Chương 10 – CÂY NHIỀU NHÁNH

Chương này tiếp tục nghiên cứu về các cấu trúc dữ liệu cây, tập trung vào các cây mà số nhánh tại mỗi nút nhiều hơn hai. Chúng ta bắt đầu từ việc trình bày các mối nối trong cây nhị phân. Kế tiếp chúng ta tìm hiểu về một lớp của cây gọi là *trie* được xem như từ điển chứa các từ. Sau đó chúng ta tìm hiểu đến cây *B-tree* có ý nghĩa rất lớn trong việc truy xuất thông tin trong các tập tin. Mỗi phần trong số này độc lập với các phần còn lại. Cuối cùng, chúng ta áp dụng ý tưởng của *B-tree* để có được một lớp khác của cây nhị phân tìm kiếm gọi là cây đỏ-đen (*red-black tree*).

10.1. Vườn cây, cây, và cây nhị phân

Như chúng ta đã thấy, cây nhị phân là một dạng cấu trúc dữ liệu đơn giản và hiệu quả. Tuy nhiên, với một số ứng dụng cần sử dụng cấu trúc dữ liệu cây mà trong đó số con của mỗi nút chưa biết trước, cây nhị phân với hạn chế mỗi nút chỉ có tối đa hai con không đáp ứng được. Phần này làm sáng tỏ một điều ngạc nhiên thú vị và hữu ích: cây nhị phân cung cấp một khả năng biểu diễn những cây khác bao quát hơn.

10.1.1. Các tên gọi cho cây

Trước khi mở rộng về các loại cây, chúng ta xét đến các định nghĩa. Trong toán học, khái niệm cây có một ý nghĩa rộng: đó là một tập bất kỳ các điểm (gọi là **đỉnh**), và tập bất kỳ các cặp nối hai đỉnh khác nhau (gọi là **cạnh** hoặc nhánh) sao cho luôn có một dãy liên tục các cạnh (**đường đi**) từ một đỉnh bất kỳ đến một đỉnh bất kỳ khác, và không có chu trình, nghĩa là không có đường đi nào bắt đầu từ một đỉnh nào đó lại quay về chính nó.

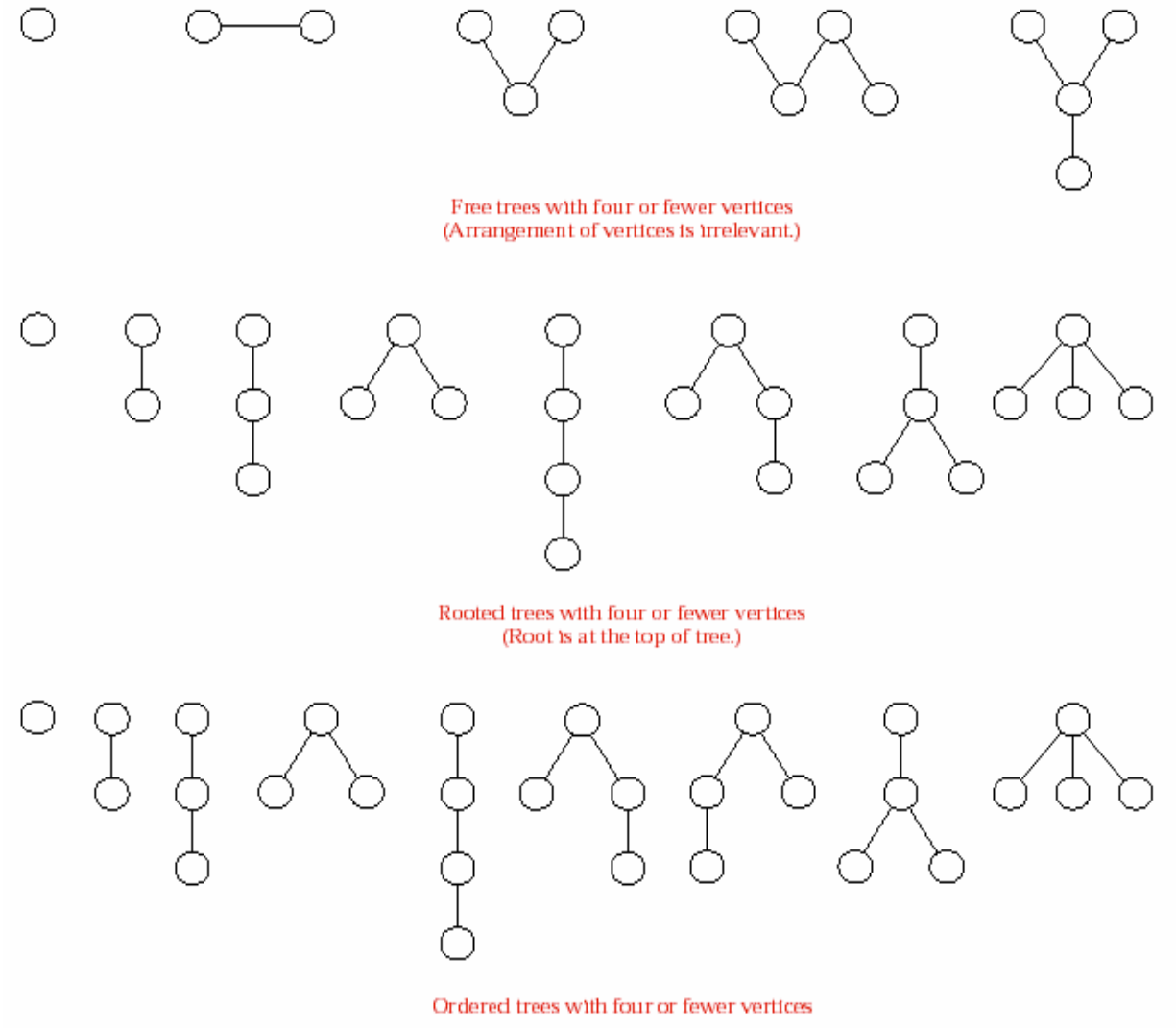
Đối với các ứng dụng trong máy tính, chúng ta thường không cần nghiên cứu cây một cách tổng quát như vậy, và khi cần làm việc với những cây này, để nhấn mạnh, chúng ta thường gọi chúng là các **cây tự do** (*free tree*). Các cây của chúng ta phần lớn luôn có một đỉnh đặc biệt, gọi là gốc của cây, và các cây dạng này chúng ta sẽ gọi là các **cây có gốc** (*rooted tree*).

Một cây có gốc có thể được vẽ theo cách thông thường của chúng ta là gốc nằm trên, các nút và nhánh khác quay xuống dưới, với các nút lá nằm dưới cùng. Mặc dù vậy, các cây có gốc vẫn chưa phải là tất cả các dạng cây mà chúng ta thường dùng. Trong một cây có gốc, thường không phân biệt trái hoặc phải, hoặc khi một nút có nhiều nút con, không thể nói rằng nút nào là nút con thứ nhất, thứ hai, v.v...Nếu không vì một lý do nào khác, sự thi hành tuần tự các lệnh thường buộc chặt một thứ tự lên các nút con của một nút. Chúng ta định nghĩa một cây có thứ

tự (*ordered tree*) là một cây có gốc trong đó các con của một nút được gán cho một thứ tự.

Lưu ý rằng các cây có thứ tự mà trong đó mỗi nút có không quá hai con vẫn chưa phải cùng một lớp với cây nhị phân. Nếu một nút trong cây nhị phân chỉ có một con, nó có thể nằm bên trái hoặc bên phải, lúc đó ta có hai cây nhị phân khác nhau, nhưng chúng cùng là một cây có thứ tự.

Như một nhận xét cuối cùng liên quan đến các định nghĩa, chúng ta hãy lưu ý rằng cây *2-tree* mà chúng ta đã nghiên cứu khi phân tích các giải thuật ở những chương trước là một cây có gốc (nhưng không nhất thiết phải là cây có thứ tự) với đặc tính là mỗi nút trong cây có 0 hoặc 2 nút con.



Hình 10.1 - Các dạng khác nhau của cây.

Hình 10.1 cho thấy rất nhiều dạng cây khác nhau với số nút nhỏ. Mỗi lớp cây kể từ cây đầu tiên có được bằng cách kết hợp các cây từ các lớp có trước theo nhiều cách khác nhau. Các cây nhị phân có thể có được từ các cây có thứ tự tương ứng, bằng cách phân biệt các nhánh trái và phải.

10.1.2. Cây có thứ tự

10.1.2.1. Hiện thực trong máy tính

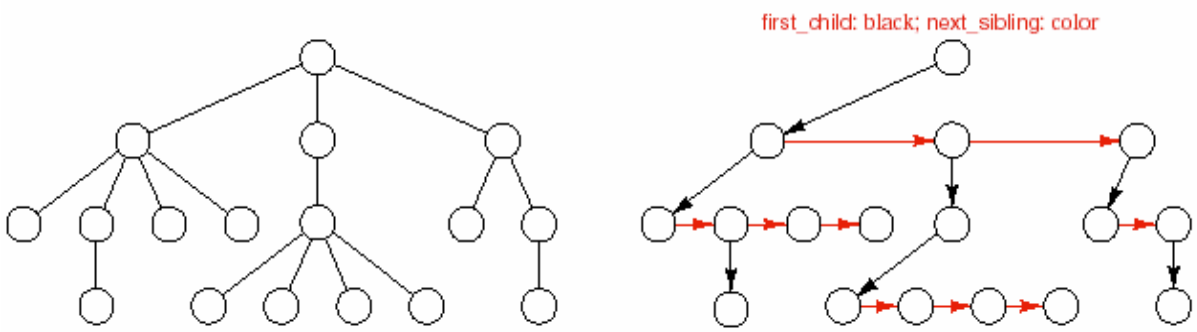
Nếu chúng ta muốn sử dụng một cây có thứ tự như một cấu trúc dữ liệu, một cách hiển nhiên để hiện thực trong bộ nhớ máy tính là mở rộng cách hiện thực chuẩn của một cây nhị phân, với số con trỏ thành viên trong mỗi nút tương ứng số cây con có thể có, thay vì chỉ có hai như đối với cây nhị phân. Chẳng hạn, trong một cây có một vài nút có đến mười cây con, chúng ta cần phải giữ đến mười con trỏ thành viên trong một nút. Nhưng như vậy sẽ dẫn đến việc cây phải chứa một số rất lớn các con trỏ chứa trị `NULL`. Chúng ta có thể tính được chính xác con số này. Nếu cây có n nút, mỗi nút có k con trỏ thành viên, thì sẽ có tất cả là n x k con trỏ. Mỗi nút có chính xác là một con trỏ tham chiếu đến nó, ngoại trừ nút gốc. Như vậy có n-1 con trỏ khác `NULL`. Tỉ lệ các con trỏ `NULL` sẽ là:

$$\frac{(n \times k) - (n - 1)}{n \times k} > 1 - \frac{1}{k}$$

Nếu một nút có thể có mười cây con, thì có hơn 90% con trỏ là `NULL`. Rõ ràng là phương pháp biểu diễn cây có thứ tự này hao tốn rất nhiều vùng nhớ. Lý do là vì, trong mỗi nút, chúng ta đã giữ một danh sách liên tục các con trỏ đến tất cả các con của nó, và các danh sách liên tục này chứa quá nhiều vùng nhớ chưa được sử dụng. Chúng ta cần tìm cách thay thế các danh sách liên tục này bởi các danh sách liên kết.

10.1.2.2. Hiện thực liên kết

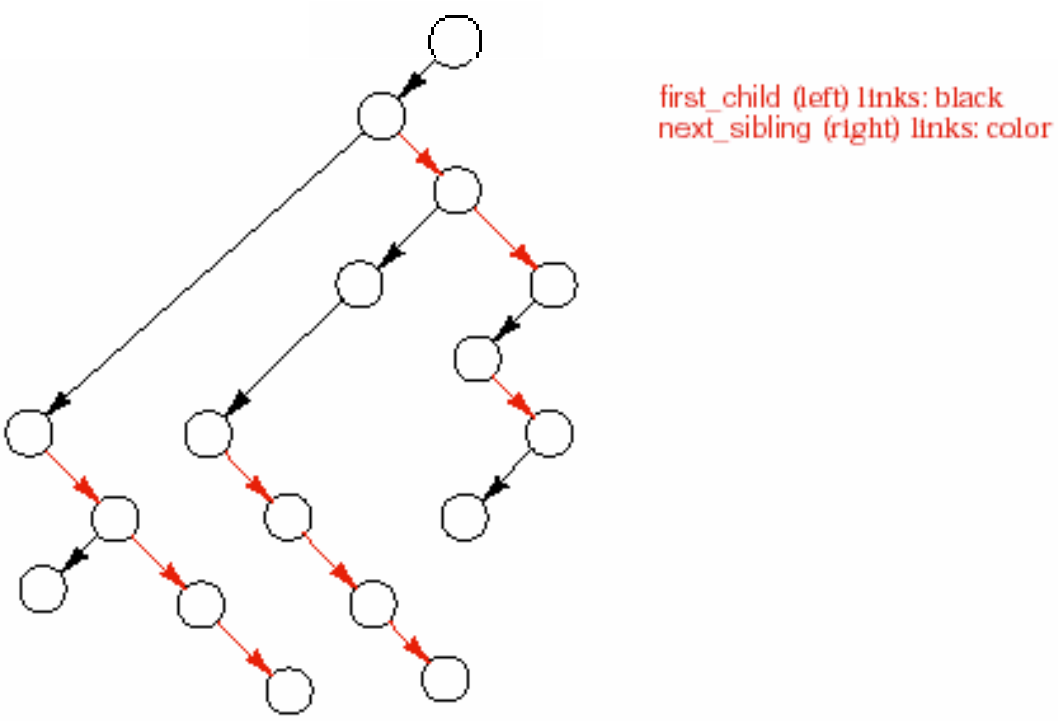
Để nắm các con của một nút trong một danh sách liên kết, chúng ta cần hai loại tham chiếu. Thứ nhất là tham chiếu từ nút cha đến nút con đầu tiên bên trái của nó, chúng ta sẽ gọi là **`first_child`**. Thứ hai, mỗi nút, ngoại trừ nút gốc, sẽ xuất hiện như một phần tử trong danh sách liên kết này, do đó nó cần thêm một tham chiếu đến nút kế trong danh sách, nghĩa là tham chiếu đến nút con kế tiếp cùng cha. Tham chiếu thứ hai này được gọi là **`next_sibling`**. Hiện thực này được minh họa trong hình 10.2.



Hình 10.2 – Hiện thực liên kết của cây có thứ tự

10.1.2.3. Sự tương ứng tự nhiên

Đối với mỗi nút của cây có thứ tự chúng ta đã định nghĩa hai tham chiếu **`first_child`** và **`next_sibling`**. Bằng cách sử dụng hai tham chiếu này chúng ta có được cấu trúc của một cây nhị phân, nghĩa là, hiện thực liên kết của một cây có thứ tự là một cây nhị phân liên kết. Nếu muốn, chúng ta có thể có được một hình ảnh dễ nhìn hơn cho cây nhị phân bằng cách sử dụng hiện thực liên kết của cây có thứ tự và quay theo chiều kim đồng hồ một góc nhỏ, sao cho các tham chiếu hướng xuống (**`first_child`**) hướng sang trái, và các tham chiếu nằm ngang (**`next_sibling`**) hướng sang phải. Đối với hình 10.2, chúng ta có được cây nhị phân ở hình 10.3.



Hình 10.3 – Hình đã được quay của hiện thực liên kết

10.1.2.4. Sự tương ứng ngược lại

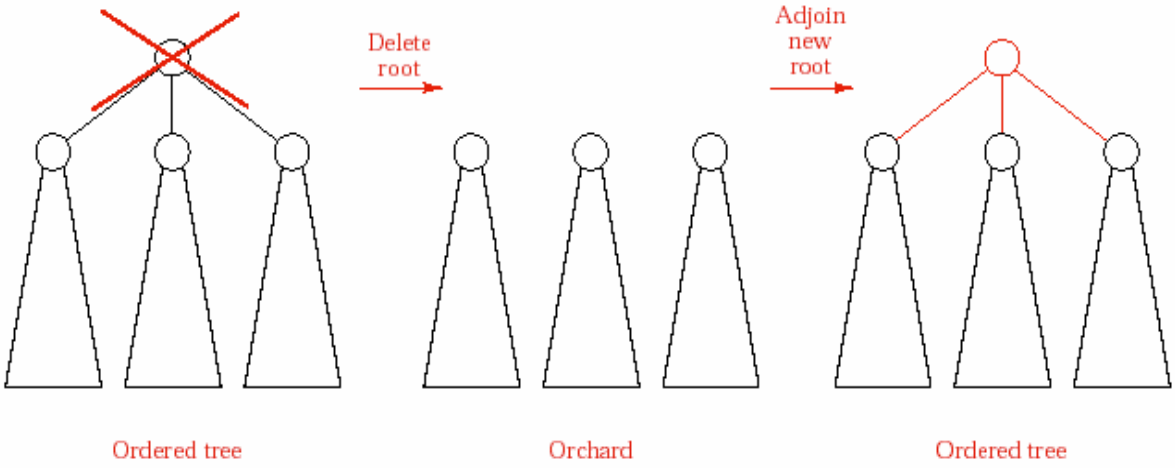
Giả sử như chúng ta làm ngược lại các bước của quá trình trên, bắt đầu từ một cây nhị phân và cố gắng khôi phục lại một cây có thứ tự. Điều quan sát đầu tiên chúng ta cần nhận thấy là không phải mọi cây nhị phân đều có thể có được từ một cây có thứ tự bởi quá trình trên: do tham chiếu **`next_sibling`** của nút gốc của cây có thứ tự luôn bằng `NULL` nên gốc của cây nhị phân tương ứng luôn có cây con bên phải rỗng. Để tìm hiểu sự tương ứng ngược lại này một cách cẩn thận, chúng ta cần phải xem xét một lớp cấu trúc dữ liệu khác qua một số định nghĩa mới dưới đây.

10.1.3. Rừng và vườn

Trong quá trình tìm hiểu về cây nhị phân chúng ta đã có kinh nghiệm về cách sử dụng đệ quy, đối với các lớp khác của cây chúng ta cũng sẽ tiếp tục làm như vậy. Sử dụng đệ quy có nghĩa là thu hẹp vấn đề thành vấn đề nhỏ hơn. Do đó chúng ta nên xem thử điều gì sẽ xảy ra nếu chúng ta lấy **một cây có gốc** hoặc **một cây có thứ tự** và cắt bỏ đi nút gốc. Những phần còn lại, nếu không rỗng, sẽ là **một tập các cây có gốc** hoặc **một tập có thứ tự các cây có thứ tự** tương ứng.

Thuật ngữ chuẩn để gọi một tập trừu tượng các cây đó là rừng (*forest*), nhưng khi chúng ta dùng thuật ngữ này, nói chung chúng ta thường hình dung đó là các **cây có gốc**. Cụm từ "rừng có thứ tự" (*ordered forest*) đôi khi còn được sử dụng để gọi **tập có thứ tự các cây có thứ tự**, do đó chúng ta sẽ đề cử một thuật ngữ có tính đặc tả tương tự cho **lớp các cây có thứ tự**, đó là thuật ngữ **vườn** (*orchard*).

Lưu ý rằng chúng ta không chỉ có được một **rừng** hoặc một **vườn** nhờ vào cách loại bỏ đi nút gốc của một **cây có gốc** hoặc một **cây có thứ tự**, chúng ta còn có thể tạo nên một **cây có gốc** hoặc một **cây có thứ tự** bằng cách bắt đầu từ một **rừng** hoặc một **vườn**, thêm một nút mới tại đỉnh, và nối các nhánh từ nút mới này đến gốc của tất cả các cây trong rừng hoặc vườn đó. Cách này được minh họa trong hình 10.4.



Hình 10.4 – Loại bỏ và thêm nút gốc.

Chúng ta sẽ sử dụng quá trình này để đưa ra một định nghĩa đệ quy mới cho các cây có thứ tự và các vườn. Trước hết, chúng ta hãy xem thử nên bắt đầu như thế nào. Chúng ta nhớ rằng một cây nhị phân có thể rỗng. Một rừng hay một vườn cũng có thể rỗng. Tuy nhiên một cây có gốc hay một cây có thứ tự không thể là cây rỗng, vì nó phải chứa ít nhất là một nút gốc. Nếu chúng ta muốn bắt đầu xây dựng cây và rừng, chúng ta có thể lưu ý rằng một cây với chỉ một nút có thể có được bằng cách thêm một gốc mới vào một rừng đang rỗng. Một khi chúng ta đã có cây này rồi thì chúng ta có thể tạo được một rừng gồm bao nhiêu cây một nút cũng được. Sau đó chúng ta có thể thêm gốc mới để tạo các cây có gốc chiều

cao là 1. Bằng cách này chúng ta có thể tiếp tục tạo nên các cây có gốc phù hợp với định nghĩa đệ quy sau:

> *Định nghĩa*: Một **cây có gốc** (*rooted tree*) bao gồm một nút đơn v, gọi là gốc (*root*) của cây, và một **rừng** F (*forest*) gồm các cây gọi là các cây con của nút gốc.

Một rừng *F* là một tập (có thể rỗng) các cây có gốc.

Một quá trình tạo tương tự cho các cây có thứ tự và vườn.

> *Định nghĩa*: Một **cây có thứ tự** *T* (*ordered tree*) bao gồm một nút đơn v, gọi là gốc (*root*) của cây,và một **vườn** *O* (*orchard*) gồm các cây được gọi là các cây con của gốc v.

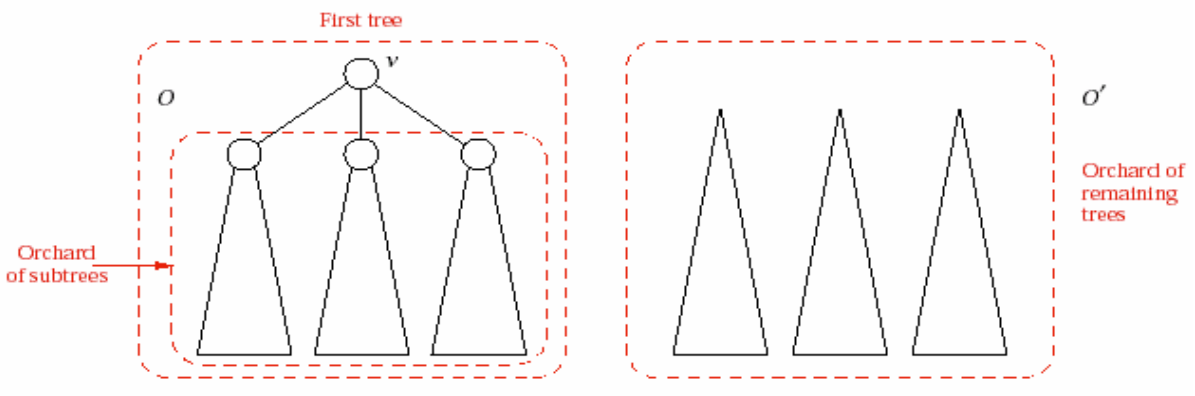
Chúng ta có thể biểu diễn cây có thứ tự bằng một cặp có thứ tự

$$T = \{v, O\}.$$

Một vườn *O* hoặc là một tập rỗng, hoặc gồm một cây có thứ tự *T*, gọi là cây thứ nhất (*first tree*) của vườn, và một vườn khác *O'* (chứa các cây còn lại của vườn). Chúng ta có thể biểu diễn vườn bằng một cặp có thứ tự

$$O = (T, O').$$

Lưu ý rằng thứ tự của các cây ẩn chứa trong định nghĩa của vườn. Một vườn không rỗng chứa cây thứ nhất và các cây còn lại tạo nên một vườn khác, vườn này lại có một cây thứ nhất và là cây thứ hai của vườn ban đầu. Tiếp tục đối với các vườn còn lại chúng ta có cây thứ ba, thứ tư, v.v...cho đến khi vườn cuối cùng là một vườn rỗng. Xem hình 10.5.



Hình 10.5 – Cấu trúc đệ quy của các cây có thứ tự và vườn.

10.1.4. Sự tương ứng hình thức

Bây giờ chúng ta có thể có một kết quả mang tính nguyên tắc cho phần này.

Định lý: *Cho S là một tập hữu hạn bất kỳ gồm các nút. Có một ánh xạ một-một f từ tập các vườn có tập nút là S đến tập các cây nhị phân có tập nút là S.*

Chứng minh định lý:

Chúng ta sẽ dùng những ký hiệu trong các định nghĩa để chứng minh định lý trên. Trước hết chúng ta cần một ký hiệu tương tự cho cây nhị phân. Một cây nhị phân B hoặc là một tập rỗng ∅ hoặc gồm một nút gốc ν và hai cây nhị phân B_1 và B_2. Ký hiệu cho một cây nhị phân không rỗng là một bộ ba

$$B = [\nu, B_1, B_2].$$

Chúng ta sẽ chứng minh định lý bằng phương pháp quy nạp toán học trên số nút trong *S*. Trường hợp thứ nhất được xét là một vườn rỗng ∅, tương ứng với một cây nhị phân rỗng.

$$f(\varnothing) = \varnothing.$$

Nếu vườn *O* không rỗng, nó được ký hiệu bằng một bộ hai

$$O = (T, O_2)$$

với T là một cây có thứ tự và O_2 là một vườn khác. Cây thứ tự T được ký hiệu bởi một cặp

$$T = \{\nu, O_1\}$$

với ν là một nút và O_1 là một vườn khác. Thay biểu thức T vào biểu thức O ta có

$$O = (\{\nu, O_1\}, O_2).$$

Theo giả thiết quy nạp, f là một ánh xạ một-một từ các vườn có ít nút hơn *S* đến các cây nhị phân, với O_1 và O_2 nhỏ hơn *O*, nên các cây nhị phân $f(O_1)$ và $f(O_2)$ được xác định bởi giả thiết quy nạp. Nếu chúng ta định nghĩa ánh xạ f từ một vườn đến một cây nhị phân bởi

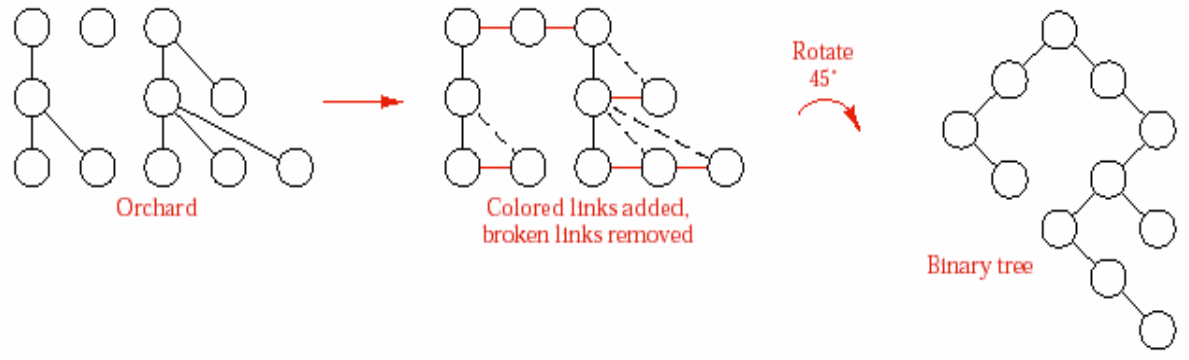
$$f(\{\nu, O_1\}, O_2) = [\nu, f(O_1), f(O_2)].$$

thì f là một sự tương ứng một-một giữa các vườn và các cây nhị phân có cùng số nút. Với bất kỳ cách thay thế nào cho các ký tự ν, O_1, và O_2 ở vế trái đều có chính xác một cách để thay thế cho chúng ở vế phải, và ngược lại.

10.1.5. Phép quay

Chúng ta có thể sử dụng dạng ký hiệu của sự tương ứng để hình dung phép biến đổi từ vườn sang cây nhị phân. Trong cây nhị phân [v, $f(O_1)$, $f(O_2)$] tham chiếu trái từ v đến nút gốc của cây nhị phân $f(O_1)$, đó là nút con thứ nhất của v trong cây có thứ tự {v, O_1}. Tham chiếu phải từ v đến nút vốn là gốc của cây có thứ tự kế tiếp về bên phải trong vườn. Có nghĩa là, "tham chiếu trái" trong cây nhị phân tương ứng với "con thứ nhất" trong cây có thứ tự, và "tham chiếu phải" tương ứng "em kế". Các quy tắc biến đổi trong hình như sau:

1. Vẽ vườn sao cho con thứ nhất của mỗi nút nằm ngay dưới nó, thay vì canh khoảng cách cho tất cả các con nằm đều bên dưới nút này.
2. Vẽ một tham chiếu thẳng đứng từ mỗi nút đến nút con thứ nhất của nó, và vẽ một tham chiếu nằm ngang từ mỗi nút đến em kế của nó.
3. Loại bỏ tất cả các tham chiếu khác còn lại.
4. Quay sơ đồ 45 độ theo chiều kim đồng hồ, sao cho các tham chiếu thẳng đứng trở thành các tham chiếu trái và các tham chiếu nằm ngang trở thành các tham chiếu phải.
5. Quá trình này được minh họa trong hình 10.6



Hình 10.6 – Chuyển đổi từ vườn sang cây nhị phân.

10.1.6. Tổng kết

Chúng ta đã xem xét ba cách biểu diễn sự tương ứng giữa các vườn và các cây nhị phân:

- Các tham chiếu **first_child** và **next_sibling**.
- Phép quay các sơ đồ.
- Sự tương đương ký hiệu một cách hình thức.

Nhiều người cho rằng cách thứ hai, quay các sơ đồ, là cách dễ nhớ và dễ hình dung nhất. Cách thứ nhất, tạo các tham chiếu, thường được dùng để viết các chương trình thực sự. Cuối cùng, cách thứ ba, sự tương đương ký hiệu một cách

hình thức, thường rất có ích trong việc chứng minh rất nhiều đặc tính của cây nhị phân và vườn.

10.2. Cây từ điển tìm kiếm: Trie

Trong các chương trước chúng ta đã thấy sự khác nhau trong việc tìm kiếm trong một danh sách và việc tra cứu trong một bảng. Chúng ta có thể áp dụng ý tưởng trong việc tra cứu bảng vào việc truy xuất thông tin trong một cây bằng cách sử dụng một khóa hoặc một phần của khóa. Thay vì tìm kiếm bằng cách so sánh các khóa, chúng ta có thể xem khóa như là một chuỗi các ký tự (chữ cái hoặc ký số), và sử dụng các ký tự này để xác định đường đi tại mỗi bước. Nếu các khóa của chúng ta chứa các chữ cái, chúng ta sẽ tạo một cây có 26 nhánh tương ứng 26 chữ cái là ký tự đầu tiên của các khóa. Mỗi cây con bên dưới lại có 26 nhánh tương ứng với ký tự thứ hai, và cứ thế tiếp tục ở các mức cao hơn. Tuy nhiên chúng ta cũng có thể tiến hành phân thành nhiều nhánh ở một số mức ban đầu, sau đó nếu cây trở nên quá lớn, chúng ta có thể dùng một vài cách thức khác nào đó để sắp thứ tự cho những mức còn lại.

10.2.1. Tries

Có một phương pháp là cắt tỉa bớt các nhánh không cần thiết trong cây. Đó là các nhánh không dẫn đến một khóa nào. Lấy ví dụ, trong tiếng Anh, không có các từ bắt đầu bởi 'bb', 'bc', 'bf', 'bg', ..., nhưng có các từ bắt đầu bởi 'ba', 'bd', 'be'. Do đó, mọi nhánh và nút cho các từ không tồn tại có thể được loại khỏi cây. Cây kết quả này được gọi là **`Trie`**. Từ này nguyên thủy được lấy từ *re**trie**val*, nhưng thường được đọc là "*try*".

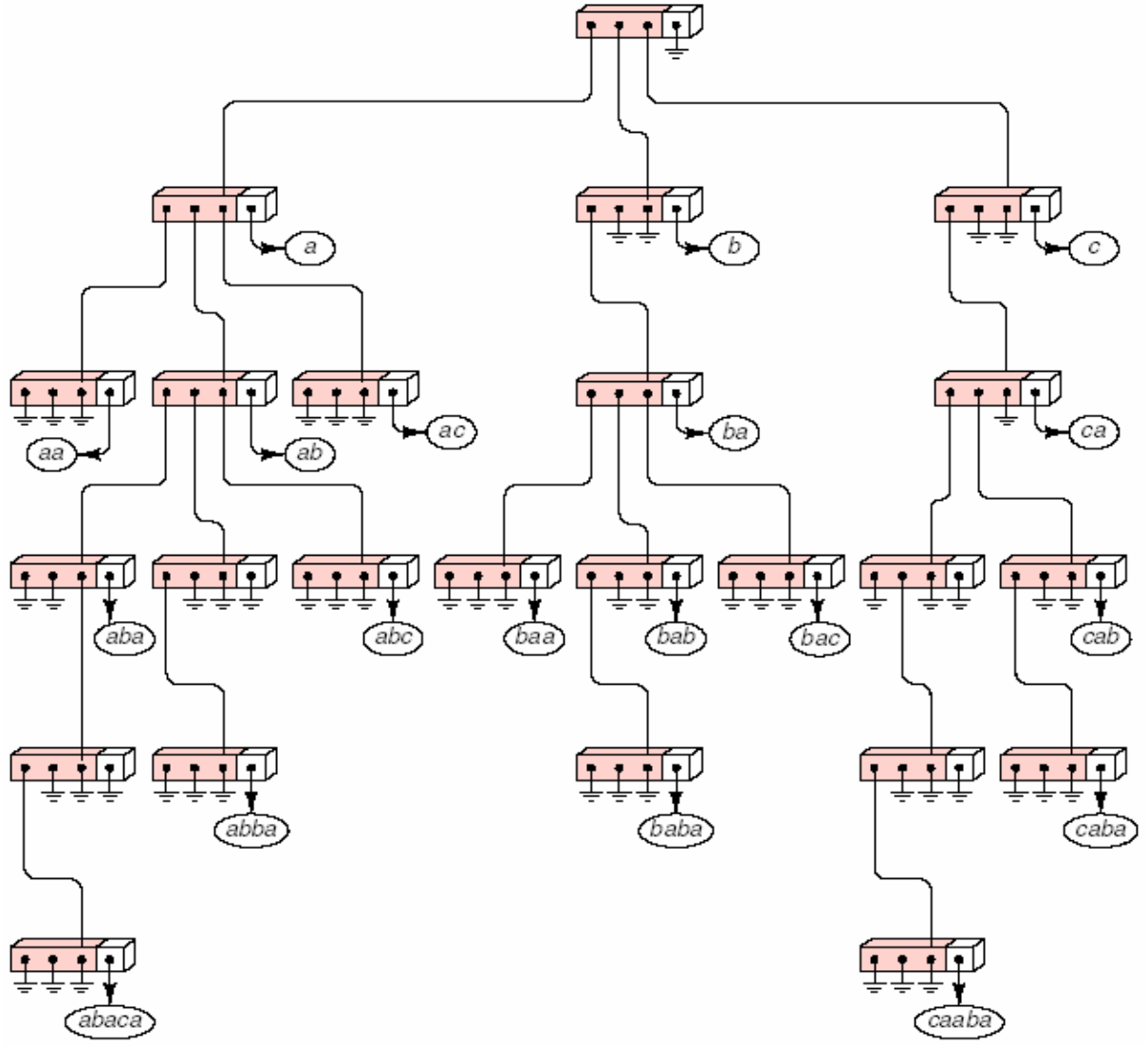
Định nghĩa: Một cây `Trie` bậc m có thể được định nghĩa một cách hình thức là một cây rỗng hoặc gồm một chuỗi nối tiếp có thứ tự của m cây `Trie` bậc m.

10.2.2. Tìm kiếm một khóa

Giả sử các từ có 3 ký tự có nghĩa gồm các từ được lưu trong cây `Trie` ở hình 10.7. Việc tìm kiếm một khóa được bắt đầu từ nút gốc. Ký tự đầu tiên của khóa được dùng để xác định nhánh nào cần đi xuống. Nhánh cần đi rỗng có nghĩa là khóa cần tìm chưa có trong cây. Ngược lại, trên nhánh được chọn này, ký tự thứ hai lại được dùng để xác định nhánh nào trong mức kế tiếp cần đi xuống, và cứ thế tiếp tục. Khi chúng ta xét đến cuối từ, là chúng ta đã đến được nút có con trỏ tham chiếu đến thông tin cần tìm. Đối với nút tương ứng một từ không có nghĩa sẽ có con trỏ tham chiếu đến thông tin là `NULL`. Chẳng hạn, từ **a** là phần đầu của từ **aba**, từ này lại là phần đầu của từ **abaca**, nhưng chuỗi ký tự **abac** không phải là một từ có nghĩa, do đó nút biểu diễn **abac** có con trỏ tham chiếu thông tin là `NULL`.

10.2.3. Giải thuật C++

Chúng ta sẽ chuyển quá trình tìm kiếm vừa được mô tả trên thành một phương thức tìm kiếm các bản ghi có khóa là các chuỗi ký tự. Chúng ta sẽ sử dụng phương thức **`char key_letter(int position)`** trả về ký tự tại vị trí **`position`** trong khóa hoặc ký tự rỗng nếu khóa có chiều dài ngắn hơn **`position`**,



Hình 10.7 – Trie chứa các từ được cấu tạo từ a, b, c.

và hàm phụ trợ **`int alphabetic_order(char symbol)`** trả về thứ tự của **`symbol`** trong bảng chữ cái. Hàm này trả về 0 cho ký tự rỗng, 27 cho các ký tự không phải chữ cái. Trong hiện thực liên kết, cây *`Trie`* chứa một con trỏ đến nút gốc của nó.

```
class Trie {
public:  // Các phương thức cập nhật, tìm kiếm, truy xuất.

private:
   Trie_node *root;
};
```

Mỗi nút của Trie cần chứa một con trỏ chỉ đến một bản ghi và một mảng các con trỏ đến các nhánh. Số nhánh là 28 tương ứng kết quả trả về của **alphabetic_order**.

```
const int num_chars = 28;
struct Trie_node {
//    Các thuộc tính
   Record *data;
   Trie_node *branch[num_chars];
//    constructors
   Trie_node();
};
```

Constructor cho **Trie_node** đơn giản chỉ gán tất cả các con trỏ là NULL.

10.2.4. Tìm kiếm trong cây Trie

Phương thức sau tìm một bản ghi chứa khóa cho trước trong cây Trie.

```
Error_code Trie::trie_search(const Key &target, Record &x) const
/*
post: Nếu tìm thấy khóa target, bản ghi x chứa khóa sẽ được trả về, phương thức trả về
      success. Ngược lại phương thức trả về not_present.
uses: Các phương thức của lớp Key.
*/
{
   int position = 0;
   char next_char;
   Trie_node *location = root;
   while (location!=NULL&&(next_char=target.key_letter(position))!=' ')
   {
      location = location->branch[alphabetic_order(next_char)];
                 // Đi xuống dần các nhánh tương ứng với các ký tự trong target.
      position++;// Để xét ký tự kế tiếp của target.
   }

   if (location != NULL && location->data != NULL) {
      x = *(location->data);
      return success;
   }
   else
      return not_present;
}
```

Điều kiện kết thúc vòng lặp là con trỏ location bằng NULL (khóa cần tìm không có trong cây), hoặc ký tự kế là rỗng (đã xét hết chiều dài khóa cần tìm). Kết thúc vòng lặp, con trỏ **location** nếu khác NULL chính là con trỏ tham chiếu bản ghi chứa khóa cần tìm.

10.2.5. Thêm phần tử vào Trie

Thêm một phần tử vào cây Trie hoàn toàn tương tự như tìm kiếm: lần theo các nhánh để đi xuống cho đến khi gặp vị trí thích hợp, tạo bản ghi chứa dữ liệu

và cho con trỏ **data** chỉ đến. Nếu trên đường đi chúng ta gặp một nhánh NULL, chúng ta phải tạo thêm các nút mới để đưa vào cây sao cho có thể tạo được một đường đi đến nút tương ứng với khóa mới cần thêm vào.

```
Error_code Trie::insert(const Record &new_entry)
/*
post: Nếu khóa của new_entry đã có trong Trie, phương thức trả về duplicate_error.
      Ngược lại new_entry được thêm vào Trie, phương thức trả về success.
uses: các phương thức của các lớp Record và Trie_node.
*/
{
   Error_code result = success;
   if (root == NULL) root = new Trie_node; // Tạo một cây Trie rỗng.
   int position = 0;                // Vị trí ký tự đang xét trong new_entry.
   char next_char;
   Trie_node *location = root;      //  Đi dần xuống các nhánh trong Trie.
   while (location != NULL &&
          (next_char = new_entry.key_letter(position)) != ' ') {
      int next_position = alphabetic_order(next_char);
      if (location->branch[next_position] == NULL)
         location->branch[next_position] = new Trie_node;
      location = location->branch[next_position];
      position++;
   }
            // Không còn nhánh để đi tiếp hoặc đã xét hết các ký tự của new_entry.
   if (location->data != NULL) result = duplicate_error;
   else location->data = new Record(new_entry);
   return result;
}
```

10.2.6. Loại phần tử trong Trie

Cách thực hiện của việc thêm và tìm kiếm phần tử cũng được áp dụng cho việc loại một phần tử trong cây Trie. Chúng ta lần theo đường đi tương ứng với khóa cần loại, khi gặp nút này, chúng ta gán NULL cho con trỏ **data**. Tuy nhiên, nếu nút này có tất cả các thuộc tính đều là các con trỏ NULL (các cây con và con trỏ **data**), chúng ta cần xóa luôn chính nó. Và điều này cần phải được thực hiện cho tất cả các nút trên của nó trên đường đi từ nó ngược về nút gốc cho đến khi gặp một nút có ít nhất một thuộc tính thành viên khác NULL. Để làm được điều này, chúng ta có thể tạo một ngăn xếp chứa các con trỏ đến các nút trên đường đi từ nút gốc đến nút cần tìm để loại. Hoặc chúng ta có thể sử dụng đệ quy trong giải thuật loại phần tử nhằm tránh việc sử dụng ngăn xếp một cách tường minh. Cả hai cách này đều được xem như bài tập.

10.2.7. Truy xuất Trie

Số bước cần thực hiện để tìm kiếm trong cây Trie (hoặc thêm nút mới vào Trie) tỉ lệ với số ký tự tạo nên một khóa, không phụ thuộc vào *logarit* của số khóa như các cách tìm kiếm dựa trên các cây khác. Nếu số ký tự nhỏ so với *logarit* cơ số 2 của số khóa, cây Trie tỏ ra có ưu thế hơn cây nhị phân tìm kiếm

nhiều. Lấy ví dụ, các khóa gồm mọi khả năng của một chuỗi 5 ký tự, thì cây `Trie` có thể chứa đến n = 26^5 = 11,881,376 khóa với mỗi lần tìm kiếm tối đa là 5 lần lặp để đi xuống 5 mức, trong khi đó cây nhị phân tìm kiếm tốt nhất có thể thực hiện đến lg n ≈ 23.5 lần so sánh các khóa.

Tuy nhiên, trong nhiều ứng dụng có số ký tự trong một khóa lớn, và tập các khóa thực sự xuất hiện lại ít so với mọi khả năng có thể có của các khóa. Trong trường hợp này, số lần lặp cần có để tìm một khóa trong cây *Trie* có thể vượt xa số lần so sánh các khóa cần có trong cây nhị phân tìm kiếm.

Cuối cùng, lời giải tốt nhất có thể là sự kết hợp của nhiều phương pháp. Cây *Trie* có thể được sử dụng cho một ít ký tự đầu của các khóa, và sau đó một phương pháp khác có thể được sử dụng cho phần còn lại của khóa.

10.3. Tìm kiếm ngoài: B-tree

Từ trước đến nay, chúng ta đã giả sử rằng mọi cấu trúc dữ liệu đều được giữ trong bộ nhớ tốc độ cao; nghĩa là chúng ta đã chỉ xem xét việc truy xuất thông tin trong (*internal information retrieval*). Với một số ứng dụng, giả thiết này có thể chấp nhận được, nhưng với nhiều ứng dụng quan trọng khác thì không. Chúng ta hãy xem xét vấn đề truy xuất thông tin ngoài (*external information retrieval*), trong đó các bản ghi cần tìm kiếm và truy xuất được lưu trong các tập tin.

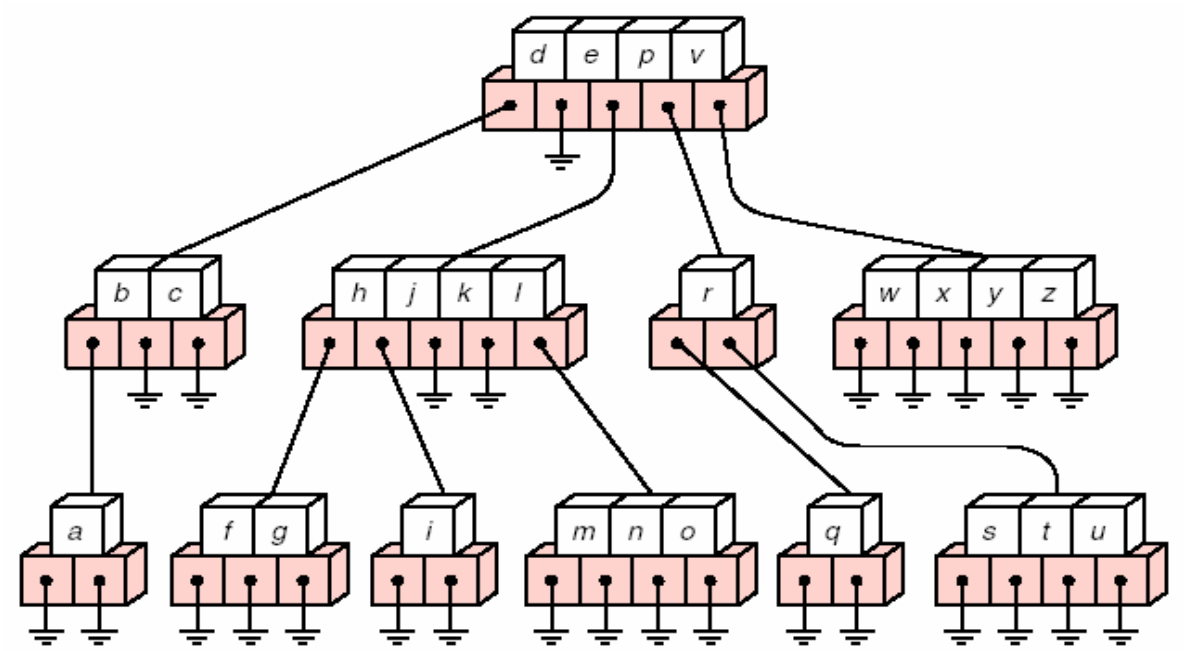
10.3.1. Thời gian truy xuất

Thời gian cần có để thâm nhập và truy xuất một từ trong bộ nhớ tốc độ cao nhiều nhất là một vài microgiây. Thời gian cần để định vị một bản ghi trong đĩa cứng được đo bằng miligiây, đối với đĩa mềm có thể vượt quá một giây. Như vậy thời gian cho một lần truy xuất ngoài lớn gấp hàng ngàn lần so với một lần truy xuất trong. Khi một bản ghi nằm trong đĩa, thực tế mỗi lần không phải chỉ đọc một từ, mà đọc một trang lớn (*page*) hay còn gọi là một khối (*block*) thông tin. Kích thước chuẩn của khối thường từ 256 đến 1024 ký tự hoặc từ.

Mục đích của chúng ta trong việc tìm kiếm ngoài là phải làm tối thiểu số lần truy xuất đĩa, do mỗi lần truy xuất chiếm thời gian đáng kể so với các tính toán bên trong bộ nhớ. Mỗi lần truy xuất đĩa, chúng ta có được một khối mà có thể chứa nhiều bản ghi. Bằng cách sử dụng các bản ghi này, chúng ta có thể chọn lựa giữa nhiều khả năng để quyết định khối nào sẽ được truy xuất kế tiếp. Nhờ đó mà toàn bộ dữ liệu không cần phải lưu đồng thời trong bộ nhớ. Khái niệm cây nhiều nhánh mà chúng ta sẽ xem xét dưới đây đặc biệt thích hợp đối với việc tìm kiếm ngoài.

10.3.2. Cây tìm kiếm nhiều nhánh

Cây nhị phân tìm kiếm được tổng quát hóa một cách trực tiếp đến cây tìm kiếm nhiều nhánh, trong đó, với một số nguyên m nào đó được gọi là bậc (*order*) của cây, mỗi nút có nhiều nhất m nút con. Nếu k ($k \le m$) là số con của một nút thì nút này chứa chính xác là k-1 khóa, và các khóa này phân hoạch tất cả các khóa của các cây con thành k tập con. Hình 10.8 cho thấy một cây tìm kiếm có 5 nhánh nằm xen kẽ các phần tử từ thứ 1 và đến thứ 4 trong mỗi nút, trong đó một vài nhánh có thể rỗng.



Hình 10.8 – Một cây tìm kiếm 5 nhánh (không phải cây B-tree)

10.3.3. Cây nhiều nhánh cân bằng

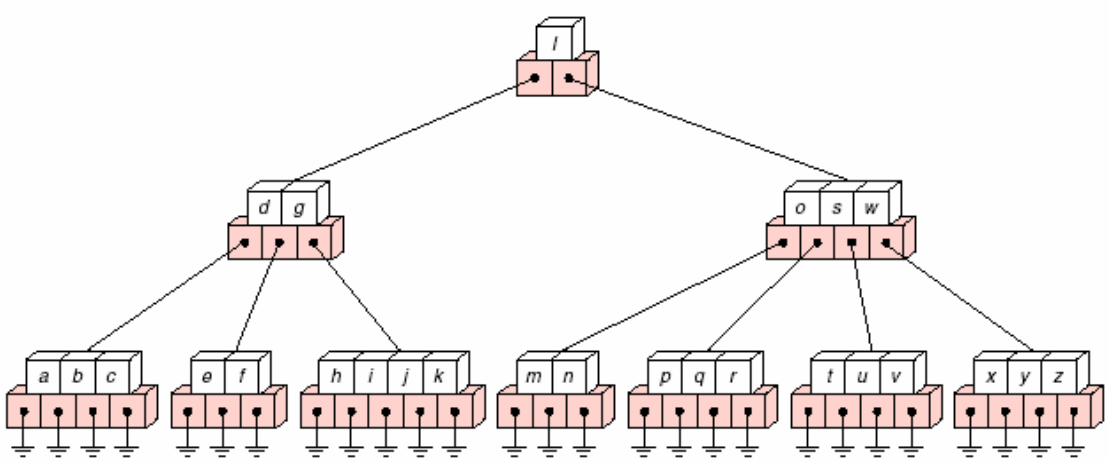
Giả sử mỗi lần đọc tập tin, chúng ta đọc lên được một khối chứa các khóa trong cùng một nút. Nhờ sự phân hoạch các khóa trong các cây con dựa trên các khóa này, chúng ta biết được nhánh nào chúng ta cần tiếp tục công việc tìm kiếm khóa cần tìm. Bằng cách này số lần đọc đĩa tối đa chính là chiều cao của cây. Và chi phí bộ nhớ cũng chỉ dành tối đa là cho các nút trên đường đi từ nút gốc đến nút có khóa cần tìm, chứ không phải toàn bộ dữ liệu lưu trong cây.

Mục đích của chúng ta sử dụng cây tìm kiếm nhiều nhánh để làm giảm việc truy xuất tập tin, do đó chúng ta mong muốn chiều cao của cây càng nhỏ càng tốt. Chúng ta có thể thực hiện điều này bằng cách cho rằng, thứ nhất, không có các cây con rỗng xuất hiện bên trên các nút lá (như vậy sự phân hoạch các khóa thành các tập con sẽ hiệu quả nhất); thứ hai, rằng mọi nút lá đều thuộc cùng một mức (để cho việc tìm kiếm được bảo đảm là sẽ kết thúc với cùng số lần truy xuất

tập tin); và, thứ ba, rằng mọi nút, ngoại trừ các nút lá có ít nhất một số nút con tối thiểu nào đó. Chúng ta đưa ra yêu cầu rằng, mọi nút, ngoại trừ các nút lá, có ít nhất là một nửa số con so với số con tối đa có thể có. Các điều kiện trên dẫn đến định nghĩa sau:

Định nghĩa: Một cây **B-tree bậc m** là một cây m nhánh, trong đó,
1. Mọi nút lá có cùng mức.
2. Mọi nút trung gian (không phải nút lá và nút gốc), có nhiều nhất m nút con khác rỗng, ít nhất là $\lceil m/2 \rceil$ nút con khác rỗng.
3. Số khóa trong mỗi nút trong nhỏ hơn số nút con khác rỗng 1 đơn vị, và các khóa này phân hoạch các khóa trong các cây con theo cách của cây tìm kiếm.
4. Nút gốc có nhiều nhất m nút con, và nếu nó không đồng thời là nút lá (trường hợp cây chỉ có 1 nút), thì nó có thể có ít nhất là 2 nút con.

Cây trong hình 10.8 không phải là cây B-tree, do một vài nút có các nút con rỗng, một vài nút có quá ít con, và các nút lá không cùng một mức. Hình 10.9 minh họa một cây B-tree có bậc là 5 với các khóa là các ký tự chữ cái. Trường hợp này mỗi nút trung gian có ít nhất 3 nút con (phân hoạch bởi 2 khóa).



Hình 10.9 – Cây B-tree bậc 5.

10.3.4. Thêm phần tử vào B-tree

Điều kiện mọi nút lá thuộc cùng mức nhấn mạnh hành vi đặc trưng của B-tree: Ngược với cây nhị phân tìm kiếm, B-tree không cho phép lớn lên tại các nút lá; thay vào đó, nó lớn lên tại gốc. Phương pháp chung để thêm phần tử vào nó như sau. Trước hết, thực hiện việc tìm kiếm để xem khóa cần thêm đã có trong cây hay chưa. Nếu chưa có, việc tìm kiếm sẽ kết thúc tại một nút lá. Khóa mới sẽ được thêm vào nút lá. Nếu nút lá vốn chưa đầy, việc thêm vào hoàn tất.

Khi nút lá cần thêm phần tử mới đã đầy, nút này sẽ được phân làm hai nút cạnh nhau trong cùng một mức, khóa chính giữa sẽ không thuộc nút nào trong hai nút này, nó được gởi ngược lên để thêm vào nút cha. Nhờ vậy, sau này, khi cần tìm kiếm, sự so sánh với khóa giữa này sẽ dẫn đường xuống tiếp cây con tương ứng bên trái hoặc bên phải. Quá trình phân đôi các nút có thể được lan truyền ngược về gốc. Quá trình này sẽ chấm dứt khi có một nút cha nào đó cần được thêm một khóa gởi từ dưới lên mà chưa đầy. Khi một khóa được thêm vào nút gốc đã đầy, nút gốc sẽ được phân làm hai và khóa nằm giữa cũng được gởi ngược lên, và nó sẽ trở thành một gốc mới. Đó chính là lúc duy nhất cây `B-tree` tăng trưởng chiều cao.

Quá trình này có thể được làm sáng tỏ bằng ví dụ thêm vào cây `B-tree` cấp 5 ở hình 10.10. Chúng ta sẽ lần lượt thêm các khóa

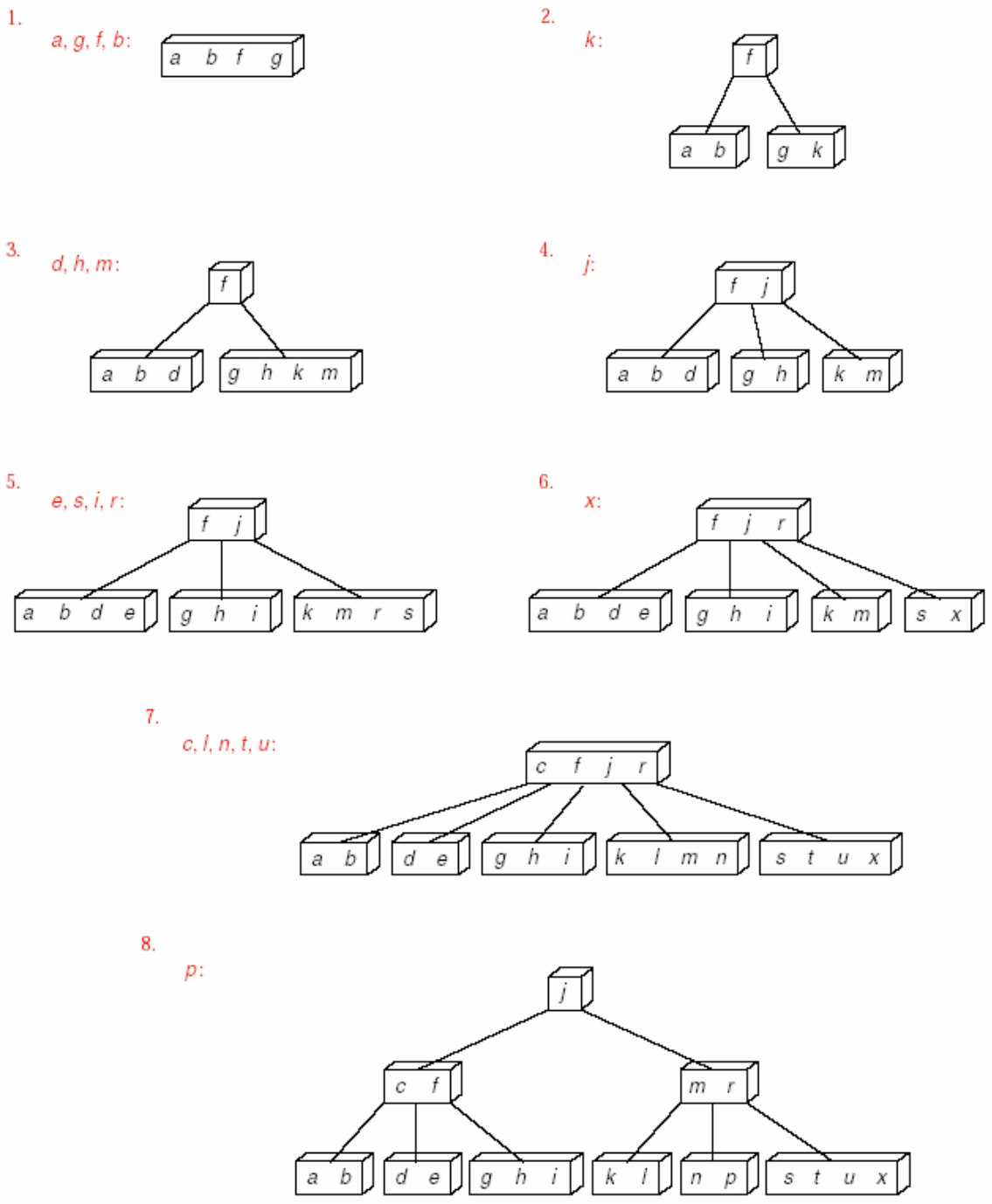
a g f b k d h m j e s i r x c l n t u p

vào một cây rỗng theo thứ tự này.

Bốn khóa đầu tiên sẽ được thêm vào chỉ một nút, như trong phần đầu của hình 10.10. Chúng được sắp thứ tự ngay khi được thêm vào. Tuy nhiên, đối với khóa thứ năm, **k**, nút này không còn chỗ. Nút này được phân làm hai nút mới, khóa nằm giữa, **f**, được chuyển lên trên và tạo nên nút mới, đó cũng là gốc mới. Do các nút sau khi phân chia chỉ chứa một nửa số khóa có thể có, ba khóa tiếp theo có thể được thêm vào mà không gặp khó khăn gì. Tuy nhiên, việc thêm vào đơn giản này cũng đòi hỏi việc tổ chức lại các khóa trong một nút. Để thêm **j**, một lần nữa lại cần phân chia một nút, và lần này khóa chuyển lên trên chính là **j**.

Một số lần thêm các khóa tiếp theo được thực hiện tương tự. Lần thêm cuối cùng, **p**, đặc biệt hơn. Việc thêm **p** vào trước tiên làm phân chia một nút vốn chứa **k, l, m, n**, và gởi khóa nằm giữa **m** lên trên cho nút cha chứa **c, f, j, r,** tuy nhiên, nút này đã đầy. Như vậy, nút này lại phân chia làm hai nút mới, và cuối cùng nút gốc mới chứa **j** được tạo ra.

Có hai điểm cần chú ý khi quan sát sự lớn lên có trật tự của `B-tree`. Thứ nhất, khi một nút được phân đôi, nó tạo ra hai nút mới, mỗi nút chỉ có một nửa số phần tử tối đa có thể có. Nhờ đó, những lần thêm tiếp theo có thể không cần phải phân chia nút lần nữa. Như vậy một lần phân chia nút là chuẩn bị cho một vài lần thêm đơn giản. Thứ hai, khóa được chuyển lên trên luôn là khóa nằm giữa chứ không phải chính khóa cần thêm vào. Do đó, nhiều lần thêm lập lại sẽ có chiều hướng cải thiện sự cân bằng cho cây, không phụ thuộc vào thứ tự các khóa được thêm vào.



Hình 10.10 – Sự lớn lên của cây B-tree.

10.3.5. Giải thuật C++: tìm kiếm và thêm vào

Để phát triển thành giải thuật C++ tìm kiếm và thêm vào một cây `B-tree`, chúng ta hãy bắt đầu với các khai báo cho cây. Để đơn giản chúng ta sẽ xây dựng cây `B-tree` trong bộ nhớ tốc độ cao, sử dụng các con trỏ chứa địa chỉ các nút trong cây. Trong phần lớn các ứng dụng, các con trỏ này có thể được thay thế bởi

địa chỉ của các khối hoặc trang trong đĩa, hoặc số thứ tự các bản ghi trong tập tin.

10.3.5.1. Các khai báo

Chúng ta sẽ cho người sử dụng tự do chọn lựa kiểu của bản ghi mà họ muốn lưu vào cây B-tree. Lớp **B-tree** của chúng ta, và lớp **node** tương ứng, sẽ có thông số template là lớp **Record**. Thông số template thứ hai sẽ là một số nguyên biểu diễn bậc của B-tree. Để có được một đối tượng *B*-tree, người sử dụng chỉ việc khai báo một cách đơn giản, chẳng hạn:**B-tree<int, 5> sample_tree;** sẽ khai báo **sample_tree** là một cây B-tree bậc 5 chứa các bản ghi là các số nguyên.

```
template <class Record, int order>
class B_tree {
public:  //  Các phương thức.

private: //  Thuộc tính:
   B_node<Record, order> *root;
         //  Các hàm phụ trợ.
};
```

Bên trong mỗi nút của B-tree chúng ta cần một danh sách các phần tử và một danh sách các con trỏ đến các nút con. Do cách danh sách này ngắn, để đơn giản, chúng ta dùng các mảng liên tục và một thuộc tính **count** để biểu diễn chúng.

```
template <class Record, int order>
struct B_node {
//  Các thuộc tính:
   int count;
   Record data[order - 1];
   B_node<Record, order> *branch[order];
//  constructor:
   B_node();
};
```

Thuộc tính **count** chứa số bản ghi hiện tại trong từng nút. Nếu **count** khác 0 thì nút có count+1 nút con khác rỗng. Nhánh **branch[0]** chỉ đến cây con chứa các bản ghi có các khóa nhỏ hơn khóa trong **data[0]**; với mỗi trị của **position** nằm giữa 1 và count-1, kể cả hai cận này, **branch[position]** chỉ đến cây con có các khóa nằm giữa hai khóa của **data[position-1]** và **data[position]**; và **branch[count]** chỉ đến cây con có các khóa lớn hơn khóa trong **data[count-1]**.

Constructor của **B_node** tạo một nút rỗng bằng cách gán **count** bằng 0.

10.3.5.2. Tìm kiếm

Như ví dụ đơn giản đầu tiên, chúng ta viết phương thức tìm kiếm trong một cây B-tree cho một bản ghi có khóa trùng với khóa của **target**. Trong phương thức tìm kiếm của chúng ta, như thường lệ, chúng ta sẽ giả thiết rằng các bản ghi này có thể được so sánh bởi các toán tử so sánh chuẩn. Cũng như việc tìm kiếm trong cây nhị phân tìm kiếm, chúng ta bắt đầu bằng cách gọi một hàm đệ quy phụ trợ.

```
template <class Record, int order>
Error_code B_tree<Record, order>::search_tree(Record &target)
/*
post: Nếu tìm thấy phần tử có khóa trùng với khóa trong target thì toàn bộ bản ghi phần tử
      này được chép vào target, phương thức trả về success. Ngược lại, phương thức trả về
      not_present .
uses: Hàm đệ quy phụ trợ recursive_search_tree
*/
{
   return recursive_search_tree(root, target);
}
```

Thông số vào cho hàm đệ quy phụ trợ **recursive_search_tree** là con trỏ đến gốc của cây con trong B-tree và bản ghi **target** chứa khóa cần tìm. Hàm sẽ trả về mã lỗi cho biết việc tìm kiếm kết thúc thành công hay không; nếu tìm thấy, **target** được cập nhật bởi bản ghi chứa khóa được tìm thấy trong cây.

Phương pháp chung để tìm kiếm bằng cách lần theo các con trỏ để đi xuống trong cây tương tự cách tìm kiếm trong cây nhị phân tìm kiếm. Tuy nhiên, trong một cây nhiều nhánh, chúng ta cần tốn nhiều công hơn trong việc xác định ra nhánh cần xuống tiếp theo trong mỗi nút. Việc này sẽ được thực hiện bởi một hàm phụ trợ khác của B-tree là **search_node**, hàm này tìm bản ghi có khóa trùng với khóa của **target** trong số các bản ghi có trong nút được tham chiếu bởi con trỏ **current**. Hàm **search_node** có sử dụng tham biến **position**, nếu tìm thấy, tham biến này sẽ nhận về chỉ số của bản ghi chứa khóa cần tìm trong nút tham chiếu bởi **current**; ngược lại nó chứa chỉ số của nhánh bên dưới tiếp theo cần tìm.

```
template <class Record, int order>
Error_code B_tree<Record, order>::recursive_search_tree
                      (B_node<Record, order> *current, Record &target)
/*
pre:  current là NULL hoặc chỉ đến gốc một cây con trong B_tree.
post: Nếu khóa trong target không tìm thấy, hàm trả về not_present. Ngược lại, target
      được cập nhật bởi bản ghi có chứa khóa tìm dược trong cây, hàm trả về success.
uses: Hàm phụ trợ recursive_search_tree một cách đệ quy và hàm search_node.
*/
{
   Error_code result = not_present;
```

```
   int position;
   if (current != NULL) {
      result = search_node(current, target, position);
      if (result == not_present)
         result=recursive_search_tree(current->branch[position],
                                                    target);
      else
         target = current->data[position];
   }
   return result;
}
```

Hàm trên được viết đệ quy để chứng tỏ sự tương tự giữa cấu trúc của nó với cấu trúc của hàm thêm phần tử trong phần tiếp theo dưới đây. Tuy nhiên, đây là đệ quy đuôi, và nó có thể được thay bởi cấu trúc lặp.

10.3.5.3. Tìm kiếm trong một nút

Hàm **search_node** dưới đây thực hiện việc tìm tuần tự. Hàm này cần xác định xem **target** đã có trong nút hiện tại hay chưa, nếu chưa, nó cần xác định nhánh nào trong số **count+1** nhánh là chứa **target**. Dưới đây là cách tìm tuần tự với biến tạm chạy từ 0 đến vị trí tìm thấy hoặc vừa vượt qua khóa của **target**.

```
template <class Record, int order>
Error_code B_tree<Record, order>::search_node
  (B_node<Record, order> *current, const Record &target, int &position)
/*
pre:  current chứa địa chỉ 1 nút trong B_tree.
post: Nếu khóa trong target được tìm thấy trong *current, thông số position sẽ chứa vị trí
      của phần tử target trong nút này, target được cập nhật lại, hàm trả về success.
      Ngược lại, hàm trả về not_present, position sẽ là chỉ số của nhánh con bên dưới cần
      tiếp tục việc tìm kiếm.
uses: Các phương thức của lớp Record.
*/
{
   position = 0;
   while (position < current->count && target >current->data[position])
      position++;      //  Tìm tuần tự.
   if (position < current->count && target == current->data[position])
      return success;
   else
      return not_present;
}
```

Đối với cây B-tree có các nút khá lớn, hàm trên cần được sửa đổi để sử dụng cách tìm nhị phân thay vì tìm tuần tự. Trong một vài ứng dụng, mỗi bản ghi của cây B-tree chứa rất nhiều dữ liệu, điều này làm cho bậc của cây trở nên tương đối nhỏ, và việc tìm tuần tự trong một nút là thích hợp. Trong nhiều ứng dụng khác, chỉ có các khóa là được chứa trong các nút, nên bậc của cây trở nên khá lớn, chúng ta cần dùng cách tìm nhị phân để tìm vị trí của một khóa trong một nút.

Một khả năng khác cũng có thể được xem xét, đó là việc sử dụng một cây nhị phân tìm kiếm thay cho mảng liên tục các phần tử trong mỗi nút của cây `B-tree`.

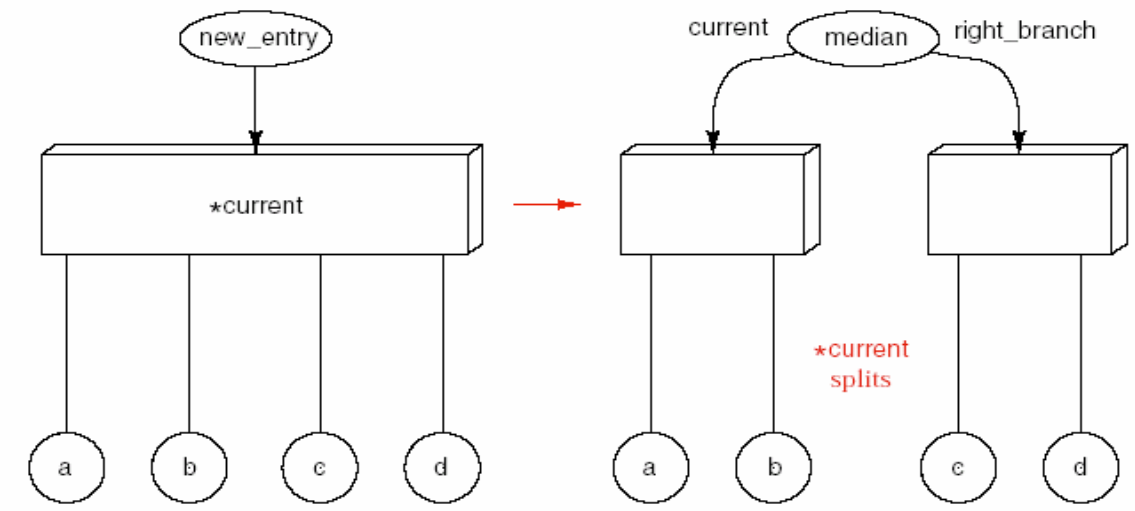
10.3.5.4. Thêm vào: phương thức `insert` và hàm đệ quy `push_down`

Việc thêm phần tử vào một cây `B-tree` có thể được xây dựng một cách tự nhiên như một hàm đệ quy. Đệ quy cho phép chúng ta giữ được vết của đường đi đến một nút trong cây, để khi quay về (khi các lần gọi đệ quy lần lượt kết thúc), chúng ta có thể thực hiện tiếp một số công việc cần thiết ở các nút thuộc mức trên theo thứ tự ngược với khi đi xuống. Nhờ vậy, chúng ta không cần sử dụng ngăn xếp một cách tường minh. Cách làm này hoàn toàn tương tự với cách mà chúng ta đã làm trong việc cân bằng lại khi thêm hoặc loại một nút trong cây cân bằng.

Như thường lệ, chúng ta cần biết chắc là khóa cần thêm chưa có trong cây. Phương thức thêm vào **`insert`** chỉ cần một thông số **`new_entry`** chứa bản ghi cần thêm. Tuy nhiên, hàm đệ quy **`push_down`** của chúng ta cần thêm ba tham biến bổ sung. Thông qua các tham biến này, một nút, sau khi gọi đệ quy xuống nút con của nó, sẽ biết được cần phải giải quyết những việc gì mà nút con của nó đã gởi gắm trở lại. Đó chính là khi một nút ở mức nào đó được phân đôi và quá trình này có thể sẽ phải lan truyền ngược về nút gốc của cây.

Hàm đệ quy **`push_down`** với thông số **`new_entry`** được gọi xuống cây con có gốc là `current` để thêm `new_entry` vào cây con này**.** Hàm `push_down` trả về `duplicate_error` nếu `new_entry` đã có trong cây; trả về `success` nếu việc thêm vào thành công và mọi chuyện đã được giải quyết triệt để trong cây con mà nó xử lý. Trong trường hợp có sự thêm `new_entry` vào cây con mà công việc còn chưa giải quyết triệt để (ngay tại nút `*current` có sự phân chia làm hai nút), hàm `push_down` sẽ trả về `overflow` để báo lên nút cha của cây con này giải quyết tiếp. Lúc đó, các tham biến sẽ có vai trò như sau. Do nút **`*current`** cần được phân đôi, chúng ta sẽ để **`current`** chỉ đến nút chứa một nửa số phần tử bên trái, và địa chỉ của nút mới chứa một nửa số phần tử bên phải sẽ được trả lên mức trên thông qua tham biến **`right_branch`**. Tham biến **`median`** được sử dụng để chứa bản ghi nằm giữa để trả lên mức trên.

Trường hợp có một nút được phân đôi được minh họa trong hình 10.11.



Hình 10.11- Hành vi của hàm push_down khi một nút được phân đôi.

Quá trình đệ quy được bắt đầu trong phương thức **insert** của B-tree. Trong trường hợp những việc cần giải quyết lan truyền lên đến tận nút gốc và lần gọi đệ quy ngoài cùng của hàm **push_down** trả về overflow, thì vẫn còn một bản ghi, **median**, cần được thêm vào cây. Một nút gốc mới cần được tạo ra để chứa bản ghi này, và chiều cao của cây B-tree tăng thêm 1. Đó là cách duy nhất để B-tree tăng chiều cao.

```
template <class Record, int order>
Error_code B_tree<Record, order>::insert(const Record &new_entry)
/*
post: Nếu khóa trong new_entry đã có trong B-tree, phương thức trả về duplicate_error.
      Ngược lại, new_entry được thêm vào cây sao cho cây vẫn thỏa điều kiện cây B-tree,
      phương thức trả về success.
uses: Các phương thức của B_node và hàm phụ trợ push_down.
*/
{
   Record median;
   B_node<Record, order> *right_branch, *new_root;
   Error_code result =push_down(root, new_entry, median, right_branch);
   if (result == overflow) {  //  Cây tăng chiều cao lên 1 đơn vị.
       //Một nút mới được tạo ra để làm gốc mới cho cây, gốc cũ của cây sẽ là gốc của cây con
         thuộc nhánh con đầu tiên của nút gốc.
      new_root = new B_node<Record, order>;
      new_root->count = 1;
      new_root->data[0] = median;
      new_root->branch[0] = root;
      new_root->branch[1] = right_branch;
      root = new_root;
      result = success;
   }
   return result;
}
```

10.3.5.5. Thêm đệ quy vào một cây con

Chúng ta hãy hiện thực hàm đệ quy **push_down**. Hàm này sử dụng con trỏ **current** tham chiếu đến gốc của cây con cần thực hiện việc tìm kiếm để thêm vào. Trong cây B-tree, bản ghi mới trước hết cần được thêm vào một nút lá. Chúng ta sẽ sử dụng điều kiện **current** == NULL để kết thúc đệ quy; nghĩa là, chúng ta sẽ tiếp tục di chuyển xuống theo cây trong khi tìm kiếm **new_entry** cho đến khi gặp phải một cây con rỗng. Do cây B-tree không lớn lên bằng cách thêm nút lá mới, chúng ta không thêm **new_entry** ngay lập tức, mà thay vào đó hàm sẽ trả về overflow, **new_entry** được gởi trả về thông qua tham biến **median** và sẽ được thêm vào một nút lá đã có ở mức trên. Việc cần làm tiếp theo cũng hoàn toàn giống với trường hợp tổng quát tại bất cứ nút nào trong cây mà chúng ta sẽ xem xét tiếp sau đây.

Khi một lần đệ quy trả về **overflow**, cũng có nghĩa là còn một bản ghi **median** vẫn chưa được thêm vào cây, và chúng ta sẽ thử thêm nó vào nút hiện tại. Nếu nút này còn chỗ trống, việc thêm sẽ hoàn tất, hàm trả về **success**. Điều này cũng làm cho các lần đệ quy trước đó sẽ lần lượt kết thúc mà không phải làm gì thêm. Ngược lại, nút ***current** được phân thành hai nút ***current** và ***right_branch,** và một bản ghi nằm giữa, **median** (có thể khác với bản ghi **median** từ lần đệ quy bên dưới trả về), được gởi ngược lên phía trên của cây, thông số trả về vẫn được giữ nguyên là **overflow**.

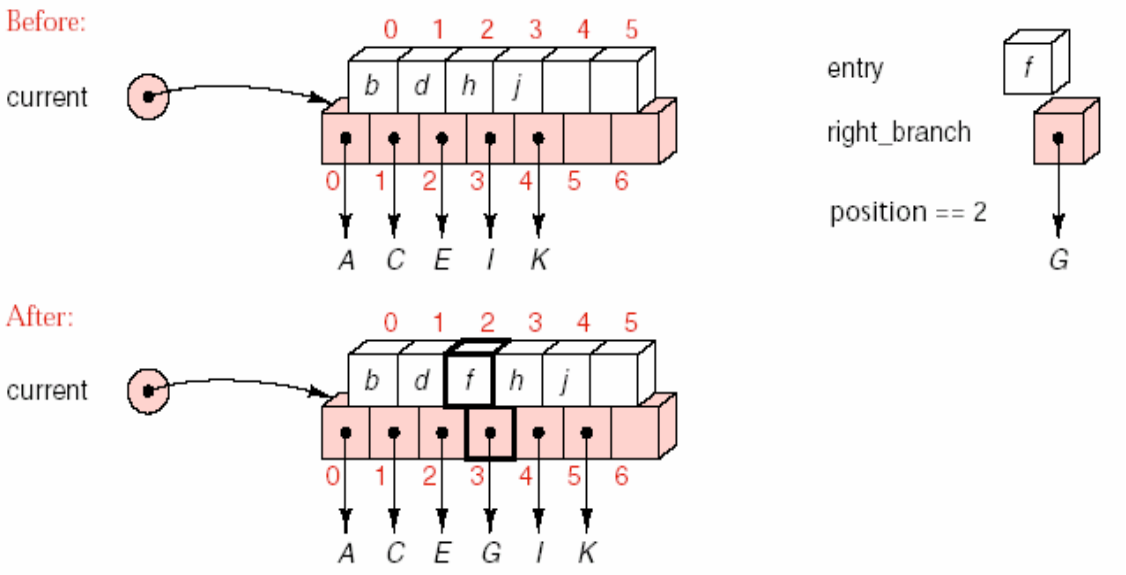
Push_down sử dụng ba hàm phụ trợ: **search_node** (giống như trong trường hợp tìm kiếm); **push_in** thêm bản ghi **median** vào nút ***current** với giả thiết rằng nút này còn chỗ trống; và **split** để chia đôi nút ***current** đã đầy thành hai nút mới, hai nút này sẽ là anh em trong cùng một mức trong cây B-tree.

```
template <class Record, int order>
Error_code B_tree<Record, order>::push_down
                                  (B_node<Record, order> *current,
                                   const Record &new_entry,
                                   Record &median,
                                   B_node<Record, order> *&right_branch)
/*
pre:  current là NULL hoặc chỉ đến một nút trong cây B_tree.
post: Nếu khóa trong new_entry đã có trong cây con có gốc current, hàm trả về
      duplicate_error. Ngược lại new_entry được chèn vào cây con, nếu diều này làm cho
      cây con cao lên, hàm trả về overflow và bản ghi median được tách ra để được chèn ở
      mức cao hơn trong cây B-tree, đồng thời right_branch chứa gốc của cây con bên phải
      bản ghi median này. Nếu cây con không cần cao lên thì hàm trả về success.
uses: Hàm push_down (một cách đệ quy), search_node, split_node, and push_in.
*/
{
   Error_code result;
   int position;
```

```
   if (current == NULL) { //  Do không thể chèn vào một cây con rỗng nên đệ quy // kết
                        thúc, việc cần làm sẽ được giải quyết ở mức trên sau đó.
      median = new_entry;
      right_branch = NULL;
      result = overflow;
   }
   else {   //   Search the current node.
      if (search_node(current, new_entry, position) == success)
         result = duplicate_error;
      else {
         Record extra_entry;
         B_node<Record, order> *extra_branch;
         result = push_down(current->branch[position], new_entry,
                            extra_entry, extra_branch);
         if (result == overflow) { // Cần giải quyết công việc nút con gởi lên.
         if (current->count < order - 1) {
               result = success;
               push_in(current, extra_entry, extra_branch, position);
            }

            else split_node(current, extra_entry, extra_branch,
                                 position, right_branch, median);
            //Bản ghi median và right_branch được cập nhật trong chính hàm này
         }
      }
   }
   return result;
}
```

10.3.5.6. Thêm một khóa vào một nút

Hàm phụ trợ kế tiếp, **push_in**, thêm bản ghi **entry** và con trỏ bên phải của nó là **right_branch** vào nút ***current**, giả sử rằng nút này còn chỗ trống để thêm vào. Hình 10.12 minh họa trường hợp này.



Hình 10.12- Hành vi của hàm `push in`.

```
template <class Record, int order>
void B_tree<Record, order>::push_in(B_node<Record, order> *current,
                                    const Record &entry, B_node<Record,
                                    order> *right_branch, int position)
/*
pre:  current chứa địa chỉ một nút trong B_tree. Nút *current chưa đầy và entry cần
      được chèn vào *current tại vị trí position, right_branch cần được cập nhật chính
      là cây con bên phải của entry trong *current.
post: entry và right_branch đã được chèn vào *current tại vị trí position.
*/
{
   for (int i = current->count; i > position; i--) {
                  // Di chuyển các phần tử cần thiết sang phải để nhường chỗ.
      current->data[i] = current->data[i - 1];
      current->branch[i + 1] = current->branch[i];
   }
   current->data[position] = entry;
   current->branch[position + 1] = right_branch;
   current->count++;
}
```

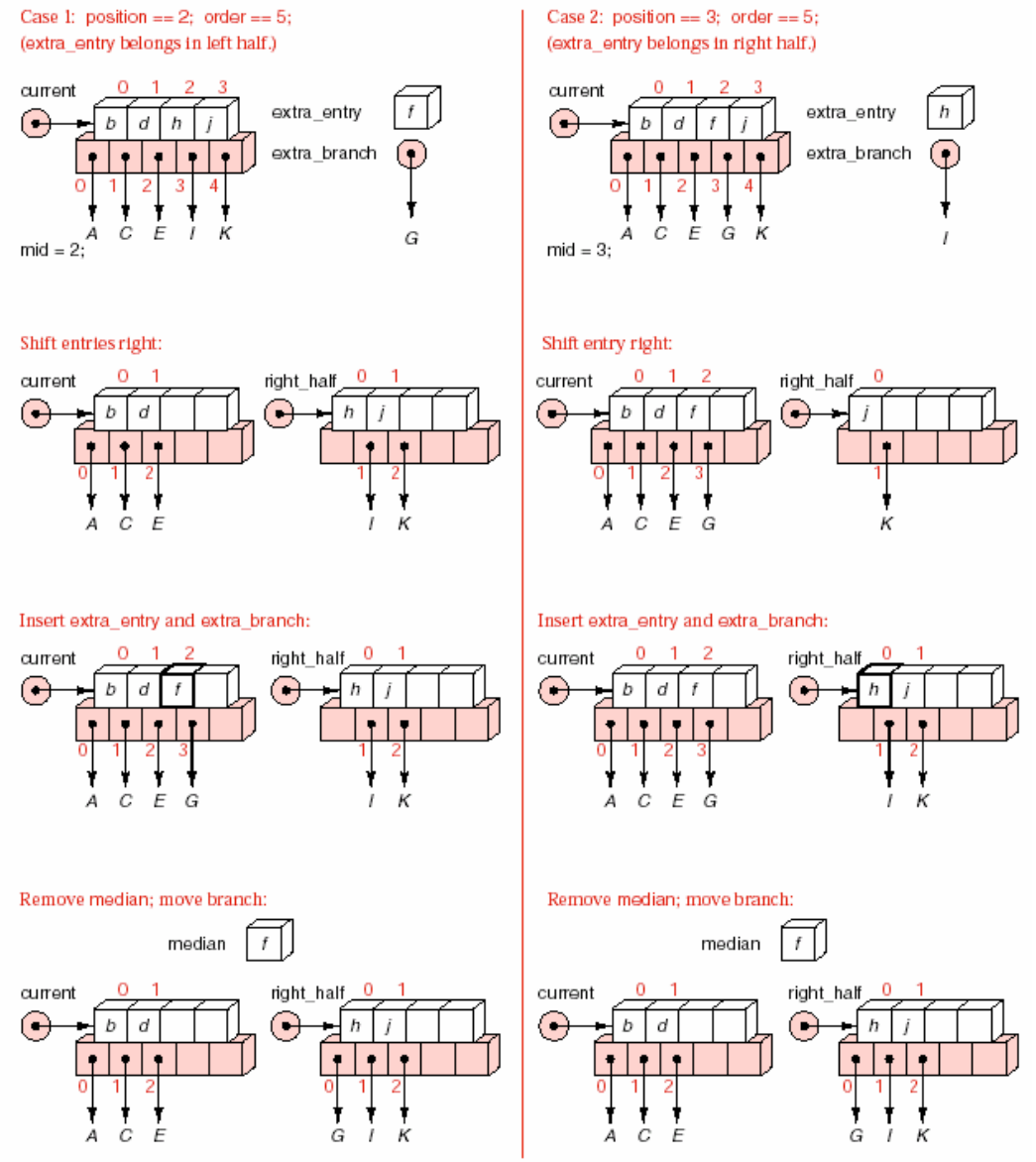
10.3.5.7. Phân đôi một nút đang đầy

Hàm phụ trợ cuối cùng cho phương thức thêm vào, **split_node**, được sử dụng khi cần thêm bản ghi **extra_entry** cùng con trỏ chỉ đến cây con **extra_branch** vào nút đã đầy ***current**. Hàm này tạo nút mới tham chiếu bởi **right_half** và chuyển một nửa số bản ghi bên phải của nút ***current** sang, gởi bản ghi nằm giữa lên phía trên của cây để nó có thể được thêm vào sau đó.

Dĩ nhiên là không thể thêm bản ghi **extra_entry** thẳng vào nút đã đầy: trước hết chúng ta cần xác định xem **extra_entry** sẽ thuộc nửa bên trái hay nửa bên phải số bản ghi sẵn có trong nút ***current**, sau đó di chuyển các bản ghi thích hợp, và cuối cùng sẽ thêm **extra_entry** vào bên tương ứng. Chúng ta sẽ chia đôi số phần tử trong nút ***current** sao cho bản ghi **median** là phần tử có khóa lớn nhất trong nửa số phần tử bên trái. Hình 10.13 minh họa điều này.

```
template <class Record, int order>
void B_tree<Record,order>::split_node
  (B_node<Record,order> *current,     // Nút cần được phân đôi.
   const Record &extra_entry,         // Phần tử mới cần chèn vào.
  B_node<Record,order>*extra_branch, // Cây con bên phải của extra_entry.
   int position,// Vị trí của extra_entry trong *current so với các phần tử đã có.
  B_node<Record,order>*&right_half,
                //Nút mới để chứa một nửa số phần tử từ *current.
   Record &median)//Phần tử giữa không nằm trong cả hai *current hoặc
  // *right_half mà sẽ được chuyển lên phía trên trong cây B_tree.
/*
pre:  current chứa địa chỉ một nút trong cây B_tree.
      Nút *current đã đầy, nhưng phần tử extra_entry cùng cây con bên phải của nó
      extra_branch cần được chèn vào vị rí position, 0 <=position <order.
```

```
post: Các phần tử đã có trong nút *current cùng với extra_entry và extra_branch (xem
      như đã được xếp vào đúng vị trí position) được phân phối vào nút *current và nút mới
      *right_half, ngoại trừ phần tử chính giữa trong số các phần tử này được đưa vào
      median.
uses: các phương thức của B_node, hàm push_in.
*/
{  right_half = new B_node<Record, order>;
   int mid = order/2;  //
   if (position <= mid){ // Trường hợp 1: extra_entry thuộc nửa bên trái.
      for (int i = mid; i < order - 1; i++){ // Chuyển các phần tử từ
             // *current sang *right_half trước rồi mới gọi hàm push_in để
             // chèn extra_entry và extra_branch vào *current sau.
         right_half->data[i - mid] = current->data[i];
         right_half->branch[i + 1 - mid] = current->branch[i + 1];
      }
      current->count = mid;
      right_half->count = order - 1 - mid;
      push_in(current, extra_entry, extra_branch, position);
   }
   else {          //  Trường hợp 2:  extra_entry thuộc nửa bên phải.
      mid++;       //  Tạm thời vẫn để phần tử cần chép vào median ở lại trong nửa bên
                                                                  trái.
      for (int i = mid; i < order - 1; i++) {// Chuyển các phần tử từ
             // *current sang *right_half trước rồi mới gọi hàm push_in để
             // chèn extra_entry và extra_branch vào *right_half sau.
         right_half->data[i - mid] = current->data[i];
         right_half->branch[i + 1 - mid] = current->branch[i + 1];
      }
      current->count = mid;
      right_half->count = order - 1 - mid;
      push_in(right_half, extra_entry, extra_branch, position - mid);
   }
      median = current->data[current->count -1];//Chép phần tử vào median
      right_half->branch[0] = current->branch[current->count];
      current->count--;
}
```



Hình 10.13 – Hành vi của hàm `split`.

10.3.6. Loại phần tử trong B-tree

10.3.6.1. Phương pháp

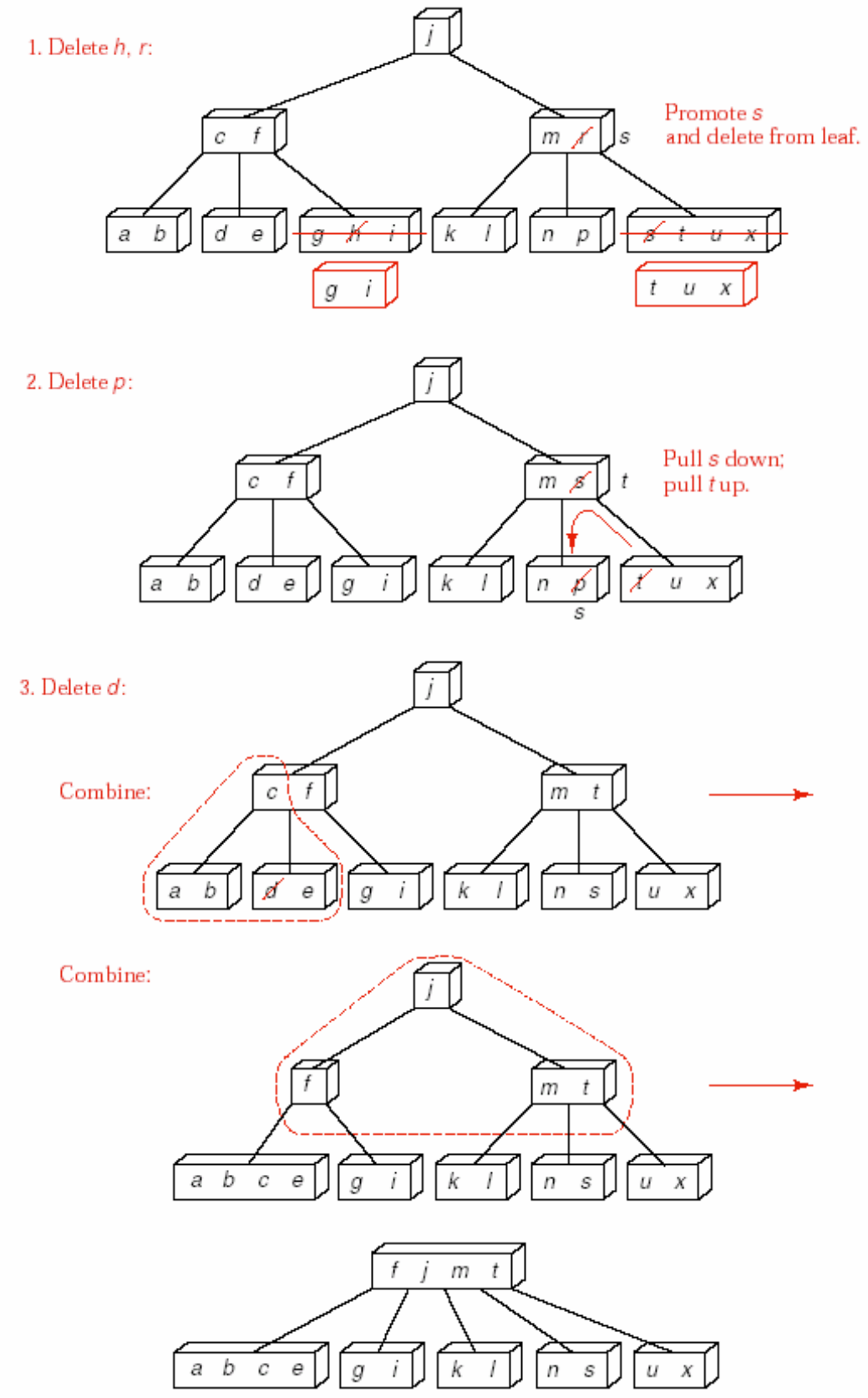
Đối với việc loại bỏ phần tử, chúng ta mong muốn rằng phần tử được loại bỏ thuộc một nút lá nào đó. Nếu phần tử này không thuộc nút lá, thì phần tử ngay kế trước nó (hoặc ngay kế sau nó) theo thứ tự tự nhiên của các khóa sẽ thuộc nút lá. Chúng ta sẽ đặt phần tử kế trước này (hoặc kế sau) thế vào chỗ của phần tử cần loại, sau đó loại vị trí của nó ra khỏi nút lá. Cách làm này rất giống với cách làm trong cây nhị phân tìm kiếm.

Nếu nút lá cần loại đi một phần tử có nhiều hơn số phần tử tối thiểu thì việc loại kết thúc. Ngược lại, nếu nút lá đang chứa số phần tử bằng số phần tử tối thiểu, thì trước hết chúng ta sẽ xem xét hai nút lá kế cận với nó và cùng một cha (hoặc chỉ một nút lá kế cận trong trường hợp nút lá đang xét nằm ở biên), nếu một trong hai có nhiều hơn số phần tử tối thiểu thì một phần tử trong số đó có thể di chuyển lên nút cha và phần tử trong nút cha sẽ di chuyển xuống nút lá đang thiếu phần tử (Chúng ta biết rằng cần phải di chuyển như vậy để **bảo đảm thứ tự giữa các phần tử**). Cuối cùng, nếu cả hai nút lá kế cận chỉ có số phần tử tối thiểu, thì nút lá đang thiếu cần kết hợp với một trong hai nút lá kế cận, **có lấy thêm một phần tử từ nút cha**, thành một nút lá mới (**Do số nút con giảm nên số phần tử trong nút cha cũng phải giảm**). Nút này sẽ chứa số phần tử không nhiều hơn số phần tử tối đa được phép. Nếu bước này làm cho nút cha còn lại số phần tử ít hơn số phần tử tối thiểu, thì việc giải quyết cũng tương tự, và quá trình này sẽ lan truyền ngược lên phía trên của cây. Quá trình lan truyền sẽ chấm dứt khi một nút cha nào đó khi cho đi một phần tử vẫn không trở nên thiếu hụt phần tử. Trong trường hợp đặc biệt, khi phần tử cuối cùng trong nút gốc bị lấy đi thì nút này cũng được giải phóng và cây sẽ giảm chiều cao.

10.3.6.2. Ví dụ

Quá trình loại bỏ trong cây `B-tree` bậc 5 sẵn có của chúng ta được minh họa trong hình 10.14. Lần loại thứ nhất không có vấn đề gì do **h** nằm trong nút lá đang có nhiều hơn số phần tử tối thiểu. Lần thứ hai, loại **r**, do **r** không thuộc nút lá, nên phần tử ngay kế sau **r** là **s** được chép đè lên **r**, và **s** được loại khỏi nút lá. Lần thứ ba, việc loại **p** làm cho nút chứa nó còn quá ít phần tử. Khóa **s** từ nút cha được chuyển xuống lấp đi sự thiếu hụt và vị trí của **s** được thế bởi **t**.

Việc loại **d** tiếp theo phức tạp hơn, nó làm cho nút còn lại quá ít phần tử, và cả hai nút kế cùng cha đều không thể sang bớt phần tử cho nó. Nút đang thiếu hụt này phải kết hợp với một trong hai nút kế, khi đó một phần tử nằm giữa chúng từ nút cha được đưa xuống (biểu diễn bởi nét rời trong sơ đồ thứ nhất của trường hợp này). Nút kết hợp được gồm các phần tử **a, b, c, e** theo sơ đồ thứ hai. Tuy nhiên, quá trình này làm cho nút cha chỉ còn lại một phần tử **f.** Ba nút phía trên của cây phải được kết hợp lại và cuối cùng chúng ta có cây như sơ đồ cuối của hình vẽ.



Hình 10.14 – Loại phần tử ra khỏi B-tree.

10.3.6.3. Hiện thực C++

Chúng ta có thể viết giải thuật loại phần tử với cấu trúc tổng thể tương tự như giải thuật thêm vào. Chúng ta sẽ sử dụng đệ quy, với một phương thức riêng để khởi động quá trình đệ quy. Thay cho việc đẩy một phần tử từ nút cha xuống trong khi gọi đệ quy xuống bên dưới, chúng ta sẽ cho hàm đệ quy trả về một nút đang thiếu phần tử thông qua tham biến. Lần gọi đệ quy bên trên sẽ phân tích điều gì đã xảy ra và thực hiện việc di chuyển các phần tử cần thiết. Khi phần tử cuối cùng trong nút gốc bị lấy đi, nút rỗng sẽ được giải phóng và chiều cao của cây giảm bớt 1.

```
template <class Record, int order>
Error_code B_tree<Record, order>::remove(const Record &target)
/*
post: Nếu khóa trong target được tìm thấy trong cây, phần tử chứa khóa này bị loại khỏi cây,
      phương thức trả về success. Ngược lại, phương thức trả về not_present.
uses: Hàm recursive_remove.
*/
{
   Error_code result;
   result = recursive_remove(root, target);
   if (root != NULL && root->count == 0) {  //  Cây giảm chiều cao.
      B_node<Record, order> *old_root = root;
      root = root->branch[0];
      delete old_root;
   }
   return result;
}
```

10.3.6.4. Loại đệ quy

Phần lớn công việc được thực hiện trong hàm đệ quy **recursive_remove**. Trước tiên nó tìm nút chứa **target**. Nếu **target** được tìm thấy và nút chứa nó không là nút lá, thì phần tử kế sau **target** sẽ được tìm và được chép đè lên nó. Phần tử kế sau này sẽ được loại bỏ. Việc loại phần tử trong nút lá được thực hiện một cách dễ dàng, và những phần công việc còn lại sẽ được tiếp tục nhờ đệ quy. Khi một lần gọi đệ quy được trả về, hàm sẽ kiểm tra xem nút tương ứng có còn đủ số phần tử hay không, nếu thiếu, nó di chuyển các phần tử để đáp ứng yêu cầu. Các hàm phụ trợ sẽ được sử dụng trong một số bước này.

```
template <class Record, int order>
Error_code B_tree<Record, order>::recursive_remove
                (B_node<Record, order> *current, const Record &target)
/*
pre:  current là NULL hoặc chứa địa chỉ nút gốc của một cây con trong B_tree.
post: Nếu khóa trong target được tìm thấy trong cây, phần tử chứa khóa này bị loại khỏi cây
      sao cho cây vẫn giữ tính chất cây B-tree, phương thức trả về success. Ngược lại,
      phương thức trả về not_present.
uses: Các hàm search_node, copy_in_predecessor,
      recursive_remove (một cách đệ quy), remove_data, và restore.
*/
```

```
{
   Error_code result;
   int position;
   if (current == NULL) result = not_present;
   else {
      if (search_node(current, target, position) == success){
                           // Khóa trong target được tìm thấy trong current.
         result = success;
         if (current->branch[position] !=NULL){//current không phải nút lá
            copy_in_predecessor(current, position);

            recursive_remove(current->branch[position],
                             current->data[position]);
         }
         else remove_data(current, position);// Loại phần tử trong nút lá.
      }
      else result =recursive_remove(current->branch[position], target);
      if (current->branch[position] != NULL)
         if (current->branch[position]->count < (order - 1) / 2)
            restore(current, position);
   }
   return result;
}
```

10.3.6.5. Các hàm phụ trợ

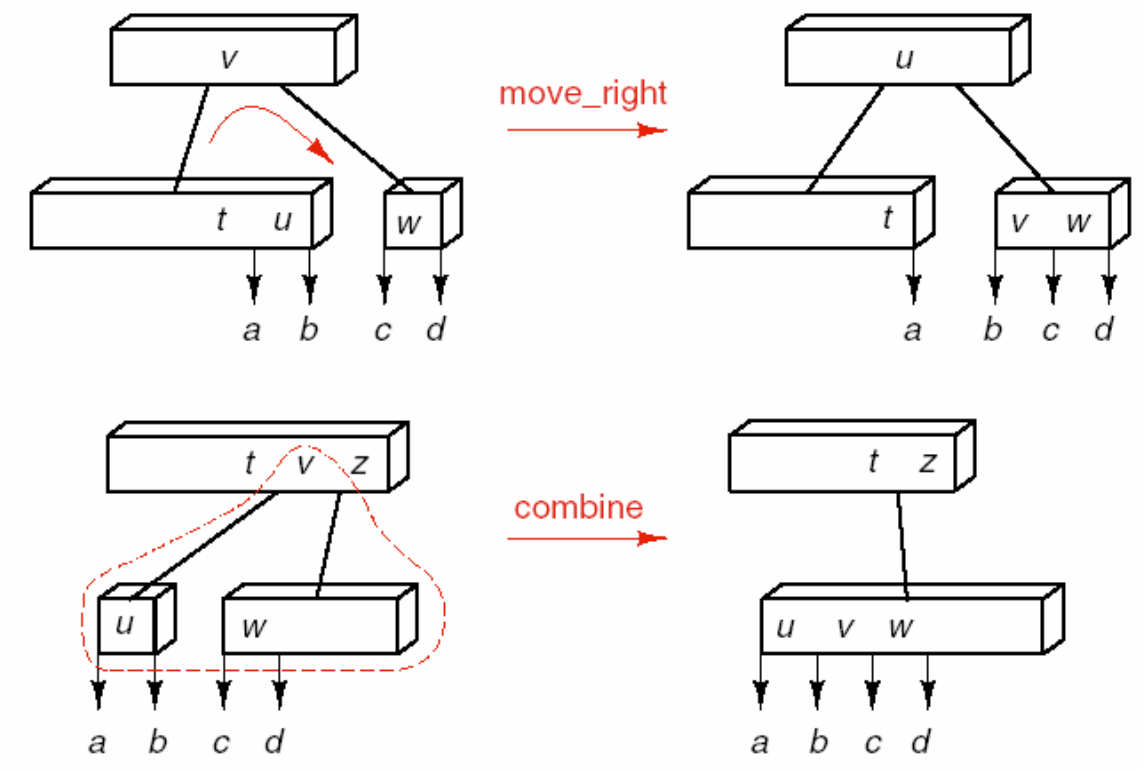
Giờ chúng ta đã có thể kết thúc quá trình loại bỏ trong một cây *B-tree* bằng cách viết các hàm phụ trợ cho nó. Hàm **remove_data** loại một phần tử và nhánh bên phải của nó khỏi một nút trong cây *B-tree*. Hàm này được gọi chỉ trong trường hợp khi một phần tử được loại khỏi một nút lá của cây.

```
template <class Record, int order>
void B_tree<Record, order>::remove_data(B_node<Record, order> *current,
                                        int position)
/*
pre:  current là địa chỉ nút lá có entry tại vị trí position.
post: entry được loại khỏi nút *current.
*/
{
   for (int i = position; i < current->count - 1; i++)
      current->data[i] = current->data[i + 1];
   current->count--;
}
```

Hàm **copy_in_predecessor** được gọi khi một phần tử cần được loại ra khỏi một nút không phải nút lá. Trong trường hợp này, phần tử ngay kế trước (theo thứ tự các khóa) sẽ được tìm bằng cách bắt đầu từ nhánh bên trái của nó và đi xuống theo các nhánh tận cùng bên phải của mỗi nút cho đến khi gặp nút lá. Phần tử tận cùng bên phải của nút lá này sẽ thay thế phần tử cần được loại.

```
template <class Record, int order>
void B_tree<Record, order>::copy_in_predecessor
                          (B_node<Record, order> *current, int position)
/*
pre:  current là địa chỉ nút không phải nút lá trong B-tree và có entry là phần tử cần loại
      tại position.
post: entry được thay thế bởi phần tử đứng ngay kế trước nó trong thứ tự tăng dần của khóa.
*/
{
   B_node<Record, order> *leaf = current->branch[position]; // Di chuyển qua
                        nhánh trái để tìm phần tử ngay kế trước entry.
   while (leaf->branch[leaf->count] != NULL)
      leaf = leaf->branch[leaf->count]; // Xuống phần tử cực phải của nhánh trái của
                                                              current.
   current->data[position] = leaf->data[leaf->count - 1];
}
```

Cuối cùng, chúng ta cần chỉ ra cách khôi phục lại số nút tối thiểu cho nút được tham chiếu bởi **`root->branch[position]`** nếu như lần đệ quy bên trong làm cho nó trở nên thiếu phần tử. Hàm chúng ta viết dưới đây hơi thiên về bên trái, nghĩa là, trước hết nó tìm nút anh em kề bên trái để xin bớt phần tử, và nó chỉ xét đến nút anh em kề bên phải khi nút kề bên trái không thừa phần tử. Các bước thực hiện được minh họa trong hình 10.15.



Hình 10.15 – Khôi phục lại số phần tử tối thiểu trong một nút.

```
template <class Record, int order>
void B_tree<Record, order>::restore(B_node<Record, order> *current,
                                    int position)
/*
pre:  current là địa chỉ một nút không phải nút lá trong B-tree;
      current->branch[position] là địa chỉ nút đang bị thiếu một phần tử.
post: Nút do current->branch[position] chỉ đến được khôi phục lại cho đủ số phần tử tối
      thiểu cần có.
uses: Các hàm move_left, move_right, combine.
*/
{  if (position == current->count)// Trường hợp không có nút anh em bên phải.
      if (current->branch[position - 1]->count > (order - 1) / 2)
         move_right(current, position - 1);
      else
         combine(current, position);
   else if (position == 0)        // Trường hợp không có nút anh em bên trái.
      if (current->branch[1]->count > (order - 1) / 2)
         move_left(current, 1);
      else
         combine(current, 1);
   else                           // Trường hợp có nút anh em cả hai bên.
      if (current->branch[position - 1]->count > (order - 1) / 2)
         move_right(current, position - 1);
      else if (current->branch[position + 1]->count > (order - 1) / 2)
         move_left(current, position + 1);
      else
         combine(current, position);
}
```

Các hành vi của ba hàm còn lại **move_left**, **move_right**, và **combine** được thể hiện rất rõ ràng trong hình 10.15.

```
template <class Record, int order>
void B_tree<Record, order>::move_left(B_node<Record, order> *current,
                                      int position)
/*
pre:  current là địa chỉ một nút trong B-tree, nhánh con tại position có nút gốc có số
      phần tử nhiều hơn số phần tử tối thiểu theo quy định, nhánh con tại position–1 có nút
      gốc đang thiếu 1 phần tử.
post: Một phần tử của *current di chuyển xuống nút gốc của nhánh con tại
      position–1, phần tử tại vị trí 0 trong nút gốc của nhánh tại position di chuyển lên
      *current (cây con tại vị trí 0 cũng như các phần tử còn lại trong nút gốc của nhánh này
      sẽ được dịch chuyển hợp lý).
*/
{  B_node<Record, order> *left_branch = current->branch[position - 1],
                         *right_branch = current->branch[position];
   left_branch->data[left_branch->count] = current->data[position - 1];
                            // Lấy một entry từ nút cha *current.
   left_branch->branch[++left_branch->count] = right_branch->branch[0];
   // Giải quyết cho cây con tại vị trí 0 trong nhánh con bên phải: số phần tử trong nhánh con
   // bên trái tăng thêm 1 nên số cây con cũng phải tăng thêm 1, đồng thời cách di chuyển này
   // vẫn bảo đảm thứ tự các khóa trong cây.
   current->data[position - 1] = right_branch->data[0];
                     // Nút cha *current lấy một entry từ nhánh con bên phải.
```

```
    right_branch->count--;
   for (int i = 0; i < right_branch->count; i++) {
                    // Dịch chuyển tất cả các phần tử về bên trái để lấp chỗ trống.
      right_branch->data[i]   = right_branch->data[i + 1];
      right_branch->branch[i] = right_branch->branch[i + 1];
   }
   right_branch->branch[right_branch->count] =
                              right_branch->branch[right_branch->count + 1];
}


template <class Record, int order>
void B_tree<Record, order>::move_right(B_node<Record, order> *current,
                                       int position)
/*
pre:  current là địa chỉ một nút trong B-tree, nhánh con tại position có nút gốc có số
      phần tử nhiều hơn số phần tử tối thiểu theo quy định, nhánh con tại position+1 có nút
      gốc đang thiếu 1 phần tử.
post: Một phần tử của *current di chuyển xuống nút gốc của nhánh con tại
      position+1. Phần tử có khóa lớn nhất (tại count-1) trong nút gốc của nhánh tại
      position di chuyển lên *current (cây con bên phải của nó được bố trí lại hợp lý).
*/
{
   B_node<Record, order> *right_branch = current->branch[position + 1],
                        *left_branch = current->branch[position];
   right_branch->branch[right_branch->count + 1] =
      right_branch->branch[right_branch->count];
   for (int i = right_branch->count ; i > 0; i--) {
      // Di chuyển sang phải để dành chỗ trống cho phần tử từ *current đưa xuống.
      right_branch->data[i] = right_branch->data[i - 1];
      right_branch->branch[i] = right_branch->branch[i - 1];
   }
   right_branch->count++;
   right_branch->data[0] = current->data[position];
                                  //  Nhận entry từ nút cha *current.
   right_branch->branch[0] = left_branch->branch[left_branch->count];
      // Bố trí lại cây con thừa ở nhánh trái do số phần tử giảm bớt 1 (thứ tự các khóa trong
                                                              B-tree vẫn bảo đảm).
   left_branch->count--;
   current->data[position] = left_branch->data[left_branch->count];
}
```

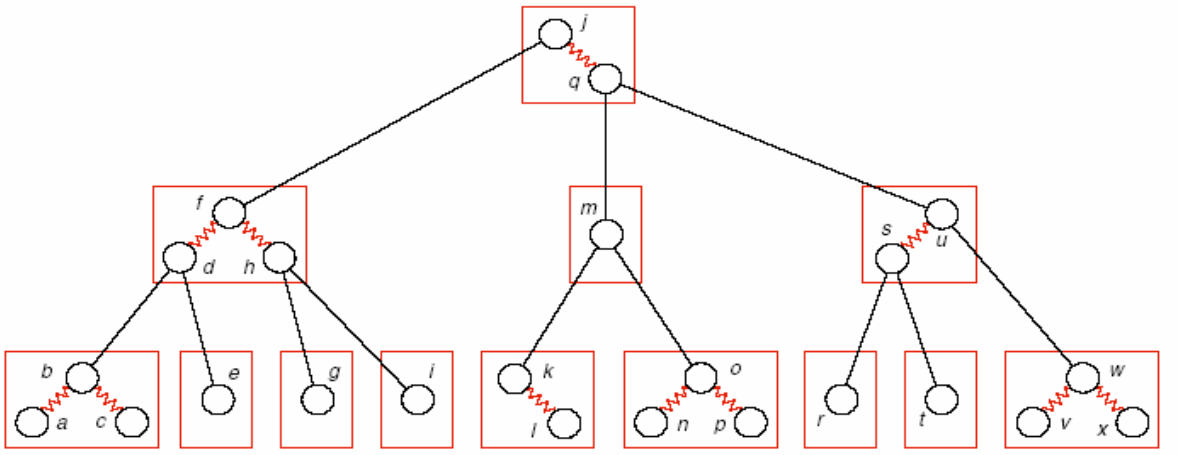
```
template <class Record, int order>
void B_tree<Record, order>::combine(B_node<Record, order> *current,
                                    int position)
/*
pre:  current chứa địa chỉ một nút trong B-tree có các nút gốc của hai nhánh con tại
      position và position – 1 cần ghép (lại do không đủ số phần tử để di chuyển qua lại
      sao cho cả 2 nút vẫn đủ số phần tử tối thiểu).
post: Hai nút gốc của hai nhánh tại position-1 và position được ghép lại (phần tử tại vị trí
      position-1 của *current cũng di chuyển xuống nút này để bảo đảm rằng số phần tử
      trong *current giảm bớt 1 khi số nhánh con giảm bớt 1).
*/


{
```

```
   int i;
   B_node<Record, order> *left_branch = current->branch[position - 1],
                        *right_branch = current->branch[position];
   left_branch->data[left_branch->count] = current->data[position - 1];
   left_branch->branch[++left_branch->count] = right_branch->branch[0];
   for (i = 0; i < right_branch->count; i++) {
      left_branch->data[left_branch->count] = right_branch->data[i];
      left_branch->branch[++left_branch->count] =
                                          right_branch->branch[i + 1];
   }
   current->count--;
   for (i = position - 1; i < current->count; i++) {
      current->data[i] = current->data[i + 1];
      current->branch[i + 1] = current->branch[i + 2];
   }
   delete right_branch;
}
```

10.4. Cây đỏ-đen

10.4.1. Dẫn nhập

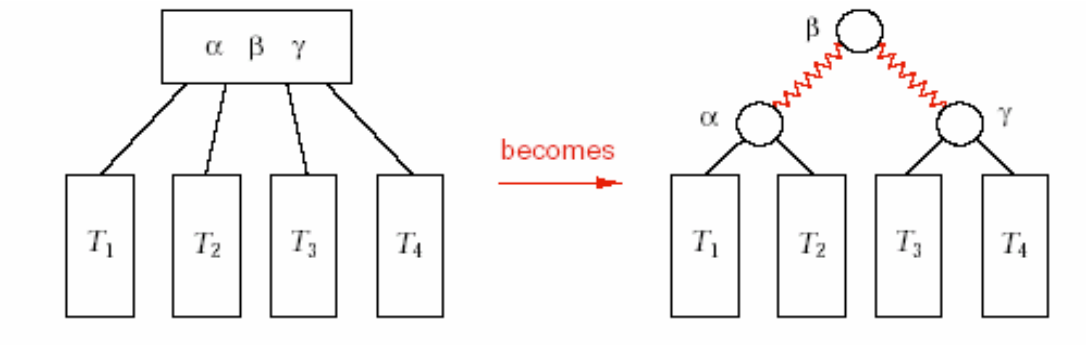
Trong phần trước, chúng ta đã sử dụng danh sách liên tục để chứa các phần tử của cây `B-tree`. Tuy nhiên, nói một các tổng quát, chúng ta có thể dùng bất kỳ cấu trúc có thứ tự nào để chứa các phần tử trong mỗi nút của `B-tree`. Một cây nhị phân tìm kiếm nhỏ là một lựa chọn tốt. Chúng ta chỉ cần chú ý phân biệt các con trỏ bên trong mỗi nút của cây `B-tree` (nối các nút của cây nhị phân tìm kiếm) với các con trỏ từ nút này đến nút khác của `B-tree`. Chúng ta hãy vẽ các tham chiếu bên trong một nút bằng các **đường xoắn màu đỏ** và những con trỏ giữa các nút trong cây `B-tree` bằng các **đường thẳng màu đen**. Xem hình 10.16.



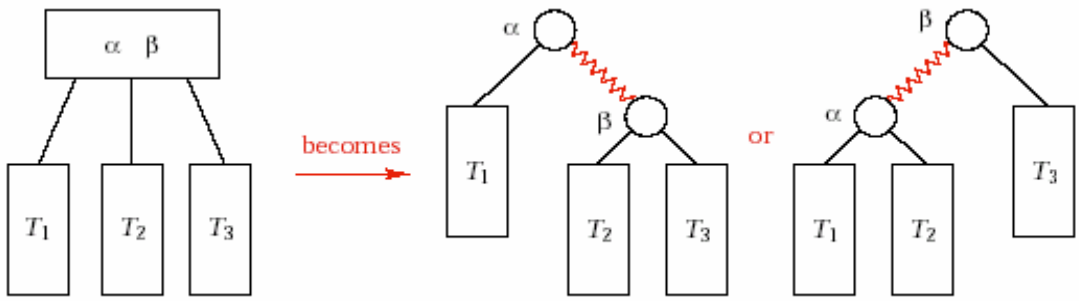
Hình 10.16 – Cây `B-tree` bậc 4 như một cây tìm kiếm nhị phân.

10.4.2. Định nghĩa và phân tích

Cấu trúc này đặc biệt có ích đối với cây B-tree bậc 4 (hình 10.16), trong đó mỗi nút của cây chứa một, hai hoặc ba phần tử. Trường hợp một nút có một phần tử thì tương tự như trong cây B-tree và cây nhị phân tìm kiếm. Trường hợp một nút có ba phần tử được biến đổi như sau:



Một nút có hai phần tử có thể có hai biểu diễn:



Nếu muốn, chúng ta chỉ cần sử dụng một trong hai cách biểu diễn trên, nhưng không có lý do gì để làm điều đó, chúng ta sẽ thấy rằng các giải thuật của chúng ta sẽ sinh ra cả hai cách biểu diễn này một cách tự nhiên. Như vậy chúng ta sẽ sử dụng cả hai cách biểu diễn cho các nút có hai phần tử trong cây B-tree.

Chúng ta có định nghĩa cơ bản cho phần này như sau: Một cây đỏ-đen (*red-black tree*) là một cây nhị phân tìm kiếm, với các tham chiếu có màu đỏ hoặc đen, có được từ một cây B-tree bậc bốn bằng cách vừa được mô tả trên.

Sau khi chuyển đổi một cây B-tree thành cây đỏ đen, chúng ta có thể sử dụng nó tương tự bất kỳ cây nhị phân tìm kiếm nào. Việc **tìm kiếm** và **duyệt** cây đỏ đen hoàn toàn giống như đối với cây nhị phân tìm kiếm; chúng ta chỉ đơn giản bỏ qua các màu của các tham chiếu. Tuy nhiên, việc **thêm và loại phần tử** đòi hỏi

nhiều công sức hơn để duy trì cấu trúc của một cây `B-tree`. Chúng ta hãy chuyển đổi các yêu cầu đối với cây `B-tree` thành các yêu cầu tương ứng đối với cây đỏ đen.

Trước hết, chúng ta hãy lưu ý một số điểm: chúng ta sẽ xem mỗi nút trong cây đỏ đen cũng **có màu như màu của tham chiếu đến nó**, như vậy chúng ta sẽ gọi các nút màu đỏ và các nút màu đen thay vì gọi các tham chiếu đỏ và các tham chiếu đen. Bằng cách này, chúng ta chỉ cần giữ thêm một thông tin cho mỗi nút để chỉ ra màu của nó.

Do nút gốc không có tham chiếu đến nó, **nó sẽ không có màu**. Để làm đơn giản một số giải thuật, **chúng ta quy ước rằng nút gốc có màu đen**. Tương tự, chúng ta sẽ xem tất cả **các cây rỗng** (tương ứng tham chiếu `NULL`) **có màu đen**.

Theo điều kiện thứ nhất trong định nghĩa của cây `B-tree`, mọi cây con rỗng phải thuộc cùng mức, nghĩa là mọi đường đi từ gốc đến mỗi cây con rỗng sẽ đi qua cùng một số **nút `B-tree`** như nhau. Mỗi **nút `B-tree`** trong cây đỏ đen luôn có một nút đen, do các con trỏ giữa các nút trong cây `B-tree` được biểu diễn bằng các đường thẳng màu đen. Đối với **nút `B-tree` có nhiều hơn một nút thì ngoài một nút đen, các nút còn lại phải có màu đỏ**. Từ đó chúng ta có "điều kiện đen" như sau:

> *Mọi đường đi từ gốc đến mỗi cây con rỗng đều đi qua cùng một số nút đen như nhau.*

Do cây `B-tree` thoả các đặc tính của cây tìm kiếm, nên cây đỏ đen cũng thỏa đặc tính này. Những phần còn lại trong định nghĩa đối với cây `B-tree` bậc 4 nói lên rằng mỗi nút chứa một, hai hoặc ba phần tử dữ liệu. Chúng ta cần một điều kiện trên cây đỏ đen để bảo đảm rằng khi các nút trong cây này được gom lại thành các **nút `B-tree`** thì mỗi **nút `B-tree`** có không quá ba nút. Nếu một **nút `B-tree`** có hai **nút của cây đỏ đen**, thì trong đó sẽ có một nút cha và một nút con, nếu một **nút `B-tree`** có ba **nút của cây đỏ đen**, thì trong đó sẽ có một nút cha và hai nút con (theo hình vẽ các cách biểu diễn trên). Nút cha trong cả hai trường hợp trên phải luôn có màu đen do tham chiếu đến nó chính là con trỏ giữa các **nút `B-tree`**. Như vậy, chúng ta thấy trong cây đỏ đen không thể có một nút đỏ mà có nút cha màu đỏ. Vậy "điều kiện đỏ" như sau đây sẽ bảo đảm rằng **mọi nút `B-tree` trong cây đỏ đen có không quá ba nút:**

Nếu một nút có màu đỏ, thì nút cha của nó phải tồn tại và có màu đen.

Do chúng ta quy ước nút gốc có màu đen nên điều kiện trên vẫn thỏa.

Chúng ta có thể tổng kết lại các điều phân tích trên đây thành một định nghĩa hình thức cho cây đỏ đen như sau (và chúng ta không cần nhắc đến cây `B-tree` nữa khi nói đến cây đỏ đen):

Định nghĩa: Một cây đỏ đen là một cây tìm kiếm nhị phân, trong đó mỗi nút có màu đỏ hoặc đen, thỏa các điều kiện sau:
1. Mọi đường đi từ nút gốc đến mỗi cây con rỗng (tham chiếu `NULL`) đều đi qua cùng một số nút đen như nhau.
2. Nếu một nút có màu đỏ thì nút cha của nó phải tồn tại và có màu đen.

Định nghĩa này dẫn đến một điều là trong cây đỏ đen không có một đường đi nào từ gốc đến một cây con rỗng có thể dài hơn gấp đôi một đường đi khác, bởi vì, theo điều kiện đen, số nút đen của tất cả các đường đi này phải bằng nhau, và theo điều kiện đỏ thì số nút đỏ phải nhỏ hơn hay bằng số nút đen. Do đó, ta có định lý sau:

Định lý: Chiều cao của một cây đỏ đen n nút không lớn hơn 2 lg n.

Thời gian tìm kiếm trong một cây đỏ đen có n nút là $O(\log n)$ trong mọi trường hợp. Chúng ta cũng sẽ thấy rằng thời gian thêm nút mới cũng là $O(\log n)$, nhưng trước hết chúng ta cần phát triển giải thuật trước.

Chúng ta nhớ lại trong phần 10.4, cây AVL, trong trường hợp xấu nhất, có chiều cao bằng 1.44 lg n, và trong trường hợp trung bình, có chiều cao thấp hơn. Sự khác nhau về chiều cao liên quan đến số nút của hai cây này là do cây đỏ đen không cân bằng tốt bằng cây AVL. Tuy nhiên, điều này không có nghĩa là các thao tác dữ liệu trên cây đỏ đen nhất thiết phải chậm hơn cây AVL, do cây AVL có thể cần đến nhiều phép quay để duy trì sự cân bằng hơn những gì mà cây đỏ đen cần đến.

10.4.3. Đặc tả cây đỏ đen

Để đặc tả một lớp C++ nhằm biểu diễn cho các đối tượng của cây đỏ đen, chúng ta cần khảo sát một vài tác vụ trên chúng. Chúng ta có thể hiện thực cây đỏ đen như là cây `B-tree` mà các nút của nó chứa các cây tìm kiếm thay vì các danh sách liên tục. Cách tiếp cận này buộc chúng ta phải viết lại nhiều phương thức và hàm phụ trợ đã có đối với cây `B-tree`, do phiên bản trước đây của `B-tree` liên quan chặt chẽ đến hiện thực liên tục của các phần tử trong mỗi nút.

Vì thế chúng ta sẽ đề xuất một hiện thực lớp cây đỏ đen thừa kế các đặc tính của lớp cây tìm kiếm trong phần 10.2.

Chúng ta bắt đầu bằng cách thêm thuộc tính màu vào mỗi nút của cây đỏ đen:

```
enum Color {red, black};

template <class Record>
struct RB_node: public Binary_node<Record> {
   Color color;
   RB_node(const Record &new_entry) { color = red; data = new_entry;
                                      left = right = NULL; }
   RB_node()                { color = red; left = right = NULL; }
   void set_color(Color c)  { color = c; }
   Color get_color() const  { return color; }
};
```

Để thuận tiện, chúng ta sử dụng các định nghĩa trong dòng (*inline definition*) cho các *contructor* và một số phương thức khác của **RB_node**. Cấu trúc **struct RB_node** rất giống với cấu trúc **struct AVL_node** dùng trong cây AVL trước kia trong phần 10.4: sự khác nhau duy nhất chỉ là thuộc tính màu thay cho thuộc tính cân bằng.

Để có thể gọi các phương thức **get_color** và **set_color** thông qua các con trỏ chỉ đến **Binary_node**, chúng ta cần bổ sung các hàm ảo tương ứng trong **struct Binary_node**, tương tự như chúng ta đã làm khi xây dựng cây AVL.

Cấu trúc của **Binary_node** đã sửa đổi như sau:

```
template <class Entry>
struct Binary_node {
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
   virtual Color get_color() const { return red; }
   virtual void set_color(Color c) { }
   Binary_node()                   { left = right = NULL; }
   Binary_node(const Entry &x)     { data = x; left = right = NULL; }
};
```

Bằng cách sửa đổi như vậy, chúng ta đã có thể sử dụng lại mọi phương thức và hàm xử lý cho cây nhị phân tìm kiếm và các nút của nó. Việc tìm kiếm và duyệt trên cây đỏ đen tương tự như đối với cây nhị phân tìm kiếm.

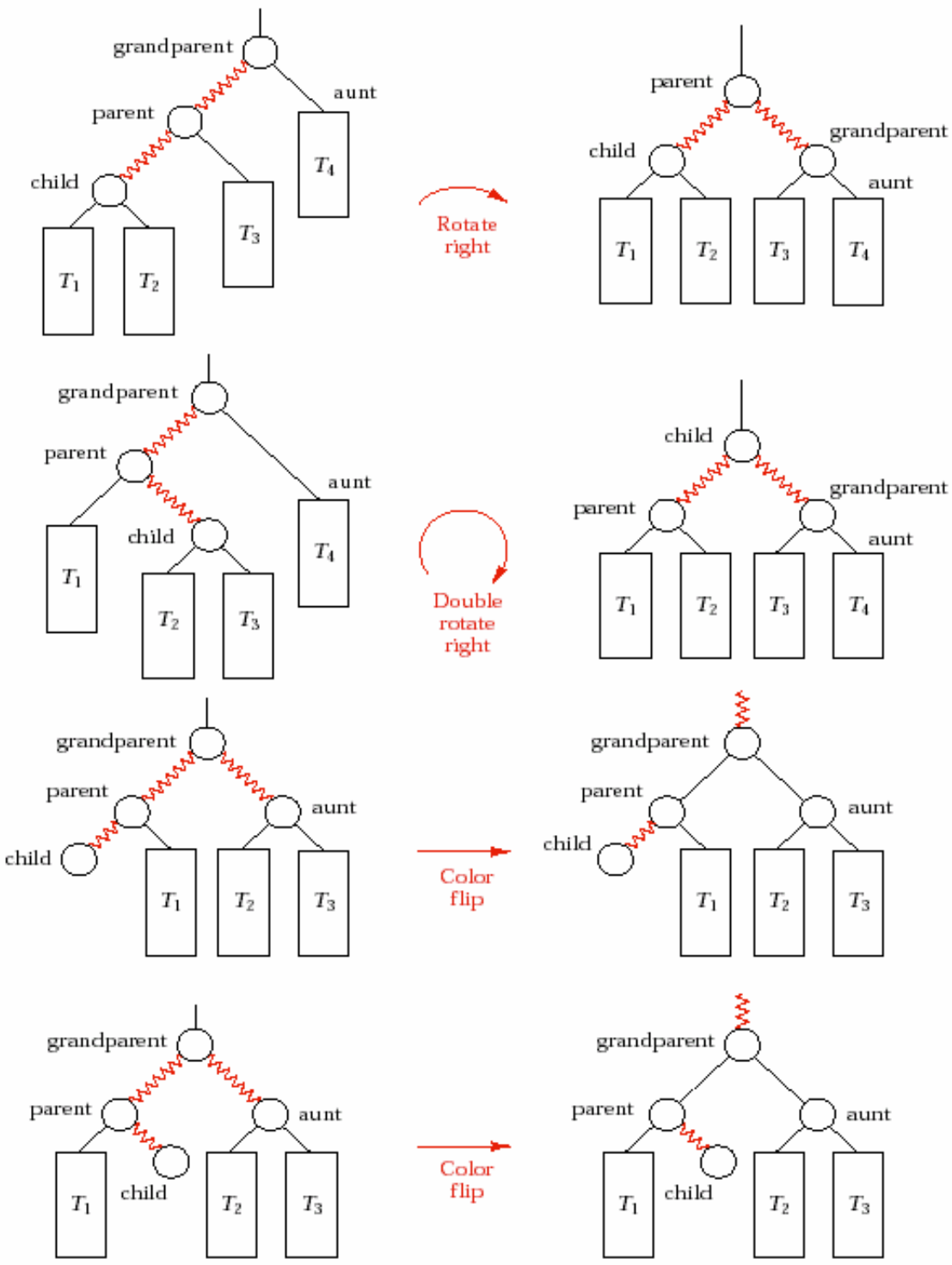
Chúng ta sẽ viết phương thức thêm phần tử vào cây đỏ đen sao cho nó vẫn giữ được tính chất của cây đỏ đen sau khi thêm vào.

```
template <class Record>
class RB_tree: public Search_tree<Record> {
```

```
public:
   Error_code insert(const Record & new_entry);
private:  //  Các hàm phụ trợ.
};
```

10.4.4. Thêm phần tử

Chúng ta hãy bắt đầu từ giải thuật đệ quy chuẩn đối với việc thêm vào một cây tìm kiếm nhị phân. Nghĩa là, chúng ta sẽ so sánh khóa mới của **target** với khóa của gốc, nếu cây không rỗng, và sau đó thêm đệ quy nút mới vào cây con trái hoặc cây con phải của gốc. Quá trình này kết thúc khi chúng ta gặp một cây con rỗng, tại đó chúng ta sẽ tạo một nút mới.



Hình 10.17 – Khôi phục các điều kiện đỏ và đen.

Nút mới này nên là màu đỏ hay là màu đen? Nếu cho nó màu đen, chúng ta đã làm tăng số nút đen trên đường đi từ gốc đến nó, và vi phạm điều kiện đen. Vậy nút mới phải có màu đỏ. Như **vậy khi thêm một nút mới vào cây đỏ đen thì**

nó luôn có màu đỏ. Nếu nút cha của nút mới này có màu đen, việc thêm vào kết thúc. Ngược lại, điều kiện đỏ bị vi phạm, và điều vi phạm này cần phải được giải quyết. Chúng ta sẽ phân tích các trường hợp và sẽ xử lý riêng rẽ cho chúng.

Giải thuật của chúng ta khá là đơn giản nếu chúng ta không xem xét các trường hợp này ngay lập tức mà trì hoãn chúng lại trong chừng mực có thể. Khi tạo một nút đỏ, chúng ta sẽ không cố gắng điều chỉnh lại cây ngay, thay vào đó, chúng ta chỉ đơn giản trả về một chỉ số trạng thái cho biết nút vừa xử lý xong có màu đỏ.

Khi một lần gọi đệ quy kết thúc, chỉ số trạng thái này được gởi ngược về lần đệ quy đã gọi nó, việc xử lý sẽ được thực hiện ở nút cha. Nếu nút cha có màu đen, các điều kiện của cây đỏ đen không bị vi phạm, quá trình xử lý kết thúc. Nếu nút cha có màu đỏ, chúng ta cũng sẽ không giải quyết ngay, mà một lần nữa lại gán chỉ số trạng thái cho biết vừa có hai nút màu đỏ. Lần đệ quy bên trên tại nút ông sẽ nhận được chỉ số trạng thái này, có kèm thêm thông tin cho biết hai nút màu đỏ vừa rồi thuộc cây con trái hay cây con phải của nó.

Sau khi các lần đệ quy trả về đến nút ông, việc xử lý sẽ được tiến hành tại đây. Chúng ta biết rằng nút ông luôn có màu đen, do trước khi xuất hiện nút con màu đỏ, các màu của nút cha và nút ông phải thỏa điều kiện của cây đỏ đen, mà nút cha đỏ thì nút ông phải đen. Quy ước nút gốc màu đen có lợi trong trường hợp này, vì nếu nút cha màu đỏ, thì nó không thể là nút gốc. Khi quá trình xử lý lan truyền về tận gốc thì nút ông phải luôn luôn tồn tại.

Cuối cùng, tại lần đệ quy của nút ông, chúng ta có thể biến đổi cây để khôi phục lại các điều kiện đỏ đen. Chúng ta chỉ cần xem xét trường hợp hai nút đỏ thuộc cây con trái của nút ông. Trường hợp ngược lại chỉ là đối xứng. Chúng ta cần phân biệt hai trường hợp tương ứng với màu của nút con còn lại của nút ông, nghĩa là nút chú của nút đỏ mới xuất hiện.

Trước tiên, giả sử nút chú có màu đen. Trường hợp này có gộp cả trường hợp nút chú rỗng, do quy ước cây con rỗng có màu đen. Các đặc tính của cây đỏ đen sẽ được khôi phục nhờ một phép quay đơn hay một phép quay kép qua phải, như hai phần đầu của hình 10.17. Trong cả hai sơ đồ này, phép quay, kéo theo sự thay đổi các màu nút tương ứng, sẽ loại được sự vi phạm điều kiện đỏ, đồng thời bảo toàn điều kiện đen do không làm thay đổi số nút đen trong bất kỳ đường đi nào từ gốc đến các nút lá.

Giờ chúng ta giả sử nút chú có màu đỏ, như hai phần bên dưới của hình 10.17. Việc biến đổi rất đơn giản: không có phép quay nào xảy ra, chỉ có sự thay đổi các màu. Nút cha và nút chú trở thành màu đen, nút ông trở thành màu đỏ. Điều kiện đỏ vẫn bảo đảm khi xét mối quan hệ giữa nút ông và nút cha, nút chú; giữa nút

cha và nút con màu đỏ mới xuất hiện bên dưới. Điều kiện đen không bị vi phạm do số nút đen trong các đường đi dọc theo cây không hề thay đổi. Tuy nhiên, khi nút ông vừa được biến đổi thành màu đỏ, điều kiện đỏ có thể bị vi phạm khi xét mối quan hệ với nút cha của nút này. Quá trình xử lý chưa thể chấm dứt. Mặc dù vậy, chúng ta có thể thấy rằng nút ông vừa được đổi sang màu đỏ hoàn toàn giống với trường hợp một nút đỏ mới xuất hiện lúc trước. Vậy chúng ta chỉ cần để lại cho các lần gọi đệ quy bên trên xử lý tại các nút bên trên nữa trong cây, bằng cách lại cho chỉ số trạng thái về lại trường hợp có một nút đỏ mới xuất hiện. Như vậy quá trình xử lý khi điều kiện đỏ bị vi phạm được lan truyền dọc lên phía trên của cây. Quá trình này có thể kết thúc tại một nút nào đó, hoặc có thể lan truyền lên tận gốc. Và khi nút gốc cần được đổi sang màu đỏ, thì chương trình ngoài cùng chỉ cần đổi nút này thành màu đen cho đúng quy ước. Điều này cũng không vi phạm điều kiện đen, do nó làm cho số nút đen trong tất cả các đường đi dọc theo cây tăng thêm một đơn vị. Và đây cũng chính là trường hợp duy nhất làm cho số nút đen trong các đường đi này tăng lên.

10.4.5. Phương thức thêm vào. Hiện thực

Chúng ta sẽ chuyển giải thuật trên thành chương trình C++. Cũng như mọi khi, phần lớn công việc đều được thực hiện bởi hàm đệ quy, phương thức **insert** chỉ cần đổi nút gốc thành màu đen khi cần và kiểm tra lỗi. Phần quan trọng nhất của giải thuật thêm phần tử vào cây đỏ đen là nắm giữ được chỉ số trạng thái mỗi khi có một lần đệ quy kết thúc. Chúng ta cần một kiểu liệt kê mới để phân biệt các trạng thái:

```
enum RB_code {okay, red_node, left_red, right_red, duplicate};
/*   Các giá trị trạng thái mà một lần đệ quy cần chuẩn bị trước khi kết thúc để trả về cho lần đệ
quy bên trên như sau (giả sử gọi nút đang được xử lý trong lần đệ quy hiện tại là *current):

okay:       Màu của *current không có sự thay đổi nào.

red_node:   Màu của *current vừa chuyển từ đen sang đỏ. Lần đệ quy bên trên khi nhận được
            thông tin này cần xem xét để xử lý.

right_red:  Màu của *current  và nút con phải của nó đều là đỏ, có sự vi phạm điều kiện đỏ.
            Lần đệ quy bên trên khi nhận được thông tin này cần thực hiện phép quay hoặc đổi
            màu cần thiết.

left_red:   Màu của *current  và nút con trái của nó đều là đỏ, có sự vi phạm điều kiện đỏ.
            Lần đệ quy bên trên khi nhận được thông tin này cần thực hiện phép quay hoặc đổi
            màu cần thiết.

duplicate:  Phần tử cần thêm vào cây đã có trong cây.
*/
```

```
template <class Record>
Error_code RB_tree<Record>::insert(const Record &new_entry)
```

```
/*
post: Nếu khóa trong new_entry đã có trong RB_tree, phương thức trả về
      duplicate_error. Ngược lại, phương thức trả về success và new_entry được thêm
      vào cây sao cho cây vẫn thỏa cây RB-tree.
uses: Các phương thức của struct RB_node và hàm đệ quy rb_insert.
*/
{
   RB_code status = rb_insert(root, new_entry);
   switch (status) {
      case red_node:
         root->set_color(black);
      case okay:
         return success;
      case duplicate:
         return duplicate_error;
      case right_red:
      case left_red:
         cout << "WARNING: Program error detected in RB_tree::insert" <<
              endl;
         return internal_error;
   }
}
```

Hàm đệ quy **rb_insert** thực hiện thực sự việc thêm phần tử mới vào cây: tìm kiếm trong cây theo cách thông thường, gặp cây con rỗng, thêm nút mới vào tại đây, các việc còn lại được thực hiện trên đường quay về của các lần gọi đệ quy. Hàm này có gọi **modify_left** hoặc **modify_right** để thực hiện các phép quay và đổi màu tương ứng với các trường hợp trong hình 10.17.

```
template <class Record>
RB_code RB_tree<Record>::rb_insert(Binary_node<Record> *&current,
                                   const Record &new_entry)
/*
pre:  current là NULL hoặc là địa chỉ của nút gốc của một cây con trong RB_tree.
post: Nếu khóa trong new_entry đã có trong RB_tree, phương thức trả về
      duplicate_error. Ngược lại, phương thức trả về success và new_entry được thêm
      vào cây con có gốc là current. Tính chất cây đỏ đen trong cây con này vẫn thỏa ngoại
      trừ màu tại nút gốc của nó và một trong hai nút con của nút gốc này. Trạng thái này sẽ
      được hàm modify_right hoặc modify_left điều chỉnh thích hợp (tương ứng các trị của
      RB_code) để trả về cho lần đệ quy bên trên của lần gọi rb_insert này.
uses: Các phương thức của lớp RB_node, các hàm rb_insert (một cách đệ quy),
      modify_left, và modify_right.
*/
{
   RB_code status,
           child_status;
   if (current == NULL) {
      current = new RB_node<Record>(new_entry);
      status = red_node;
   }

   else if (new_entry == current->data)
      return duplicate;
   else if (new_entry < current->data) {
```

```
      child_status = rb_insert(current->left, new_entry);
      status = modify_left(current, child_status);
   }
   else {
      child_status = rb_insert(current->right, new_entry);
      status = modify_right(current, child_status);
   }
   return status;
}
```

Hàm **modify_left** dựa vào trạng thái của các nút con để thực hiện phép quay hay sửa đổi màu tương ứng, đồng thời cập nhật lại và trả về chỉ số trạng thái cho **rb_insert**. Chính trong hàm này chúng ta quyết định trì hoãn công việc khôi phục các đặc tính của cây đỏ đen. Khi **modify_left** được gọi, chúng ta biết rằng việc thêm vào vừa được thực hiện trong cây con bên trái của nút hiện tại, biết được màu của nút hiện tại, và thông qua chỉ số trạng thái, chúng ta còn biết được trường hợp nào đã xảy ra ở cây con trái của nó. Bằng cách sử dụng các thông tin này, chúng ta có thể xác định chính xác những việc cần làm để khôi phục các đặc tính của cây đỏ đen.

```
template <class Record>
RB_code RB_tree<Record>::modify_left(Binary_node<Record> *&current,
                                     RB_code &child_status)
/*
pre:  Cây con bên trái của current vừa được thêm nút mới, trị trong child_status sẽ quyết
      định việc xử lý kế tiếp trong hàm này.
post: Việc đổi màu hoặc quay cần thiết đã được thực hiện, trạng thái thích hợp được trả về bởi
      hàm này.
uses: Các phương thức của struct RB_node, các hàm rotate_right,
      double_rotate_right, và flip_color.
*/
{
   RB_code status = okay;
   Binary_node<Record> *aunt = current->right;
   Color aunt_color = black;
   if (aunt != NULL) aunt_color = aunt->get_color();
   switch (child_status) {
   case okay:
      break;  //  Việc xử lý đã kết thúc và không cần lan truyền lên trên nữa.
   case red_node:
      if (current->get_color() == red)
         status = left_red;
      else
         status = okay;// current màu đen, nút con trái màu đỏ, đã thỏa cây
                                                                RB-tree.
      break;
   case left_red:
      if (aunt_color == black) status = rotate_right(current);
      else                     status = flip_color(current);
      break;
   case right_red:
      if (aunt_color == black) status = double_rotate_right(current);
      else                     status = flip_color(current);
```

```
      break;
   }
   return status;
}
```

Hàm phụ trợ **modify_right** cũng tương tự, nó xử lý cho các tình huống cây có dạng như những hình ảnh phản chiếu qua gương của các tình huống trong hình 10.17. Hàm đổi màu **flip_color** được xem như bài tập. Các hàm quay dựa trên cơ sở các hàm quay của cây AVL, có thêm việc gán lại các màu và chỉ số trạng thái thích hợp.

10.4.6. Loại một nút

Cũng như cây B-tree, việc loại bỏ một nút phức tạp hơn việc thêm vào, đối với cây đỏ đen, việc loại nút còn khó khăn hơn rất nhiều. Việc thêm vào tạo ra một nút mới màu đỏ dẫn đến nguy cơ vi phạm điều kiện đỏ, chúng ta cần xem xét một cách cẩn thận để giải quyết các vi phạm này. Việc loại một nút đỏ ra khỏi cây không khó lắm. Tuy nhiên, việc loại một nút đen khỏi cây dẫn đến nguy cơ vi phạm điều kiện đen, và nó đòi hỏi chúng ta phải xem xét rất nhiều trường hợp đặc biệt để khôi phục điều kiện đen cho cây.