` (sau khi đã giải phóng nút đó). Công việc cũng vẫn dễ dàng khi nút cần loại chỉ có một cây con khác rỗng: tham chiếu từ nút cha của nút cần loại được chỉ đến cây con khác rỗng đó.

Khi nút cần loại có đến hai cây con khác rỗng, vấn đề trở nên phức tạp hơn nhiều. Cây con nào sẽ được tham chiếu từ nút cha? Đối với cây con còn lại cần phải làm như thế nào? Hình 9.11 minh họa trường hợp này. Trước tiên, chúng ta

cần tìm nút ngay kế trước nút cần loại trong phép duyệt *inorder* (còn gọi là nút cực phải của cây con trái) bằng cách đi xuống nút con trái của nó và sau đó đi về bên phải liên tiếp nhiều lần cho đến khi không thể đi được nữa. Nút cực phải của cây con trái này sẽ không có nút con bên phải, cho nên nó có thể được loại đi một cách dễ dàng. Như vậy dữ liệu của nút cần loại sẽ được chép đè bởi dữ liệu của nút này, và nút này sẽ được loại đi. Bằng cách này cây vẫn còn giữ được đặc tính của cây nhị phân tìm kiếm, do giữa nút cần loại và nút ngay kế trước nó trong phép duyệt *inorder* không còn nút nào khác, và thứ tự duyệt *inorder* vẫn không bị xáo trộn. (Cũng có thể làm tương tự khi chọn để loại nút ngay kế sau của nút cần loại - nút cực trái của cây con phải - sau khi chép dữ liệu của nút này lên dữ liệu của nút cần loại).



Hình 9.11 – Loại một phần tử ra khỏi cây nhị phân tìm kiếm

Chúng ta bắt đầu bằng một hàm phụ trợ sẽ loại đi một nút nào đó trong cây nhị phân tìm kiếm. Hàm này có thông số là địa chỉ của nút cần loại. Thông số này phải là tham biến để việc thay đổi nó làm thay đổi thực sự con trỏ được gởi làm thông số. Ngoài ra, mục đích của hàm là cập nhật lại cây nên trong chương trình gọi, thông số thực sự phải là một

trong các tham chiếu đến chính một nút của cây, chứ không phải chỉ là một bản sao của nó. Nói một cách khác, nếu nút con trái của nút x cần bị loại thì hàm sẽ được gọi như sau

```
remove_root(x->left),
```

nếu chính `root` cần bị loại thì hàm sẽ gọi

```
remove_root(root).
```

Cách gọi sau đây không đúng do khi `y` thay đổi, `x->left` không hề thay đổi:

```
y = x->left; remove_root(y);
```

Hàm phụ trợ `remove_root` được hiện thực như sau:

```
template <class Record>
Error_code Search_tree<Record>::remove_root(Binary_node<Record>
                                                    *&sub_root)
/*
pre: sub_root là NULL, hoặc là địa chỉ của nút gốc của một cây con mà nút gốc này cần được
     loại khỏi cây nhị phân tìm kiếm.
post: Nếu sub_root là NULL, hàm trả về not_present. Ngược lại, gốc của cây con này sẽ
     được loại sao cho cây còn lại vẫn là cây nhị phân tìm kiếm. Thông số sub_root được gán
     lại gốc mới của cây con, hàm trả về success.
*/
{
   if (sub_root == NULL) return not_present;
   Binary_node<Record> *to_delete = sub_root;  // Nhớ lại nút cần loại.
   if (sub_root->right == NULL)
      sub_root = sub_root->left;
   else if (sub_root->left == NULL)
      sub_root = sub_root->right;
   else {                          // Cả 2 cây con đều rỗng.
      to_delete = sub_root->left;  // Về bên trái để đi tìm nút đứng ngay trước nút cần
                                   //    loại trong thứ tự duyệt inorder..

      Binary_node<Record> *parent = sub_root;
      while (to_delete->right != NULL) {     // to_delete sẽ đến được nút
         parent = to_delete;                 // cần tìm và parent sẽ là
         to_delete = to_delete->right;       // nút cha của nó.
      }
      sub_root->data = to_delete->data;      // Chép đè lên dữ liệu cần loại.

      if (parent == sub_root)                // Trường hợp đặc biệt: nút con
           sub_root->left = to_delete->left; // trái của nút cần loại cũng
                                             // chính là nút đứng ngay trước
                                             // nó trong thứ tự duyệt inorder.
      else parent->right = to_delete->left;
   }
   delete to_delete; // Loại phần tử cực phải của cây con trái của phần tử cần loại.
   return success;
}
```

Chúng ta cần phải cẩn thận phân biệt giữa trường hợp nút ngay trước nút cần loại trong thứ tự duyệt *inorder* là chính nút con trái của nó với trường hợp chúng ta cần phải di chuyển về bên phải để tìm. Trường hợp thứ nhất là trường hợp đặc biệt, nút con trái của nút cần loại có cây con bên phải rỗng. Trường hợp thứ hai là trường hợp tổng quát hơn, nhưng cũng cần lưu ý là chúng ta đi tìm nút có cây con phải là rỗng chứ không phải tìm một cây con rỗng.

Phương thức **remove** dưới đây nhận thông số là dữ liệu của nút cần loại chứ không phải con trỏ chỉ đến nó. Để loại một nút, việc đầu tiên cần làm là đi tìm nó trong cây. Chúng ta kết hợp việc tìm đệ quy trong cây với việc loại bỏ như sau:

```
template <class Record>
Error_code Search_tree<Record>::remove(const Record &target)
/*
post: Nếu tìm được dữ liệu có khóa trùng với khóa trong target thì dữ liệu đó sẽ bị loại khỏi
      cây sao cho cây vẫn là cây nhị phân tìm kiếm, phương thức trả về success. Ngược lại,
      phương thức trả về not_present.
uses: Hàm search_and_destroy
*/
{
   return search_and_destroy(root, target);
}
```

Như thường lệ, phương thức trên gọi một hàm đệ quy phụ trợ có thông số là con trỏ root.

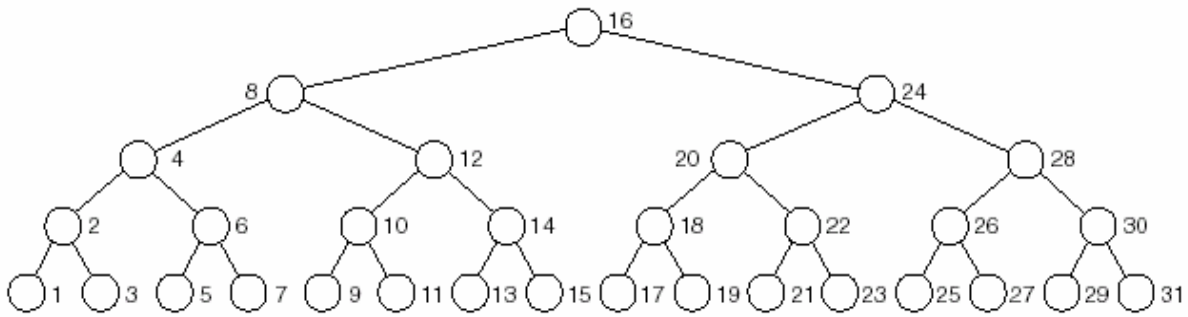
```
template <class Record>
Error_code Search_tree<Record>::search_and_destroy(
           Binary_node<Record>* &sub_root, const Record &target)
/*
pre: sub_root là NULL hoặc là địa chỉ của gốc của một cây con của cây nhị phân tìm kiếm.
post: Nếu khóa trong target không có trong cây con sub_root, hàm trả về not_present.
     Ngược lại, nút có chứa dữ liệu tìm thấy sẽ được loại sao cho tính chất cây nhị phân tìm kiếm
     vẫn đượcbảo toàn, hàm trả về success.
uses: Hàm search_and_destroy (một cách đệ quy) và hàm remove_root.
*/
{
   if (sub_root == NULL || sub_root->data == target)
      return remove_root(sub_root);
   else if (target < sub_root->data)
      return search_and_destroy(sub_root->left, target);
   else
      return search_and_destroy(sub_root->right, target);
}
```

9.4. Xây dựng một cây nhị phân tìm kiếm

Giả sử chúng ta có một danh sách các dữ liệu đã có thứ tự, hoặc có thể là một file các bản ghi có các khóa đã có thứ tự. Nếu chúng ta muốn sử dụng các dữ liệu

này để tìm kiếm thông tin, hoặc thực hiện một số thay đổi nào đó, chúng ta có thể tạo một cây nhị phân tìm kiếm từ danh sách hoặc file này.

Chúng ta có thể bắt đầu từ một cây rỗng và đơn giản sử dụng giải thuật thêm vào cây để thêm từng phần tử. Tuy nhiên do các phần tử đã có thứ tự, cây tìm kiếm của chúng ta sẽ trở thành một chuỗi các mắc xích rất dài, và việc sử dụng nó trở nên rất chậm chạp với tốc độ của tìm tuần tự chứ không phải là tìm nhị phân. Thay vào đó chúng ta mong muốn rằng các phần tử sẽ được xây dựng thành một cây càng rậm rạp càng tốt, có nghĩa là cây không bị cao quá, để giảm thời gian tạo cây cũng như thời gian tìm kiếm. Chẳng hạn, khi số phần tử n bằng 31, chúng ta muốn cây sẽ có được dạng như hình 9.12. Đây là cây có được sự cân bằng tốt nhất giữa các nhánh, và được gọi là cây nhị phân đầy đủ.



Hình 9.12 – Cây nhị phân đầy đủ với 31 nút.

Trong hình 9.12, các phần tử được đánh số theo thứ tự mà giá trị của chúng tăng dần. Đây cũng là thứ tự tự duyệt cây *inorder*, và cũng là thứ tự mà chúng sẽ được thêm vào cây theo giải thuật của chúng ta. Chúng ta cũng sẽ dùng các con số này như là nhãn của các nút trong cây. Nếu chúng ta xem xét kỹ sơ đồ trên, chúng ta có thể nhận thấy một đặc tính quan trọng của các nhãn. Các nhãn của các nút lá chỉ toàn số lẻ. Các nhãn của các nút ở mức trên các nút lá một bậc là 2, 6, 10, 14, 18, 22, 26, 30. Các số này đều gấp đôi một số lẻ, có nghĩa chúng đều là số chẵn, và chúng đều không chia hết cho 4. Trên mức cao hơn một bậc các nút có nhãn 4, 12, 20 và 28, đây là những con số chia hết cho 4, nhưng không chia hết cho 8. Cuối cùng, các nút ngay dưới nút gốc có nhãn là 8 và 24. và nút gốc có nhãn là 16.

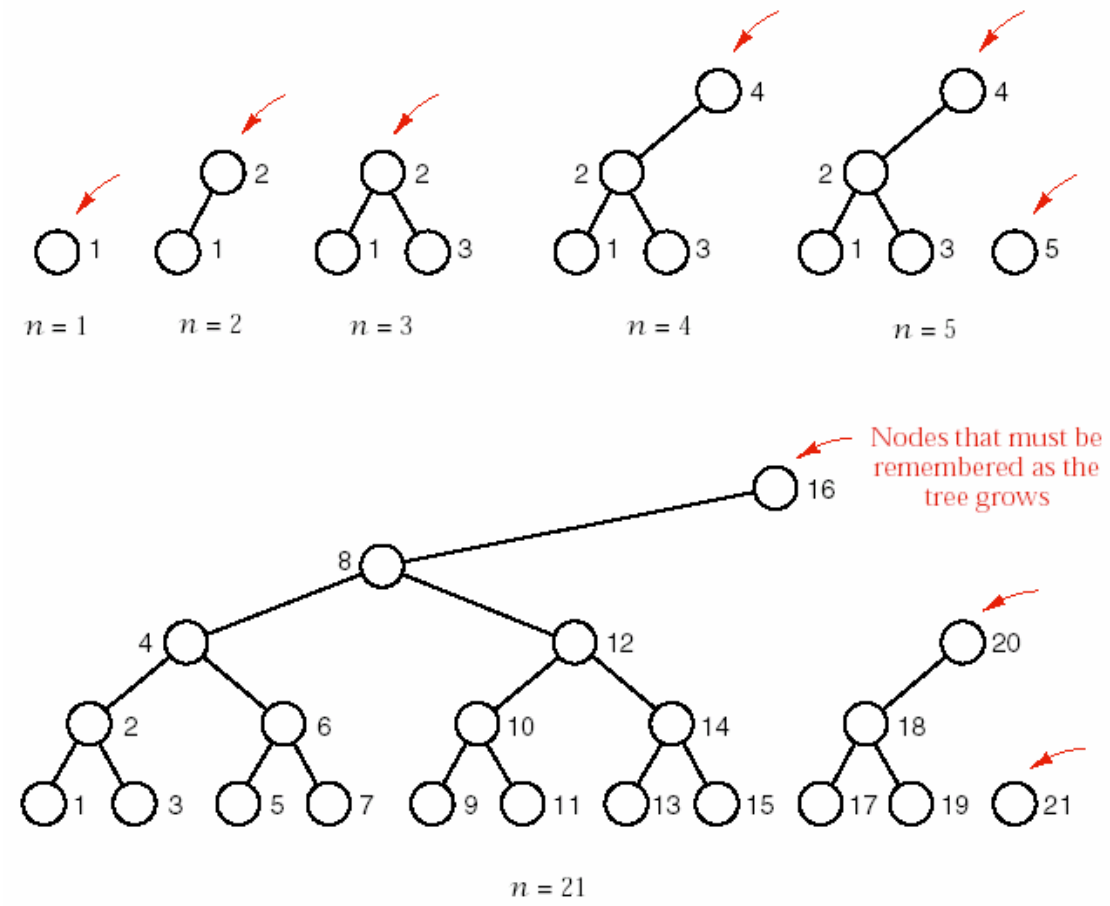
Nếu các nút của một cây nhị phân đầy đủ có các nhãn theo thứ tự duyệt inorder, bắt đầu từ 1, thì nhãn của mỗi nút là một số có số lần chia chẵn cho 2 bằng với hiệu giữa mức của nó với mức của các nút lá.

Như vậy, nếu cho mức của các nút lá là 1, khi thêm một nút mới, dựa vào nhãn của nó chúng ta sẽ tính được mức tương ứng.

Giả sử chúng ta không biết trước số nút sẽ tạo cây. Điều này được giải thích rằng, khi các nút đến từ một file hoặc một danh sách liên kết, chúng ta không có cách gì thuận tiện để đếm trước số nút cả. Điều giả thiết này cũng còn một ưu điểm là chúng ta không cần phải lo lắng về việc số nút có phải là một số lũy thừa của 2 trừ đi 1 hay không. Trong trường hợp số nút không phải là số lũy thừa của 2 trừ đi 1, cây được tạo ra sẽ không phải là một cây đầy đủ và đối xứng hoàn toàn như hình 9.12. Với giả thiết cây sẽ là một cây đầy đủ, chúng ta sẽ đưa dần các nút vào cây, cho đến khi mọi phần tử đã được đưa vào cây, chúng ta sẽ xác định cách để kết thúc việc tạo cây.

9.4.1. Thiết kế giải thuật

Khi nhận phần tử thứ nhất có nhãn là 1, chúng ta sẽ tạo một nút lá có các con trỏ left và right đều là `NULL`. Nút số 2 nằm trên nút 1, như hình 9.13. Do nút 2 nắm giữ nút 1, bằng cách nào đó chúng ta cần nhớ địa chỉ nút 1 cho đến khi có nút 2. Nút số 3 lại là nút lá, nhưng nó là nút con phải của nút 2, vậy chúng ta cần nhớ lại địa chỉ nút 2



Hình 9.13 – Tạo các nút đầu tiên cho một cây nhị phân tìm kiếm

Làm như vậy liệu chúng ta có cần phải nắm giữ một danh sách các con trỏ đến tất cả các nút đã được đưa vào cây, để sau đó chúng mới được gắn vào con trỏ

`left` hoặc `right` của cha chúng khi cha chúng xuất hiện sau hay không? Câu trả lời là không. Do khi nút 2 được thêm vào, mọi mối nối với nút 1 đã hoàn tất. Nút 2 cần được nhớ cho đến khi nút 4 được thêm vào, để tạo liên kết trái của nút 4. Tương tự, nút 4 cần được nhớ cho đến khi nút 8 được thêm vào. Trong hình 9.13, các mũi tên chỉ các nút cần được nhớ lại khi cây đang lớn lên.

Chúng ta thấy rằng để thực hiện các mối nối sau đó, đối với mỗi mức chúng ta chỉ cần nhớ lại duy nhất một con trỏ chỉ đến một nút, đó chính là nút cuối cùng trong mức đó. Chúng ta nắm giữ các con trỏ này trong một danh sách gọi là **`last_node`**, và danh sách này cũng sẽ rất nhỏ. Lấy ví dụ, một cây có 20 mức có thể chứa đến 2^{20}-1 > 1,000,000 nút, nhưng chỉ cần 20 phần tử trong `last_node`.

Khi một nút được thêm vào, chúng ta có thể gán con trỏ right của nó là `NULL`, có thể chỉ là tạm thời, vì nút con phải của nó (nếu có) sẽ được thêm vào sau. Con trỏ trái của một nút mới sẽ là `NULL` nếu đó là nút lá. Ngược lại, nút con trái của nó chính là nút ở ngay mức thấp hơn mức của nó mà địa chỉ đang chứa trong `last_node`. Chúng ta cũng có thể xử lý cho các nút lá tương tự như các nút khác bằng cách, nếu cho mức của nút lá là 1, thì chúng ta sẽ cho phần tử đầu tiên, tại vị trí 0, của `last_node` luôn mang trị `NULL`. Các mức bên trên mức của nút lá sẽ là 2, 3, ...

9.4.2. Các khai báo và hàm main

Chúng ta có thể định nghĩa một lớp mới, gọi là **`Buildable_tree`**, dẫn xuất từ lớp `Search_tree`.

```
template <class Record>
class Buildable_tree: public Search_tree<Record> {
public:
   Error_code build_tree(const List<Record> &supply);
 private: //  Các hàm phụ trợ.
};
```

Bước đầu tiên của `build_tree` là nhận các phần tử. Để đơn giản chúng ta sẽ cho rằng các phần tử này được chứa trong một danh sách các `Record` gọi là **`supply`**. Tuy nhiên, chúng ta cũng có thể viết lại hàm để nhận các phần tử từ một hàng đợi, hoặc một tập tin, hoặc thậm chí từ một cây nhị phân tìm kiếm khác với mong muốn cân bằng lại cây này.

Khi nhận các phần tử mới để thêm vào cây, chúng ta sẽ cập nhật lại một biến `count` để biết được có bao nhiêu phần tử đã được thêm vào. Rõ ràng là trị của `count` còn được dùng để lấy dữ liệu từ danh sách `supply`. Quan trọng hơn nữa, trị của `count` còn xác định mức của nút đang được thêm vào cây, nó sẽ được gởi cho một hàm chuyên lo việc tính toán này.

Sau khi tất cả các phần tử từ `supply` đã được thêm vào cây nhị phân tìm kiếm mới, chúng ta cần tìm gốc của cây và nối tất cả các cây con phải còn rời rạc.

```
template <class Record>
Error_code Buildable_tree<Record>::build_tree
                                    (const List<Record>&supply)
/*
post: Nếu dữ liệu trong supply có thứ tự tăng dần, cây nhị phân tìm kiếm khá cân bằng sẽ
      được tạo ra từ các dữ liệu này, phương thức trả về success. Ngược lại, cây nhị phân tìm
      kiếm chỉ được tạo ra từ mảng các dữ liệu tăng dần dài nhất tính bắt đầu từ đầu danh sách
      supply, phương thức trả về fail.
uses: Các phương thức của lớp List và các hàm build_insert, connect_subtrees, và
      find_root
*/
{  Error_code ordered_data = success;      // Sẽ được gán lại là fail nếu dữ liệu
                                              không tăng dần.
   int count = 0;
   Record x, last_x;
   List<Binary_node<Record> *> last_node; // Chỉ đến các nút cuối cùng của mỗi mức
                                              trong cây.
   Binary_node<Record> *none = NULL;
   last_node.insert(0, none); // luôn là NULL (dành cho các nút lá trong cây).
   while (supply.retrieve(count, x) == success) {
      if (count > 0 && x <= last_x) {
         ordered_data = fail;
         break;
      }
      build_insert(++count, x, last_node);
      last_x = x;
   }
   root = find_root(last_node);
   connect_trees(last_node);
   return ordered_data;
}
```

9.4.3. Thêm một nút

Phần trên đã xem xét cách nối các liên kết trái của các nút, nhưng trong quá trình xây dựng cây, các liên kết phải của các nút có lúc vẫn còn là `NULL` và chúng cần được thay đổi sau đó. Khi một nút mới được thêm vào, nó có thể sẽ có một cây con phải không rỗng. Do nó là nút mới nhất và lớn nhất được đưa vào cây cho đến thời điểm hiện tại, các nút trong cây con phải của nó chưa có. Ngược lại, một nút mới được thêm cũng có thể là nút con phải của một nút nào đó đã được đưa vào trước. Đồng thời, nó có thể là con trái của một nút có khóa lớn hơn, trong trường hợp này thì nút cha của nó chưa có. Chúng ta có thể xác định từng trường hợp đang xảy ra nhờ vào danh sách `last_node`. Nếu mức **`level`** của một nút đang được thêm vào lớn hơn hoặc bằng 1, thì nút cha của nó có mức **`level+1`**. Chúng ta tìm phần tử thứ `level+1` trong `last_node`. Nếu con trỏ `right` của nút này vẫn còn là `NULL` thì nút đang được thêm mới chính là nút con phải của nó.

Ngược lại, nếu nó đã có con phải thì nút đang được thêm mới phải là con trái của một nút nào đó sẽ được thêm vào sau. Xem hình 9.1.

Chúng ta có thể viết hàm thêm một nút mới vào cây như sau:

```
template <class Record>
void Buildable_tree<Record>::build_insert(int count,
                                  const Record &new_data,
                                  List<Binary_node<Record>*> &last_node)
/*
post: Một nút mới chứa new_data được thêm vào cây. Mức của nút này bằng số lần count chia
      chẵn cho 2 cộng thêm 1, nếu xem mức của nút lá bằng 1.
uses: Các phương thức của lớp List.
*/
{
   int level;
   for (level = 1; count % 2 == 0; level++)
      count /= 2;    // Sử dụng count để tính mức của nút kế.
   Binary_node<Record> *next_node = new Binary_node<Record>(new_data),
                       *parent;
   last_node.retrieve(level - 1, next_node->left); // Nút mới được tạo ra
                           // nhận con trái chính là nút có địa chỉ đang được lưu
                           // trong last_node tại phần tử tương ứng với  mức
                           // ngay dưới mức của nút mới.
   if (last_node.size() <= level)
      last_node.insert(level, next_node); // Nút mới là nút đầu tiên xuất hiện
                                          // trong mức của nó.
   else
      last_node.replace(level, next_node);// Đã có nhiều nút cùng mức và nút
                                          // mới này chính là nút xuất hiện
                                          // mới nhất trong các nút cùng mức
                                          // nên cần lưu lại địa chỉ.
   if (last_node.retrieve(level + 1, parent)==success
                                           && parent->right==NULL)
                                          // Đây là trường hợp nút cha của
                                          // nút mới đã tồn tại và nút cha này
                                          // sẽ nhận nó làm con phải.
      parent->right = next_node;
}
```

9.4.4. Hoàn tất công việc

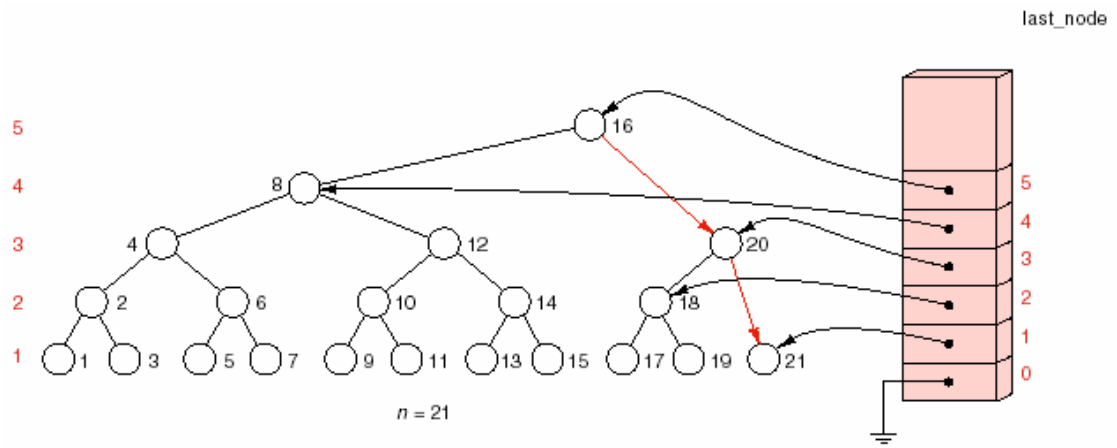
Tìm gốc của cây vừa được tạo là một việc dễ dàng: gốc chính là nút ở mức cao nhất trong cây, con trỏ chỉ đến nó chính là phần tử cuối cùng trong danh sách last_node. Cây có 21 nút như hình 9.13 có nút cao nhất là nút 16 ở mức 5, đó chính gốc của cây. Các con trỏ đến các nút cuối của mỗi mức được chứa trong last_node như hình vẽ 9.14.

Chúng ta có hàm như sau:

```
template <class Record>
Binary_node<Record> *Buildable_tree<Record>::find_root
                                (List<Binary_node<Record>*> &last_node)
/*
pre: Danh sách last_node chứa địa chỉ các nút cuối cùng của mỗi mức trong cây nhị phân tìm
     kiếm.
post: Trả về địa chỉ nút gốc của cây nhị phân tìm kiếm đã được tạo ra.
uses: Các phương thức của lớp List.
*/
{
   Binary_node<Record> *high_node;
   last_node.retrieve(last_node.size() - 1, high_node);
      // Tìm địa chỉ nút gốc của cây tại phần tử tương ứng với mức cao nhất trong cây.
   return high_node;
}
```

Cuối cùng, chúng ta cần xác định cách nối các cây con còn nằm ngoài. Chẳng hạn, với cây có n = 21, chúng ta cần nối ba thành phần ở hình 9.13 vào một cây duy nhất. Theo hình vẽ chúng ta thấy rằng một số nút trong các phần trên của cây có thể vẫn còn con trỏ right là NULL, trong khi các nút đã thêm vào cây bây giờ phải trở thành nút con phải của chúng.

Bất kỳ nút nào trong số các nút có con trỏ right vẫn còn là NULL, trừ nút lá, đều là một trong các nút nằm trong last_node. Với n=21, đó là các nút 16 và 20



Hình 9.14 – Hoàn tất cây nhị phân tìm kiếm.

tại các vị trí 5 và 3 tương ứng trong last_node trong hình 9.14.

Trong hàm sau đây chúng ta dùng con trỏ **high_node** để chỉ đến các nút có con trỏ right là NULL. Chúng ta cần xác định con trỏ **lower_node** chỉ đến nút con phải của **high_node**. Con trỏ lower_node có thể được xác định bởi nút cao nhất trong last_node mà không phải là nút con trái của high_node**.** Để xác định một

nút có phải là con trái của high_node hay không chúng ta so sánh khóa của nó với khóa của high_node.

```
template <class Record>
void Buildable_tree<Record>::connect_trees
                          (const List<Binary_node<Record>*> &last_node)
/*
pre: Danh sách last_node chứa địa chỉ các nút cuối cùng của mỗi mức trong cây nhị phân tìm
     kiếm. Cây nhị phân tìm kiếm được đã được tạo gần hoàn chỉnh.
post: Các liên kết cuối cùng trong cây được nối lại.
uses: Các phương thức của lớp List.
*/
{  Binary_node<Record> *high_node, //  from last_node with NULL right child
                       *low_node;  //  candidate for right child of high_node
   int high_level = last_node.size() - 1,
       low_level;
   while (high_level > 2) {        //  Nodes on levels 1 and 2 are already OK.
      last_node.retrieve(high_level, high_node);
      if (high_node->right != NULL)
         high_level--;             //  Search down for highest dangling node.
      else {                       //  Case: undefined right tree
         low_level = high_level;
         do {                      //  Find the highest entry not in the left
                                                              subtree.
            last_node.retrieve(--low_level, low_node);
         }while (low_node != NULL && low_node->data < high_node->data);
         high_node->right = low_node;
         high_level = low_level;
      }
   }
}
```

9.4.5. Đánh giá

Cây nhị phân tìm kiếm do giải thuật trên đây tạo ra không luôn là một cây cân bằng tốt nhất. Như chúng ta thấy, hình 9.14 là cây có n = 21 nút. Nếu nó có 31 nút, nó mới có sự cân bằng tốt nhất. Nếu nút thứ 32 được thêm vào thì nó sẽ trở thành gốc của cây, và tất cả 31 nút đã có sẽ thuộc cây con trái của nó. Trong trường hợp này, các nút lá nằm cách nút gốc 5 bước tìm kiếm. Như vậy cây có 32 nút thường sẽ phải cần số lần so sánh nhiều hơn số lần so sánh cần thiết là một. Với cây có 32 nút, nếu nút gốc được chọn một cách tối ưu, thì đa số các nút lá cần 4 bước tìm kiếm từ nút gốc, chỉ có một nút lá là cần 5 bước.

Một lần so sánh dôi ra trong tìm nhị phân không phải là một sự trả giá cao, và rõ ràng rằng cây được tạo ra bởi phương pháp trên đây của chúng ta sẽ không bao giờ có số mức nhiều hơn số mức tối ưu quá một đơn vị. Còn có nhiều phương pháp phức tạp hơn để tạo ra một cây nhị phân tìm kiếm đạt được sự cân bằng cao nhất có thể, nhưng một phương pháp đơn giản như trên đây cũng rất cần thiết, đặc biệt là phương pháp này **không cần biết trước số nút sẽ được thêm vào cây.**

Trong phần 9.5 chúng ta sẽ tìm hiểu về cây AVL, trong đó việc thêm hay loại phần tử luôn bảo đảm cây vẫn gần với trạng thái cân bằng. Tuy nhiên, đối với nhiều ứng dụng, giải thuật đơn giản mà chúng ta mô tả ở đây đã là thích hợp.

9.5. Cân bằng chiều cao: Cây AVL

Giải thuật trong phần 9.4 có thể được sử dụng để xây dựng một cây nhị phân tìm kiếm gần như cân bằng, hoặc để khôi phục sự cân bằng. Tuy nhiên, trong nhiều ứng dụng, việc thêm và loại phần tử trong cây xảy ra thường xuyên, và với một thứ tự không biết trước. Trong một vài ứng dụng loại này, điều quan trọng cần có là tối ưu thời gian tìm kiếm bằng cách luôn duy trì cây gần với tình trạng cân bằng. Từ năm 1962 hai nhà toán học người Nga, G. M. Adel'Son-Vel'Skil và E. M. Landis, đã mô tả một phương pháp nhằm đáp ứng yêu cầu này, và cây nhị phân tìm kiếm này được gọi là cây AVL.

Cây AVL đạt được mục đích là việc tìm kiếm, thêm vào, loại bỏ phần tử trong một cây n nút có được thời gian là O(log n), ngay cả trong trường hợp xấu nhất. Chiều cao của cây AVL n nút, mà chúng ta sẽ xem xét sau, sẽ không bao giờ vượt quá 1.44lg n, và như vậy ngay cả trong trường hợp xấu nhất, các hành vi trong cây AVL cũng không thể chậm hơn một cây nhị phân tìm kiếm ngẫu nhiên. Tuy nhiên, trong phần lớn các trường hợp, thời gian tìm kiếm thật sự thường rất gần với lg n, và như vậy hành vi của các cây AVL rất gần với hành vi của một cây nhị phân tìm kiếm cân bằng hoàn toàn lý tưởng.

9.5.1. Định nghĩa

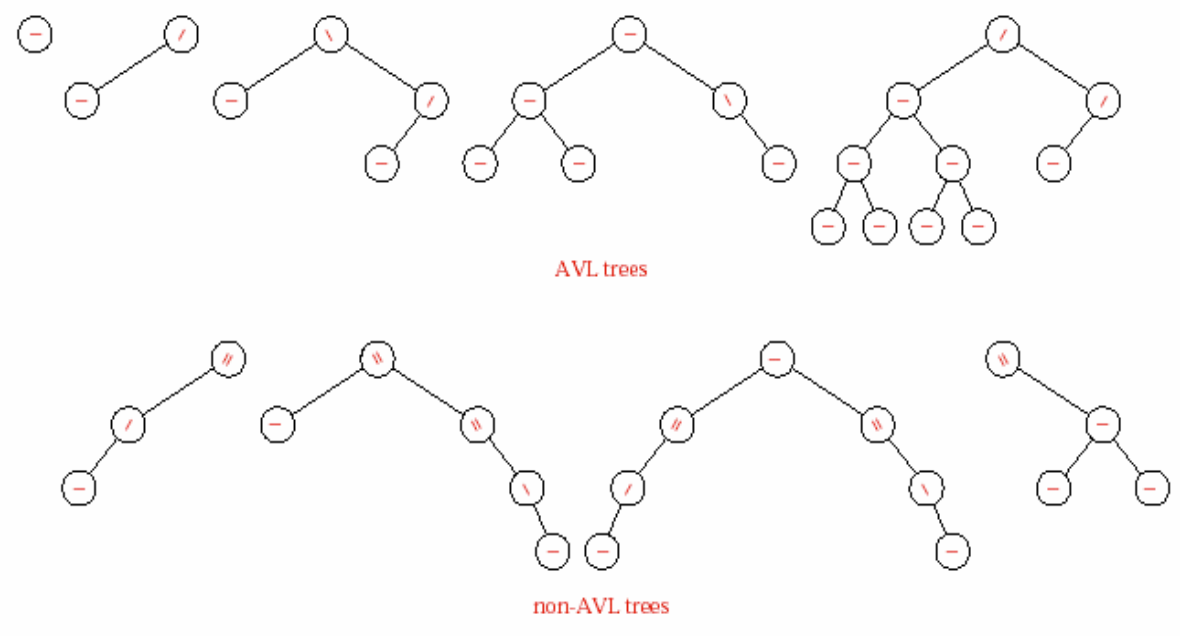
Trong một cây cân bằng hoàn toàn, các cây con trái và cây con phải của bất kỳ một nút nào cũng phải có cùng chiều cao. Mặc dù chúng ta không thể luôn luôn đạt được điều này, nhưng bằng cách xây dựng cây nhị phân tìm kiếm một cách cẩn thận, chúng ta luôn có thể bảo đảm rằng chiều cao của cây con trái và chiều cao của cây con phải của bất kỳ một nút nào đều hơn kém nhau không quá 1. Chúng ta có định nghĩa sau:

> *Định nghĩa*: Cây AVL là một cây nhị phân tìm kiếm trong đó chiều cao cây con trái và chiều cao cây con phải của nút gốc hơn kém nhau không quá 1, và cả hai cây con trái và phải này đều là cây AVL.

Mỗi nút của cây AVL có thêm một thông số cân bằng mang trị `left-higher`, `equal-height`, hoặc `right-higher` tương ứng trường hợp cây con trái cao hơn, bằng, hoặc thấp hơn cây con phải.

Trong sơ đồ, chúng ta dùng '/' để chỉ nút có cây con trái cao hơn cây con phải, '-' chỉ nút cân bằng có hai cây con cao bằng nhau, và '\' chỉ nút có cây con phải cao hơn cây con trái. Hình 9.15 minh họa một vài cây AVL nhỏ và một số cây nhị phân không thỏa định nghĩa cây AVL.

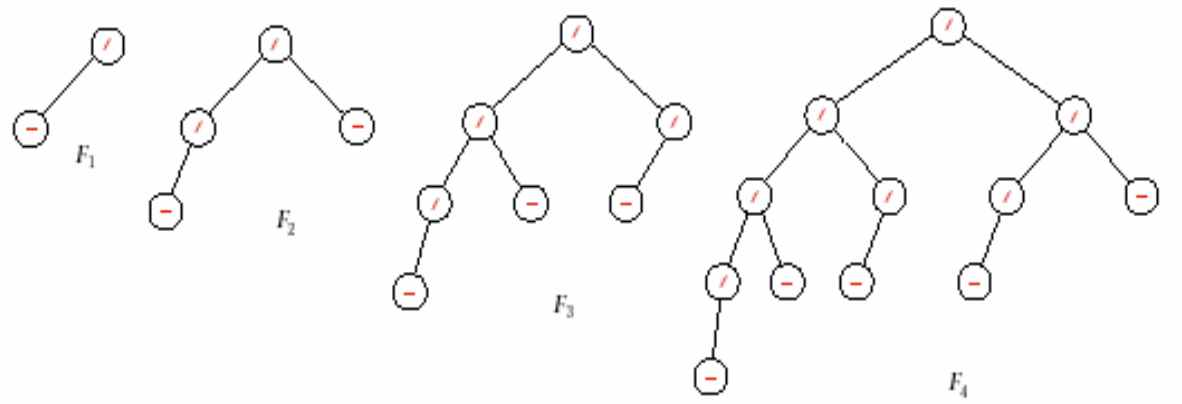
Lưu ý rằng, định nghĩa không yêu cầu mọi nút lá có cùng mức hoặc phải ở các mức kế nhau. Hình 9.16 minh họa một vài cây AVL rất không đối xứng, với các cây con trái cao hơn các cây con phải.



Hình 9.15 - Các ví dụ về cây AVL và các cây nhị phân khác.

Chúng ta dùng kiểu liệt kê để khai báo thông số cân bằng:

```
enum Balance_factor {left_higher,equal_height,right_higher};
```



Hình 9.16 – Một số cây AVL không đối xứng với cây con trái cao hơn cây con phải.

Thông số cân bằng phải có trong tất cả các nút của cây AVL, chúng ta cần bổ sung đặc tả một node như sau:

```
template <class Record>
struct AVL_node: public Binary_node<Record> {
   Balance_factor balance;
// constructors:
   AVL_node();
   AVL_node(const Record &x);
// Các hàm ảo được định nghĩa lại:
   void set_balance(Balance_factor b);
   Balance_factor get_balance() const;
};
```

Một điểm cần lưu ý trong đặc tả này là các con trỏ left và right của Binary_node có kiểu Binary_node*. Do đó các con trỏ mà AVL_node thừa kế cũng có kiểu này. Tuy nhiên, trong một cây AVL, rõ ràng là chúng ta cần các nút của cây AVL tham chiếu đến các nút khác của cây AVL. Sự không tương thích về kiểu của con trỏ này không phải là một vấn đề nghiêm trọng, vì một con trỏ đến một đối tượng của một lớp cơ sở cũng có thể chỉ đến một đối tượng của lớp dẫn xuất. Trong trường hợp của chúng ta, các con trỏ left và rigth của một AVL_node có thể chỉ đến các AVL_node khác một cách hợp lệ. Lợi ích của việc hiện thực AVL_node thừa kế từ Binary_node là sự sử dụng lại tất cả các phương thức của cây nhị phân và cây tìm kiếm. Tuy nhiên, chúng ta cần bảo đảm rằng khi thêm một nút mới vào cây AVL, chúng ta chỉ thêm đúng các nút AVL mà thôi.

Chúng ta sẽ sử dụng các phương thức get_balance và set_balance để xác định thông số cân bằng của AVL_node.

```
template <class Record>
void AVL_node<Record>::set_balance(Balance_factor b)
{
   balance = b;
}
```

```
template <class Record>
Balance_factor AVL_node<Record>::get_balance() const
{
   return balance;
}
```

Chúng ta sẽ gọi hai phương thức này thông qua con trỏ chỉ đến nút của cây, chẳng hạn left->get_balance(). Tuy nhiên, cách gọi này có thể gặp vấn đề đối với trình biên dịch C++ do left là con trỏ chỉ đến một Binary_node chứ không phải AVL_node. Trình biên dịch phải từ chối biểu thức left->get_balance() do nó không chắc rằng đó có là phương thức của đối tượng *left hay không. Chúng ta giải quyết vấn đề này bằng cách thêm vào các phiên bản giả (*dummy version*) của các phương thức get_balance() và set_balance() cho cấu trúc Binary_node. Các phương thức giả được thêm vào này chỉ để dành cho các hiện thực của cây AVL dẫn xuất.

Sau khi chúng ta bổ sung các phương thức giả cho cấu trúc Binary_node, trình biên dịch sẽ chấp nhận biểu thức left->set_balance(). Tuy nhiên, vẫn còn một vấn đề mà trình biên dịch vẫn không thể giải quyết được: nó nên sử dụng phương thức của AVL_node hay phương thức giả? Sự lựa chọn đúng chỉ có thể được thực hiện trong thời gian chạy của chương trình, khi kiểu của *left đã được biết. Một cách tương ứng, chúng ta phải khai báo các phiên bản set_balance và get_balance của Binary_node là các phương thức ảo (*virtual*). Điều này có nghĩa là sự lựa chọn phiên bản nào sẽ được thực hiện trong thời gian chạy của chương trình. Chẳng hạn, nếu

`set_balance()` được gọi như một phương thức của `AVL_node`, thì phiên bản của AVL sẽ được sử dụng, nếu nó được gọi như một phương thức của `Binary_node` thì phiên bản giả sẽ được sử dụng.

Dưới đây là đặc tả của `Binary_node` đã được sửa đổi:

```
template <class Entry>
struct Binary_node {
//    data members:
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
//    constructors:
   Binary_node();
   Binary_node(const Entry &x);
//    virtual methods:
   virtual void set_balance(Balance_factor b);
   virtual Balance_factor get_balance() const;
};
```

```
template <class Entry>
void Binary_node<Entry>::set_balance(Balance_factor b)
{
}
```

```
template <class Entry>
Balance_factor Binary_node<Entry>::get_balance() const
{
   return equal_height;
}
```

Ngoài ra không có sự thay đổi nào khác trong các lớp và các phương thức của chúng ta, và mọi hàm xử lý cho các nút trước kia của chúng ta bây giờ đã có thể sử dụng cho các nút AVL.

Bây giờ chúng ta đã có thể đặc tả lớp cho cây AVL. Chúng ta chỉ cần định nghĩa lại các phương thức thêm và loại phần tử để có thể duy trì cấu trúc cây cân bằng. Các phương thức khác của cây nhị phân tìm kiếm đều có thể được thừa kế mà không cần thay đổi gì.

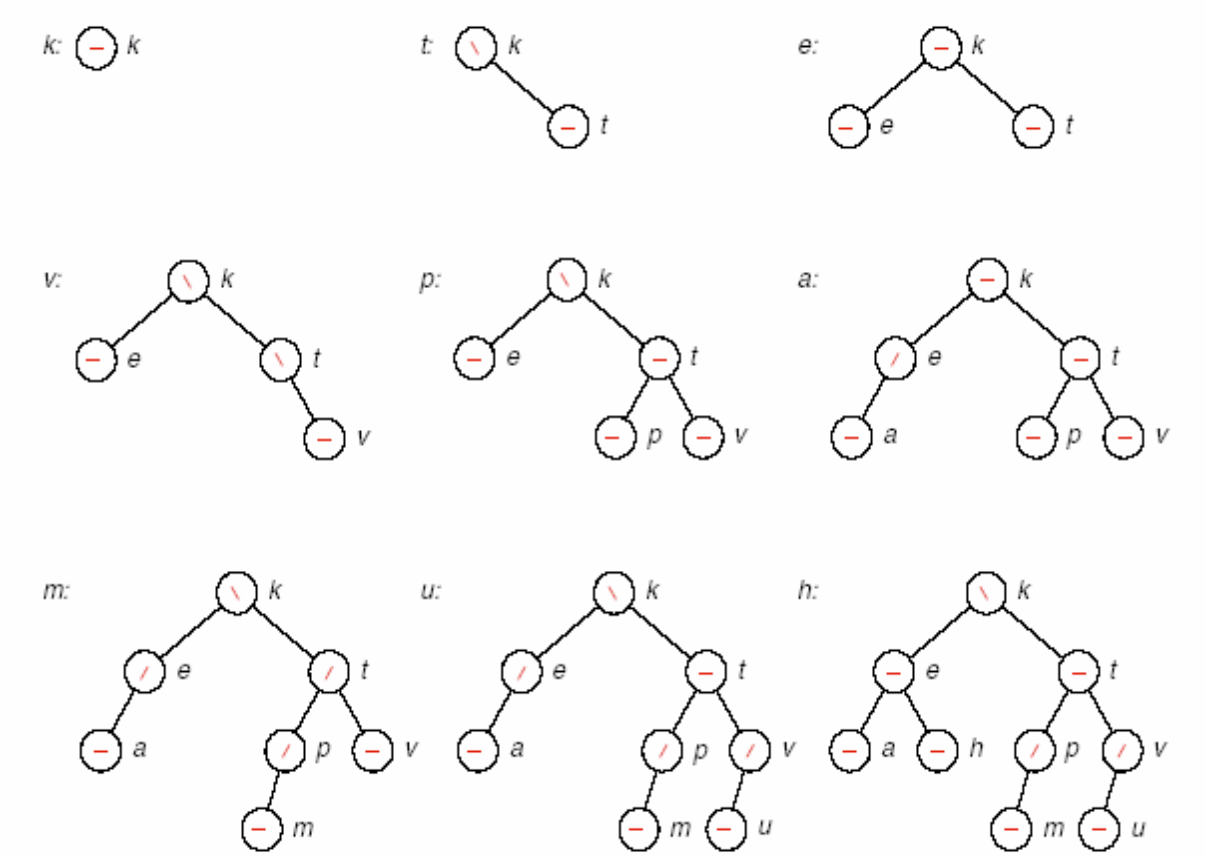
```
template <class Record>
class AVL_tree: public Search_tree<Record> {
public:
   Error_code insert(const Record &new_data);
   Error_code remove(const Record &old_data);
 private: //  Các hàm phụ trợ.
};
```

Thuộc tính dữ liệu lớp này thừa kế được là con trỏ root. Con trỏ này có kiểu `Binary_node<Record>*` và do đó, như chúng ta đã thấy, nó có thể chứa địa chỉ của một nút của cây nhị phân nguyên thủy hoặc địa chỉ của một nút của cây AVL. Chúng ta phải bảo đảm rằng phương thức `insert` được định nghĩa lại chỉ tạo ra các nút có kiểu `AVL_node`, làm như vậy mới bảo đảm rằng mọi nút được truy xuất thông qua con trỏ `root` của cây AVL đều là các nút AVL.

9.5.2. Thêm một nút

9.5.2.1. Dẫn nhập

Chúng ta hãy theo dõi sự lớn lên của cây AVL trong hình 9.17 qua việc thêm một số phần tử.



Hình 9.17 – Thêm nút vào cây AVL: các trường hợp đơn giản

Quá trình thêm ở hình 9.17 được tiến hành chính xác theo cùng một cách với quá trình thêm vào cây nhị phân tìm kiếm thông thường, ngoại trừ việc cập nhật lại thông số cân bằng. Tuy nhiên, chúng ta lưu ý rằng các thông số cân bằng chỉ có thể được xác định sau khi việc thêm vào đã được thực hiện. Khi v được thêm vào theo hình 9.17, thông số cân bằng trong gốc k thay đổi, nhưng nó không thay đổi khi p được thêm kế sau đó. Cả v và p đều được thêm vào cây con phải t của nút gốc, và chỉ sau khi việc thêm vào hoàn tất, thông số cân bằng tại nút gốc k mới có thể được xác định.

Chúng ta có thể thêm một nút mới vào một cây AVL bằng cách như sau. Trước hết theo đúng giải thuật thêm vào một cây nhị phân tìm kiếm bình thường : so sánh khóa của nút mới với khóa của nút gốc, và thêm nút mới vào cây con trái hoặc cây con phải thích hợp.

Trong quá trình lần tìm xuống các nhánh, nếu khóa cần thêm vào chưa có thì việc thêm nút mới cuối cùng sẽ được thực hiện tại một cây con rỗng. Từ cây con rỗng trở thành cây con có một nút, chiều cao nó đã tăng lên. Điều này có thể ảnh hưởng đến các nút trên đường đi từ nút cha của nó trở lên gốc. Vì vậy trong quá trình duyệt cây đi xuống để tìm vị trí thêm vào, chúng ta cần lưu lại vết để có thể lần ngược về để xử lý. Chúng ta có thể dùng ngăn xếp, hoặc đơn giản hơn là dùng hàm đệ quy.

Tham biến `taller` của hàm đệ quy được gán bằng `true` tại lần đệ quy cuối cùng, đó chính là lúc nút mới được tạo ra tại một cây con rỗng làm cho nó tăng chiều cao như đã nói ở trên. Những việc sau đây cần được thực hiện tại mỗi nút nằm trên đường đi từ nút cha của nút vừa được tạo ra cho đến nút gốc của cây. Trong khi mà tham biến `taller` trả lên còn là `true`, việc giải quyết cứ phải tiếp tục cho đến khi có một nút nhận được trị trả về của tham biến này là `false`. Một khi có một cây con nào đó báo lên rằng chiều cao của nó không bị tăng lên thì các nút trên của nó sẽ không bị ảnh hưởng gì, giải thuật kết thúc. Ngược lại, gọi nút đang được xử lý trong hàm đệ quy là `sub_root`. Sau khi gọi đệ quy xuống cây con và nhận được thông báo trả về rằng chiều cao cây con có tăng (`taller == true`), cần xét các trường hợp sau:

1. Cây con tăng chiều cao vốn là cây con thấp hơn cây con còn lại, thì chỉ có thông số cân bằng trong `sub_root` là cần thay đổi, cây có gốc là `sub_root` cũng không thay đổi chiều cao. Tham biến `taller` được gán về `false` bắt đầu từ đây.

2. Trước khi một cây con báo lên là tăng chiều cao thì hai cây con vốn cao bằng nhau. Trường hợp này cần thay đổi thông số cân bằng trong `sub_root`, đồng thời cây có gốc là `sub_root` cũng cao lên. Tham biến `taller` vẫn giữ nguyên là `true` để nút trên của `sub_root` giải quyết tiếp.

3. Cây con tăng chiều cao vốn là cây con cao hơn cây con còn lại. Khi đó `sub_root` sẽ có hai cây con có chiều cao chênh lệch là 2, không thỏa định nghĩa cây AVL. Đây là trường hợp chúng ta cần đến hàm `right_balance` hoặc `left_balance` để cân bằng lại cây có gốc là `sub_root` này. Chúng ta sẽ nghiên cứu nhiệm vụ của hai hàm này sau, nhưng tại đây cũng có thể nói trước rằng việc cân bằng lại luôn giải quyết được triệt để. Có nghĩa là cây có gốc là `sub_root` sẽ không bị tăng chiều cao, và như vậy, tham biến `taller` cũng được gán về `false` bắt đầu từ đây.

Với các quyết định trên, chúng ta có thể viết các phương thức và các hàm phụ trợ để thêm một nút mới vào cây AVL.

```
template <class Record>
Error_code AVL_tree<Record>::insert(const Record &new_data)
/*
post: Nếu khóa trong new_data đã có trong cây, phương thức trả về duplicate_error. Ngược
      lại, new_data được thêm vào cây sao cho cây vẫn thoả cây AVL, phương thức trả về
      success.
uses: Hàm avl_insert.
*/
{  bool taller;
   return avl_insert(root, new_data, taller);
}
```

```
template <class Record>
Error_code AVL_tree<Record>::avl_insert(Binary_node<Record> *&sub_root,
           const Record &new_data, bool &taller)
/*
pre: sub_root là NULL hoặc là gốc một cây con trong cây AVL.
post: Nếu khóa trong new_data đã có trong cây, phương thức trả về duplicate_error. Ngược
      lại, new_data được thêm vào cây sao cho cây vẫn thoả cây AVL, phương thức trả về
      success. Nếu cây con có tăng chiều cao, thông số taller được gán trị true, ngược lại
      nó được gán trị false.
uses: Các phương thức của AVL_node; hàm avl_insert một cách đệ quy, các hàm
      left_balance và right_balance.
*/
{  Error_code result = success;
   if (sub_root == NULL) {
      sub_root = new AVL_node<Record>(new_data);
      taller = true;
   }

   else if (new_data == sub_root->data) {
      result = duplicate_error;
      taller = false;
   }

   else if (new_data < sub_root->data) {     //  Thêm vào cây con trái.
      result = avl_insert(sub_root->left, new_data, taller);
      if (taller == true)
         switch (sub_root->get_balance()) { //  Change balance factors.

         case left_higher:
            left_balance(sub_root);
            taller = false;    // Cân bằng lại luôn làm cho cây không bị cao lên.
            break;

         case equal_height:
            sub_root->set_balance(left_higher); // Cây con có gốc sub_root
                                                // đã bị cao lên, taller vẫn là true.
            break;

         case right_higher:
            sub_root->set_balance(equal_height);
            taller = false; // Cây con có gốc sub_root không bị cao lên.
            break;
         }
   }
```

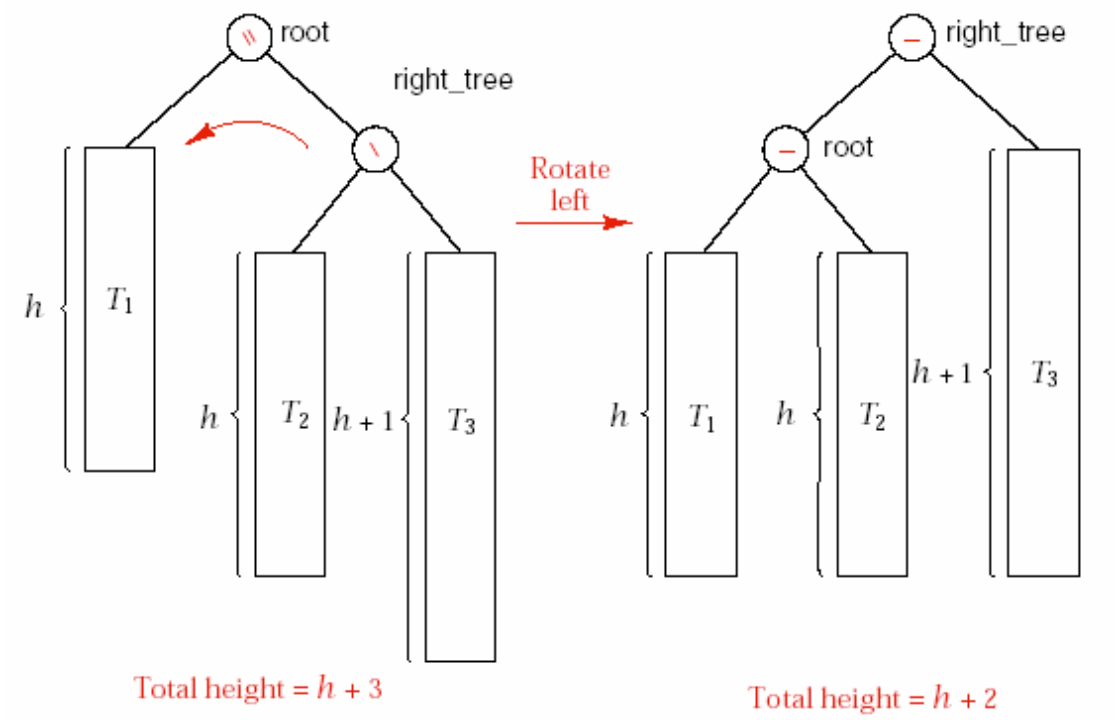
```
   else {                                        //  Thêm vào cây con phải.
      result = avl_insert(sub_root->right, new_data, taller);
      if (taller == true)
         switch (sub_root->get_balance()) {

         case left_higher:
            sub_root->set_balance(equal_height);
            taller = false;       // Cây con có gốc sub_root không bị cao lên.

            break;

         case equal_height:
            sub_root->set_balance(right_higher);// Cây con có gốc sub_root // đã
                                          bị cao lên, taller  vẫn là true.
            break;

         case right_higher:
            right_balance(sub_root);
            taller = false;    // Cân bằng lại luôn làm cho cây không bị cao lên.
            break;
         }
   }
   return result;
}
```

9.5.2.2. Các phép quay

Chúng ta hãy xét trường hợp nút mới được thêm vào cây con cao hơn và chiều cao của nó tăng lên, chênh lệch chiều cao hai cây con trở thành 2, và cây không còn thoả điều kiện của cây AVL. Chúng ta cần tổ chức lại một phần của cây để khôi phục lại sự cân bằng. Hàm **`right_balance`** sẽ xem xét trường hợp cây có cây con phải cao hơn cây con trái 2 đơn vị. (Trường hợp ngược lại được giải quyết trong hàm **`left_balance`** mà chúng ta sẽ xem như bài tập). Cho **`root`** là gốc của cây và **`right_tree`** là gốc của cây con phải của nó.

Có ba trường hợp cần phải xem xét, phụ thuộc vào thông số cân bằng của gốc của `right_tree`.

***Trường hợp 1*: Cây con phải của `right_tree` cao hơn cây con trái của nó**

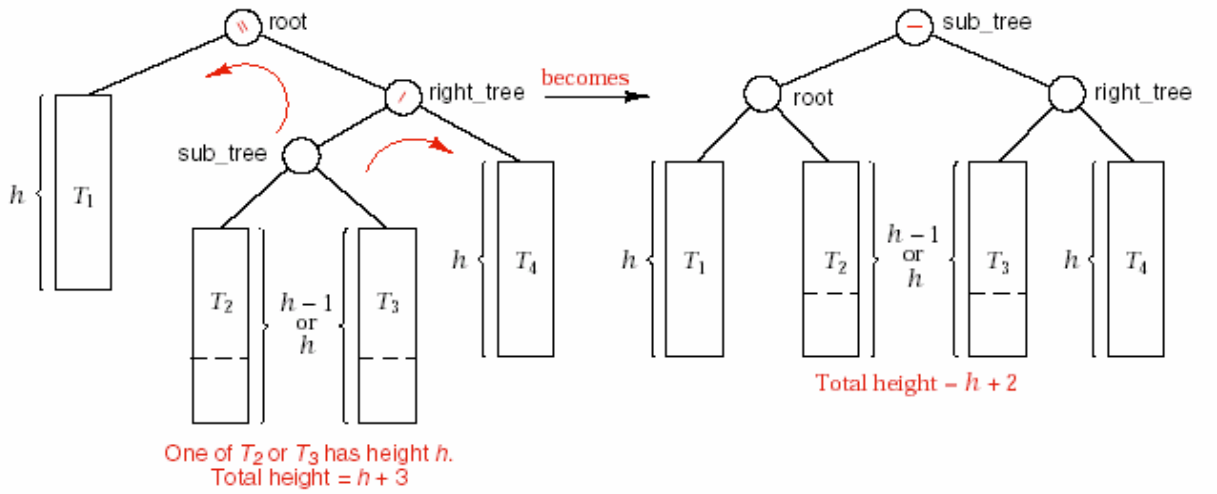


Hình 9.18 – Trường hợp 1: Khôi phục sự cân bằng bởi phép quay trái.

Trường hợp thứ nhất, khi thông số cân bằng trong nút gốc của `right_tree` là `right_higher`, xem hình 9.18, cần thực hiện phép quay trái (*left rotation*). Chúng ta cần quay nút gốc của **`right_tree`** hướng về **`root`**, hạ **`root`** xuống thành cây con trái. Cây con T2, vốn là cây con trái của `right_tree`, trở thành cây con phải của `root`. Do T2 gồm các nút có khóa nằm giữa khóa của `root` và khóa của nút gốc của `right_tree` nên cách thay đổi như vậy vẫn bảo đảm thứ tự các khóa của một cây nhị phân tìm kiếm. Phép quay trái được hiện thực trong hàm **rotate_left** dưới đây. Chúng ta đặc biệt lưu ý rằng, khi thực hiện theo một trật tự thích hợp, các bước gán tạo thành một sự quay vòng của các trị trong ba biến con trỏ. Sau phép quay, chiều cao của toàn bộ cây giảm 1, nhưng do nó vừa tăng như đã nói ở trên (và đây cũng chính là nguyên nhân gây nên sự mất cân bằng cần phải cân bằng lại bằng phép quay này) , nên chiều cao cuối cùng của cây xem như không đổi.

```
template <class Record>
void AVL_tree<Record>::rotate_left(Binary_node<Record> *&sub_root)
/*
pre:  sub_root là địa chỉ của gốc của một cây con trong cây AVL. Cây con này có cây con phải
      không rỗng.
post: sub_root được gán lại địa chỉ của nút vốn là con phải của nó, nút gốc của cây con ban
      đầu sẽ thành nút con trái của cây con mới.
*/
```

```
{
   if (sub_root == NULL || sub_root->right == NULL) // các trường hợp
                                                    // không thể xảy ra
      cout << "WARNING: program error detected in rotate_left" << endl;

else {
      Binary_node<Record> *right_tree = sub_root->right;
      sub_root->right = right_tree->left;
      right_tree->left = sub_root;
      sub_root = right_tree;
   }
}
```



Hình 9.19 – Trường hợp 2: Khôi phục sự cân bằng bởi phép quay kép.

***Trường hợp 2*: Cây con trái của `right_tree` cao hơn cây con phải của nó**

Trường hợp thứ hai, khi thông số trong `right_tree` là `left_higher`, phức tạp hơn. Theo hình vẽ 9.19, nút **`sub_tree`** cần di chuyển lên hai mức để thành nút gốc mới. Cây con phải T3 của **`sub_tree`** sẽ trở thành cây con trái của **`right_tree`**. Cây con trái T2 của **`sub_tree`** trở thành cây con phải của **`root`**. Quá trình này còn được gọi là phép quay kép (*double rotation*), do sự biến đổi cần qua hai bước. Thứ nhất là cây con có gốc là **`right_tree`** được quay sang phải để **`sub_tree`** trở thành gốc của cây này. Sau đó cây có gốc là **`root`** quay sang trái để **`sub_tree`** trở thành gốc.

Trong trường hợp thứ hai này, thông số cân bằng mới của **`root`** và **`right_tree`** phụ thuộc vào thông số cân bằng trước đó của **`sub_tree`**. Thông số cân bằng mới của **`sub_tree`** luôn là `equal_height`. Hình 9.19 minh họa các cây con của **`sub_tree`** có chiều cao bằng nhau, nhưng thực tế **`sub_tree`** có thể là left_higher hoặc right_higher. Các thông số cân bằng kết quả sẽ là:

old **sub_tree**	*new* **root**	*new* **right_tree**	*new* **sub_tree**
-	-	-	-
/	-	\	-
\	/	-	-

***Trường hợp 3*: Hai cây con của `right_tree` cao bằng nhau**

Cuối cùng, chúng ta xét trường hợp thứ ba, khi cả hai cây con của **`right_tree`** cao bằng nhau, nhưng thực tế trường hợp này không thể xảy ra.

Cũng nên nhắc lại rằng trường hợp cần giải quyết ở đây là do chúng ta vừa thêm một nút mới vào cây có gốc **`right_tree`,** làm cho cây này có chiều cao lớn hơn chiều cao cây con trái của **`root`** là 2 đơn vị. Nếu cây `right_tree` có hai cây con cao bằng nhau sau khi nhận thêm một nút mới vào cây con trái hoặc cây con phải của nó, thì **nó đã không thể tăng chiều cao**, do chiều cao của nó vẫn luôn bằng chiều cao của cây con vốn cao hơn cộng thêm 1. Vậy, trước khi thêm nút mới mà cây có gốc là `right_tree` đã cao hơn cây con trái của root 2 đơn vị là vô lý và sai giả thiết rằng cây vốn phải thỏa điều kiện của cây AVL.

9.5.2.3. Hàm right_balance

Chúng ta có hàm `right_balance` dưới đây. Các hàm `rotate_right` và `left_balance` cũng tương tự `rotate_left` và `right_balance`, chúng ta xem như bài tập.

```
template <class Record>
void AVL_tree<Record>::right_balance(Binary_node<Record> *&sub_root)
/*
pre:  sub_root chứa địa chỉ gốc của cây con trong AVL mà tại nút gốc này đang vi phạm điều
      kiện AVL: cây con phải cao hơn cây con trái 2 đơn vị.
post: Việc cân bằng lại giúp cho cây con có gốc sub_root đã thoả điều kiện cây AVL.
uses: các phương thức của AVL_node; các hàm rotate_right và rotate_left.
*/
{
   Binary_node<Record> *&right_tree = sub_root->right;

   switch (right_tree->get_balance()) {

   case right_higher: // Thực hiện 1 phép quay đơn sang trái.
      sub_root->set_balance(equal_height);
      right_tree->set_balance(equal_height);
      rotate_left(sub_root);
      break;

   case equal_height:                          // Trường hợp không thể xảy ra.
      cout << "WARNING:program error detected in right_balance" <<endl;
```

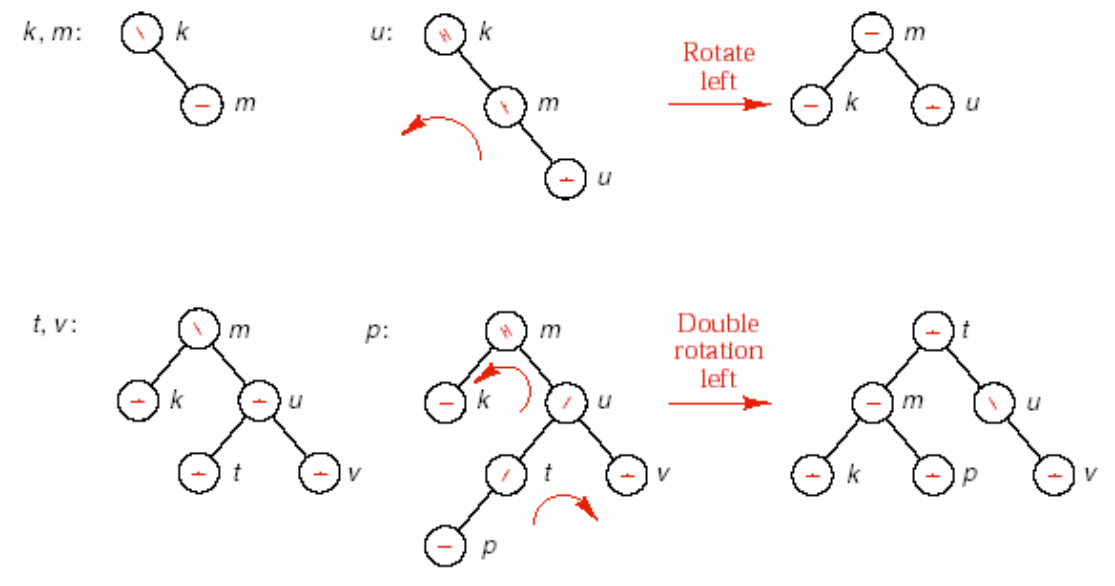
```
   case left_higher:   // Quay kép: quay đơn sang phải, rồi quay đơn sang trái.
      Binary_node<Record> *sub_tree = right_tree->left;
      switch (sub_tree->get_balance()) {

      case equal_height:
         sub_root->set_balance(equal_height);
         right_tree->set_balance(equal_height);
         break;

      case left_higher:
         sub_root->set_balance(equal_height);
         right_tree->set_balance(right_higher);
         break;

      case right_higher:
         sub_root->set_balance(left_higher);
         right_tree->set_balance(equal_height);
         break;
      }
      sub_tree->set_balance(equal_height);
      rotate_right(right_tree);
      rotate_left(sub_root);
      break;
   }
}
```

Hình 9.20 minh họa một ví dụ việc thêm vào cần có quay đơn và quay kép.

9.5.2.4. Hành vi của giải thuật

Số lần mà hàm `avl_insert` gọi đệ quy chính nó để thêm một nút mới có thể lớn bằng chiều cao của cây. Thoạt nhìn, dường như là mỗi lần gọi đệ quy đều dẫn đến một lần quay đơn hoặc quay kép cho cây con tương ứng, nhưng thực ra, nhiều nhất là chỉ có một phép quay (đơn hoặc kép) là được thực hiện. Để nhìn thấy điều này, chúng ta biết rằng các phép quay được thực hiện chỉ trong các hàm `right_balance` và `left_balance`, và các hàm này được gọi chỉ khi chiều cao của cây con có tăng lên. Tuy nhiên, khi các hàm này thực hiện xong, các phép quay đã làm cho chiều cao cây con trở về giá trị ban đầu, như vậy, đối với các lần gọi đệ quy trước đó chưa kết thúc, chiều cao cây con tương ứng sẽ không thay đổi, và sẽ không có một phép quay hay một sự thay đổi các thông số cân bằng nào nữa.

Phần lớn việc thêm vào cây AVL không dẫn đến phép quay. Ngay cả khi phép quay là cần thiết, nó thường xảy ra gần với nút lá vừa được thêm vào. Mặc dù giải thuật thêm vào một cây AVL là phức tạp, nhưng chúng ta có lý do để tin rằng thời gian chạy của nó không khác bao nhiêu so với việc thêm vào một cây nhị phân tìm kiếm bình thường có cùng chiều cao. Chúng ta còn có thể hy vọng rằng chiều cao của cây AVL nhỏ hơn rất nhiều chiều cao của cây nhị phân tìm kiếm bình thường, và như vậy cả hai việc thêm vào và loại bỏ nút sẽ hiệu quả hơn nhiều so với cây nhị phân tìm kiếm bình thường.



Hình 9.20 – Thêm nút vào cây AVL: các trường hợp cần có phép quay.

9.5.3. Loại một nút

Việc loại một nút x ra khỏi một cây AVL cũng theo ý tưởng tương tự, cũng bao gồm phép quay đơn và phép quay kép. Chúng ta sẽ phác thảo các bước của giải thuật dưới đây, hiện thực cụ thể của hàm xem như bài tập.

1. Chúng ta chỉ cần xem xét trường hợp nút x cần loại có nhiều nhất là một cây con, vì đối với trường hợp x có hai cây con, chúng ta sẽ tìm nút y là nút kế trước nó (hoặc nút kế sau nó) trong thứ tự duyệt *inorder* để loại thay cho nó. Từ nút con trái của x nếu di chuyển sang phải cho đến khi không thể di chuyển được nữa, chúng ta gặp một nút y không có con phải. Lúc đó chúng ta sẽ chép dữ liệu trong y vào x, không thay đổi thông số cân bằng cũng như con trái và con phải của x. Cuối cùng nút y sẽ được loại đi thay cho nút x theo các bước dưới đây.

2. Gọi nút cần loại là x. Loại nút x ra khỏi cây. Do x có nhiều nhất một con, việc loại x đơn giản chỉ là nối tham chiếu từ nút cha của x đến nút con của nó (hoặc `NULL` nếu x không có con nào). Chiều cao cây con có gốc là x giảm đi 1, và điều này **có thể ảnh hưởng đến các nút nằm trên đường đi từ x ngược về gốc của cây**. Vì vậy trong quá trình duyệt cây để xuống đến nút x, chúng ta cần lưu vết bằng cách sử dụng ngăn xếp để có thể xử lý lần ngược về. Để đơn giản, chúng ta có thể dùng **hàm đệ quy với tham biến `shorter`** kiểu `bool` cho biết chiều cao của cây con có bị giảm đi hay không. Các việc cần làm tại mỗi nút phụ thuộc vào trị của tham biến `shorter` mà con nó trả về sau khi việc gọi đệ quy xuống cây con kết thúc, vào thông số cân bằng của nó, và đôi khi phụ thuộc vào cả thông số cân bằng của nút con của nó.

3. Biến `bool shorter` được khởi gán trị true. Các bước sau đây sẽ được thực hiện tại mỗi nút nằm trên đường đi từ nút cha của x cho đến nút gốc của cây. Trong khi mà tham biến `shorter` trả lên còn là `true`, việc giải quyết cứ tiếp tục cho đến khi có một nút nhận được trị trả về của tham biến này là `false`. Một khi có một cây con nào đó báo rằng chiều cao của nó không bị giảm đi thì các nút trên của nó sẽ không bị ảnh hưởng gì cả. Ngược lại, gọi nút p là nút vừa gọi đệ quy xuống nút con xong và nhận thông báo `shorter` là `true`, cần xét các trường hợp sau:

   Trường hợp 1: Nút p hiện tại có thông số cân bằng là **`equal_height`**. Thông số này sẽ thay đổi tùy thuộc vào cây con trái hay cây con phải của nó đã bị ngắn đi. Biến `shorter` đổi thành **`false`**.

   Trường hợp 2: Nút p hiện tại có thông số cân bằng **không là `equal_height`**. **Cây con vốn cao hơn vừa bị ngắn đi**. Thông số cân bằng của p chỉ cần đổi về `equal_height`. Trường hợp này cây gốc p cũng bị giảm chiều cao nên tham biến `shorter` vẫn là `true` để trả lên trên cho cha của p giải quyết tiếp.

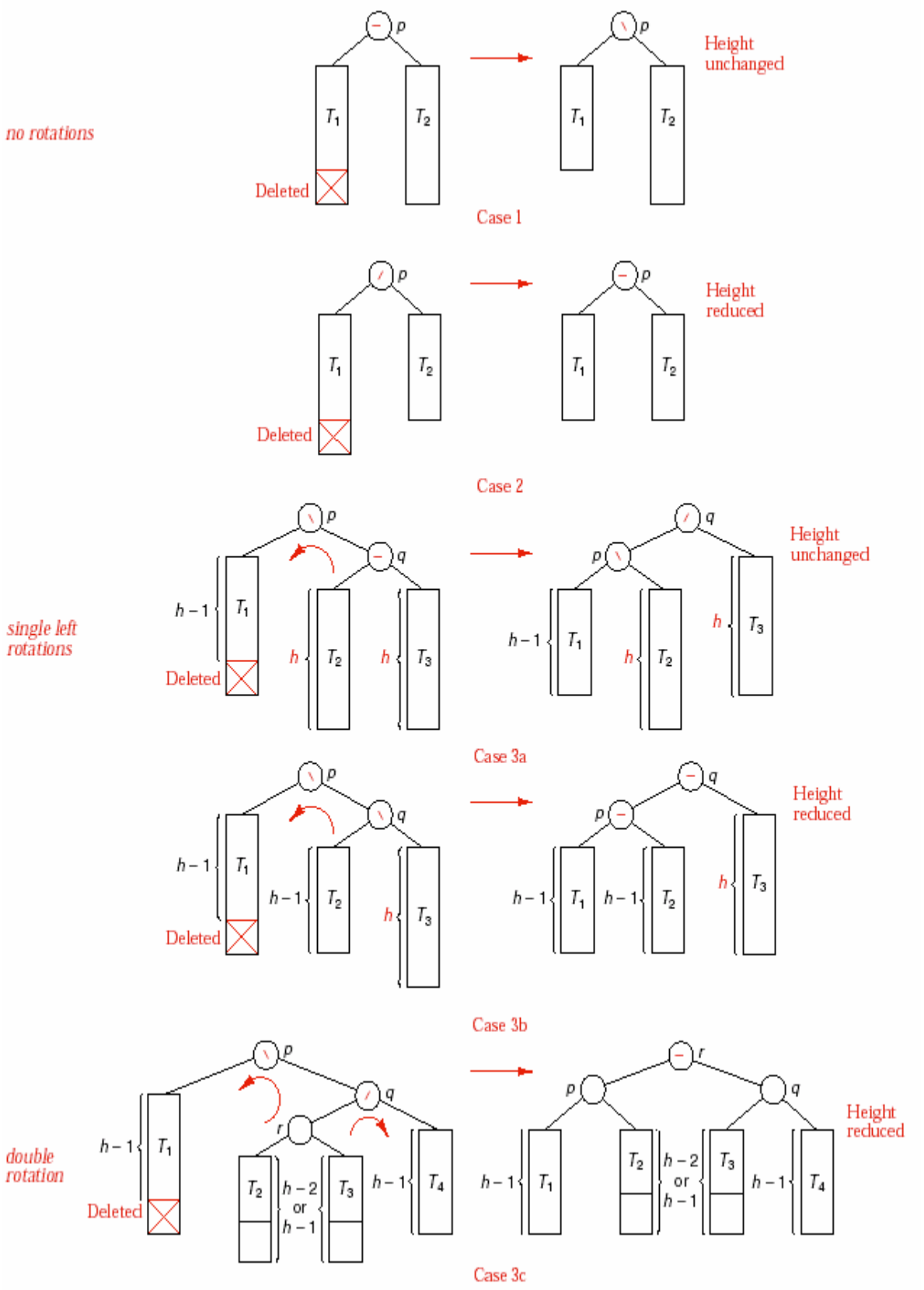
   Trường hợp 3: Nút p hiện tại có thông số cân bằng **không là `equal_height`. Cây con vốn thấp hơn vừa bị ngắn đi**. Như vậy tại nút p **vi phạm điều kiện của cây AVL**, và chúng ta thực hiện phép quay để khôi phục lại sự cân bằng như sau. Gọi q là gốc của cây con cao hơn của p, có 3 trường hợp tương ứng với thông số cân bằng trong q:

      Trường hợp 3a: **Thông số cân bằng của q là `equal_height`**. Thực hiện một phép **quay đơn** để thay đổi các thông số cân bằng của p và q, trạng thái cân bằng sẽ được khôi phục, `shorter` đổi thành **`false.`**

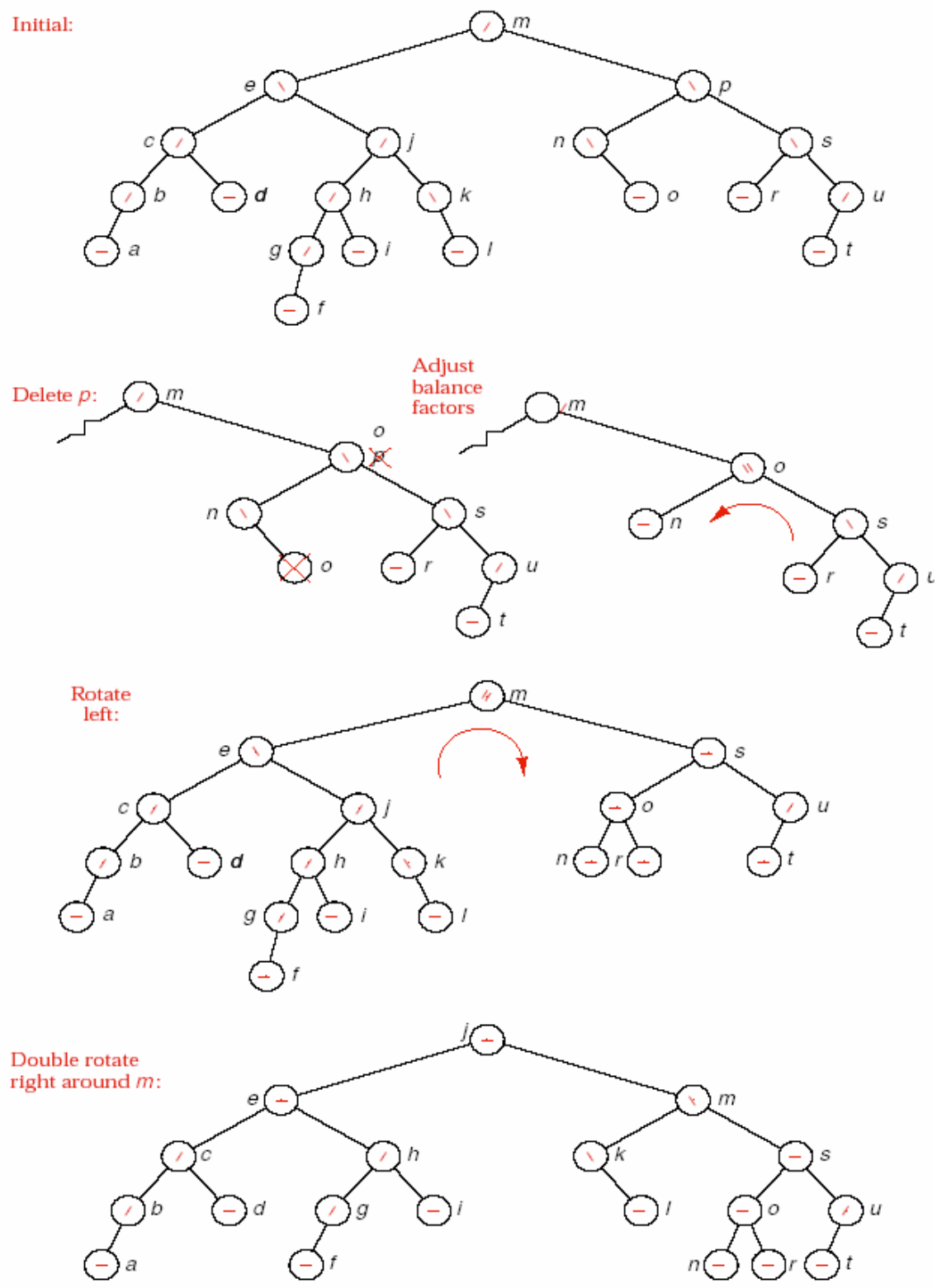
      Trường hợp 3b: **Thông số cân bằng của q giống với thông số cân bằng của p**. Thực hiện một phép **quay đơn** để chuyển các thông số *này thành `equal_height`, `shorter` vẫn là* ***`true.`***

      Trường hợp 3c: **Thông số cân bằng của q ngược với thông số cân bằng của p.** Thực hiện một phép **quay kép** (trước hết quay quanh q, sau đó quay quanh p), thông số cân bằng của gốc mới sẽ là `equal_height`, các thông số cân bằng của p và q được biến đổi thích hợp, `shorter` vẫn là **`true.`**

Trong các trường hợp 3a, b, c, chiều của các phép quay phụ thuộc vào cây con trái hay cây con phải bị ngắn đi. Hình 9.21 minh họa một vài trường hợp, và một ví dụ loại một nút được minh họa trong hình 9.22.



Hình 9.21 – Các trường hợp loại một nút ra khỏi cây AVL.



Hình 9.22 – Ví dụ loại một nút ra khỏi cây AVL.

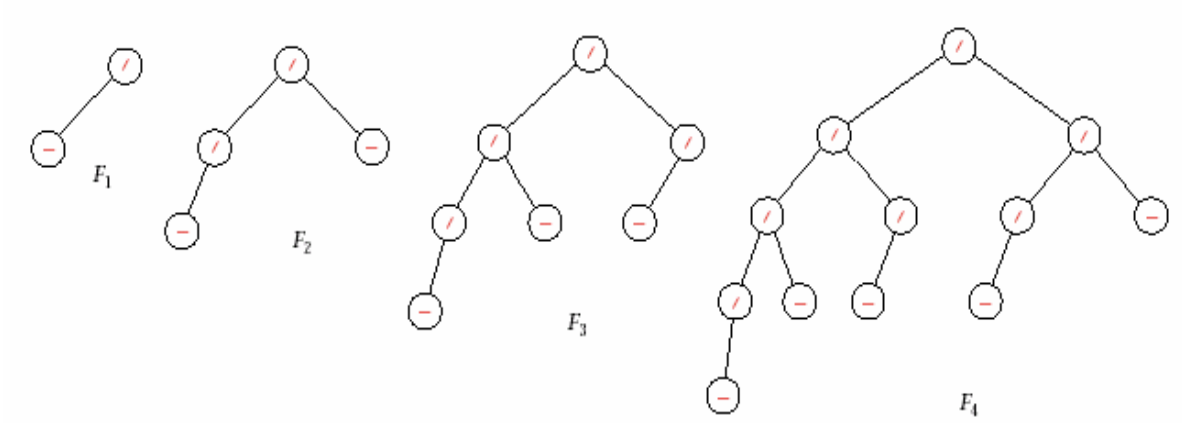
9.5.4. Chiều cao của cây AVL

Việc tìm chiều cao của một cây AVL trung bình là một việc rất khó, do đó việc xác định số bước trung bình cần thực hiện bởi các giải thuật trong phần này cũng không dễ. Cách dễ nhất cho chúng ta là xác định những gì sẽ xảy ra trong trường hợp xấu nhất, và các kết quả này sẽ cho chúng ta thấy rằng các hành vi của cây AVL trong trường hợp xấu nhất cũng không tệ hơn các hành vi của các cây ngẫu nhiên. Bằng chứng thực nghiệm cho thấy các hành vi trung bình của các cây AVL tốt hơn rất nhiều so với các cây ngẫu nhiên, hầu như nó còn tốt ngang với những gì sẽ có được từ một cây cân bằng hoàn toàn.

Để xác định chiều cao tối đa có thể có được của một cây AVL có n nút, chúng ta sẽ xem thử với một chiều cao h cho trước, cây AVL sẽ có số nút tối thiểu là bao nhiêu. Gọi F_h là một cây như vậy, thì cây con trái và cây con phải của nút gốc của F_h

sẽ là F_l và F_r. Một trong hai cây con này phải có chiều cao là h-1, giả sử cây F_l, và cây con còn lại có chiều cao h-1 hoặc h-2. Do F_h có số nút tối thiểu của một cây AVL có chiều cao h, F_l cũng phải có số nút tối thiểu của cây AVL cao h-1 (có nghĩa là F_l có dạng F_{h-1}), và F_r phải có chiều cao h-2 với số nút tối thiểu (F_r có dạng F_{h-2}).

Các cây được tạo bởi quy tắc trên, nghĩa là các cây thỏa điều kiện cây AVL nhưng càng thưa càng tốt, được gọi là các cây *Fibonacci*. Một số ít cây đầu tiên có thể được thấy trong hình 9.23.



Hình 9.23 – Các cây Fibonacci

Với ký hiệu |T| biểu diễn số nút trong cây T, chúng ta có công thức sau:

$$| F_h | = | F_{h-1} | + | F_{h-2} | + 1.$$

với $| F_0 | = 1$ và $| F_1 | = 2$. Bằng cách thêm 1 vào cả hai vế, chúng ta thấy con số $| F_h | + 1$ thỏa định nghĩa của các số *Fibonacci* bậc 2. Bằng cách tính các số *Fibonacci* chúng ta có

$$| F_h | + 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2}$$

Lấy *logarit* hai vế và chỉ giữ lại các số lớn chúng ta có

$$h \approx 1.44 \lg | F_h |.$$

Kết quả cho thấy một cây AVL n nút thưa nhất sẽ có chiều cao xấp xỉ 1.44 lg n. Một cây nhị phân cân bằng hoàn toàn có n nút có chiều cao bằng lg n, còn cây suy thoái sẽ có chiều cao là n. Thời gian chạy các giải thuật trên cây AVL được bảo đảm không vượt quá 44 phần trăm thời gian cần thiết đối với cây tối ưu. Trong thực tế, các cây AVL còn tốt hơn rất nhiều. Điều này có thể được chứng minh như sau, ngay cả đối với các cây *Fibonacci* - các cây AVL trong trường hợp xấu nhất – thời gian tìm kiếm trung bình chỉ có 4 phần trăm lớn hơn cây tối ưu. Phần lớn các cây AVL sẽ không thưa thớt như các cây *Fibonacci*, và do đó thời gian tìm kiếm trung bình trên các cây AVL trung bình rất gần với cây tối ưu. Thực nghiệm cho thấy số lần so sánh trung bình khoảng lg n + 0.25 khi n lớn.