**3.3 CÂY TÌM KIẾM CÂN BẰNG**

2-3 search trees: Bước chính để có được sự linh hoạt mà chúng ta cần để đảm bảo sự cân bằng trong cây tìm kiếm là cho phép các nút trong cây của chúng ta giữ nhiều hơn một khóa. Cụ thể, khi gọi các nút trong BST chuẩn là 2-nút (chúng giữ hai liên kết và một khóa), giờ đây chúng ta cũng cho phép 3-nút, giữ ba liên kết và hai khóa. Cả 2-nút và 3-nút đều có một liên kết cho mỗi khoảng được bao quanh bởi các khóa của nó.

Search: Thuật toán tìm kiếm khóa trong cây 2-3 tổng quát trực tiếp thuật toán tìm kiếm cho BST. Để xác định xem khóa có nằm trong cây hay không, chúng ta so sánh khóa đó với các khóa ở gốc. Nếu khóa đó bằng bất kỳ khóa nào trong số chúng, chúng ta có một kết quả tìm kiếm; nếu không, chúng ta theo liên kết từ gốc đến cây con tương ứng với khoảng giá trị khóa có thể chứa khóa tìm kiếm. Nếu liên kết đó là null, chúng ta có một kết quả tìm kiếm không thành công; nếu không, chúng ta tìm kiếm đệ quy trong cây con đó.

Insert into a 2-node: Để chèn một nút mới vào cây 2-3, chúng ta có thể thực hiện tìm kiếm không thành công rồi móc vào nút ở dưới cùng, như chúng ta đã làm với BST, nhưng cây mới sẽ không duy trì được trạng thái cân bằng hoàn hảo. Lý do chính khiến cây 2-3 hữu ích là chúng ta có thể chèn và vẫn duy trì được trạng thái cân bằng hoàn hảo. Thật dễ dàng để thực hiện nhiệm vụ này nếu nút mà tìm kiếm kết thúc là một nút 2: chúng ta chỉ cần thay thế nút bằng nút 3 chứa khóa của nút đó và khóa mới cần chèn. Nếu nút mà tìm kiếm kết thúc là nút 3,chúng ta phải làm nhiều việc hơn.

Insert into a tree consisting of a single 3-node: Để khởi động trước khi xem xét trường hợp chung, giả sử chúng ta muốn chèn vào một cây 2-3 nhỏ chỉ gồm một nút 3 duy nhất. Một cây như vậy có hai khóa, nhưng không có chỗ cho một khóa mới trong một nút của nó. Để có thể thực hiện việc chèn, chúng ta tạm thời đặt khóa mới vào một nút 4, một phần mở rộng tự nhiên của loại nút của chúng ta có ba khóa và bốn liên kết. Việc tạo 4 nút tiện lợi vì dễ dàng chuyển đổi nó thành cây 2-3 được tạo thành từ ba nút 2, một nút có khóa ở giữa (ở gốc), một nút có khóa nhỏ nhất trong ba khóa (được trỏ đến bởi liên kết bên trái của gốc) và một nút có khóa lớn nhất trong ba khóa (được trỏ đến bởi liên kết bên phải của gốc).

Insert into a 3-node whose parent is a 2-node: Để khởi động lần thứ hai, giả sử rằng tìm kiếm kết thúc tại một nút 3 ở dưới cùng có nút cha là nút 2. Trong trường hợp này, chúng ta vẫn có thể tạo chỗ cho khóa mới trong khi vẫn duy trì sự cân bằng hoàn hảo trong cây, bằng cách tạo một nút 4 tạm thời như vừa mô tả, sau đó tách nút 4 như vừa mô tả, nhưng sau đó, thay vì tạo một nút mới để giữ khóa giữa, hãy di chuyển khóa giữa đến nút cha của nút đó.

Insert into a 3-node whose parent is a 3-node: Bây giờ giả sử rằng tìm kiếm kết thúc tại một nút có nút cha là một nút 3. Một lần nữa, chúng ta tạo một nút 4 tạm thời như vừa mô tả, sau đó chia nó ra và chèn khóa giữa của nó vào nút cha. Nút cha là một nút 3, vì vậy chúng ta thay thế nó bằng một nút 4 mới tạm thời chứa khóa giữa từ nút chia 4. Sau đó, chúng ta thực hiện chính xác cùng một phép biến đổi trên nút đó. Nghĩa là, chúng ta chia nút 4 mới và chèn khóa giữa của nó vào nút cha của nó. Mở rộng sang trường hợp chung thì rõ ràng: chúng ta tiếp tục lên cây, chia các nút 4 và chèn khóa giữa của chúng vào nút cha của chúng cho đến khi đạt đến nút 2, mà chúng ta thay thế bằng nút 3 không cần phải chia thêm nữa, hoặc cho đến khi đạt đến nút 3 ở gốc.

Splitting the root: Nếu chúng ta có 3 nút dọc theo toàn bộ đường dẫn từ điểm chèn đến gốc, chúng ta sẽ có một nút 4 tạm thời tại gốc.Trong trường hợp này, chúng ta có thể tiến hành theo chính xác cách tương tự như khi chèn vào một cây gồm một nút 3 duy nhất.Chúng ta chia nút 4 tạm thời thành ba nút 2, tăng chiều cao của cây lên 1. Lưu ý rằng phép biến đổi cuối cùng này duy trì sự cân bằng hoàn hảo vì nó được thực hiện tại gốc.

Local transformations: Chia một nút 4 tạm thời trong một cây 2-3 liên quan đến một trong sáu phép biến đổi, được tóm tắt ở cuối trang tiếp theo. Nút 4 có thể là gốc; nó có thể là con trái hoặc con phải của một 2 nút; hoặc nó có thể là con trái, con giữa hoặc con phải của một nút 3. Cơ sở của thuật toán chèn cây 2-3 là tất cả các phép biến đổi này đều hoàn toàn cục bộ: không cần kiểm tra hoặc sửa đổi bất kỳ phần nào của cây ngoài các nút và liên kết đã chỉ định. Số lượng liên kết đã thay đổi cho mỗi phép biến đổi bị giới hạn bởi một hằng số nhỏ. Đặc biệt,các phép biến đổi có hiệu lực khi chúng ta tìm thấy các mẫu đã chỉ định ở bất kỳ đâu trong cây,không chỉ ở dưới cùng.

Global properties: Hơn nữa,các phép biến đổi cục bộ này bảo toàn các thuộc tính toàn cục là cây được sắp xếp và cân bằng hoàn hảo: số lượng liên kết trên đường dẫn từ gốc đến bất kỳ liên kết null nào đều giống nhau. Để tham khảo, sơ đồ hoàn chỉnh minh họa điểm này cho trường hợp 4-node là con giữa của 3-node được hiển thị ở trên. Nếu độ dài của mọi đường dẫn từ gốc đến liên kết null là h trước phép biến đổi, thì nó là h sau phép biến đổi.

Không giống như BST chuẩn, mọc từ trên xuống, 2-3 cây mọc từ dưới lên. Nếu bạn dành thời gian nghiên cứu kỹ hình ở trang tiếp theo, cung cấp trình tự 2-3 cây do máy khách kiểm tra lập chỉ mục chuẩn của chúng tôi tạo ra và trình tự 2-3 cây được tạo ra khi cùng một khóa được chèn theo thứ tự tăng dần, bạn sẽ hiểu rõ cách 2-3 cây được xây dựng. Hãy nhớ rằng trong BST, trình tự tăng dần cho 10 khóa dẫn đến cây xấu nhất có chiều cao là 9.Trong 2-3 cây, chiều cao là 2.

Mô tả trước đó đủ để định nghĩa một triển khai bảng ký hiệu với 2-3 cây làm cấu trúc dữ liệu cơ bản. Phân tích 2-3 cây khác với phân tích BST vì mối quan tâm chính của chúng tôi là hiệu suất trường hợp xấu nhất, trái ngược với hiệu suất trường hợp trung bình (nơi chúng tôi phân tích hiệu suất dự kiến ​​theo mô hình khóa ngẫu nhiên). Trong các triển khai bảng ký hiệu, chúng tôi thường không kiểm soát được thứ tự mà khách hàng chèn khóa vào bảng và phân tích trường hợp xấu nhất là một cách để cung cấp đảm bảo hiệu suất.

Red-black BSTs: Thuật toán chèn cho 2-3 cây vừa mô tả không khó hiểu; giờ đây, chúng ta sẽ thấy rằng nó cũng không khó để triển khai. Chúng ta sẽ xem xét một biểu diễn đơn giản được gọi là BST đỏ-đen dẫn đến một triển khai tự nhiên. Cuối cùng, không cần nhiều mã, nhưng để hiểu cách thức và lý do tại sao mã hoàn thành công việc đòi hỏi phải xem xét cẩn thận.

Encoding 3-nodes: Ý tưởng cơ bản đằng sau BST đỏ-đen là mã hóa 2-3 cây bằng cách bắt đầu với BST chuẩn (được tạo thành từ 2 nút) và thêm thông tin bổ sung để mã hóa 3 nút. Chúng tôi coi các liên kết là có hai loại khác nhau: liên kết đỏ, liên kết với nhau hai nút 2 để biểu diễn 3 nút và liên kết đen, liên kết cây 2-3. Cụ thể, chúng tôi biểu diễn 3 nút là hai nút 2 được kết nối bằng một liên kết đỏ duy nhất nghiêng về bên trái (một trong 2 nút là con bên trái của nút kia).

An equivalent definition: Một cách khác để tiến hành là định nghĩa BST đỏ-đen là BST

có liên kết đỏ và đen và đáp ứng ba hạn chế sau:

■ Liên kết đỏ nghiêng về bên trái.

■ Không có nút nào có hai liên kết đỏ được kết nối với nó.

■ Cây có sự cân bằng đen hoàn hảo: mọi đường dẫn từ gốc đến liên kết null đều có

cùng số liên kết đen.

Có sự tương ứng 1-1 giữa BST đỏ-đen được định nghĩa theo cách này và 2-3 cây.

Color representation: Để thuận tiện, vì mỗi ​​nút được trỏ đến chính xác bởi một liên kết (từ nút cha của nó), chúng tôi mã hóa màu của các liên kết trong các nút, bằng cách thêm một biến thể hiện boolean color vào kiểu dữ liệu Node của chúng tôi, biến này là true nếu liên kết từ nút cha là màu đỏ và false nếu tit là màu đen. Theo quy ước, các liên kết null là màu đen. Để rõ ràng hơn trong mã của chúng tôi, chúng tôi định nghĩa các hằng số RED và BLACK để sử dụng trong việc thiết lập và kiểm tra biến này. Chúng tôi sử dụng phương thức riêng isRed() để kiểm tra màu của liên kết của một nút đến nút cha của nó. Khi chúng tôi tham chiếu đến màu của một nút, chúng tôi đang tham chiếu đến màu của liên kết trỏ đến nút đó, và ngược lại.

Rotations: Việc triển khai mà chúng ta sẽ xem xét có thể cho phép các liên kết màu đỏ nghiêng phải hoặc hai liên kết màu đỏ liên tiếp trong một hoạt động, nhưng nó luôn sửa các điều kiện này trước khi hoàn tất, thông qua việc sử dụng hợp lý một hoạt động gọi là xoay để chuyển hướng của các liên kết màu đỏ. Trước tiên, giả sử rằng chúng ta có một liên kết màu đỏ nghiêng phải cần được xoay để nghiêng sang trái (xem sơ đồ bên trái). Hoạt động này được gọi là xoay trái.

Resetting the link in the parent after a rotation: Cho dù là trái hay phải, mỗi phép quay đều để lại cho chúng ta một liên kết. Chúng ta luôn sử dụng liên kết được trả về bởi rotateRight() hoặc rotateLeft() để đặt lại liên kết thích hợp trong phần tử cha (hoặc gốc của cây). Đó có thể là liên kết phải hoặc trái, nhưng chúng ta luôn có thể sử dụng nó để đặt lại liên kết trong phần tử cha. Liên kết này có thể đỏ hoặc đen—cả rotateLeft() và rotateRight() đều giữ nguyên màu của nó bằng cách đặt x.color thành h.color. Điều này có thể cho phép hai liên kết màu đỏ liên tiếp xảy ra trong cây, nhưng các thuật toán của chúng ta cũng sẽ sử dụng phép quay để sửa tình trạng này khi nó phát sinh. Ví dụ, mã

h = rotateLeft(h);

xoay trái một liên kết màu đỏ nghiêng phải nằm bên phải nút h, thiết lập h để trỏ đến gốc của cây con kết quả (chứa tất cả các nút giống như cây con được h trỏ đến trước khi xoay, nhưng gốc khác). Tính dễ viết loại mã này là lý do chính chúng tôi sử dụng các triển khai đệ quy của phương pháp BST, vì nó làm cho việc xoay trở thành một phần bổ sung dễ dàng cho việc chèn bình thường, như bạn sẽ thấy.

Insert into a single 2-node: BST đỏ-đen với 1 khóa chỉ là một 2 nút duy nhất. Việc chèn khóa thứ hai ngay lập tức cho thấy nhu cầu phải có một phép toán xoay. Nếu khóa mới nhỏ hơn khóa trong cây, chúng ta chỉ cần tạo một nút (màu đỏ) mới với khóa mới và chúng ta đã hoàn thành: chúng ta có một BST đỏ-đen tương đương với một 3 nút duy nhất. Nhưng nếu khóa mới lớn hơn khóa trong cây, thì việc gắn một nút (màu đỏ) mới sẽ tạo ra một liên kết màu đỏ nghiêng về bên phải và mã root = rotateLeft(root); hoàn tất việc chèn bằng cách xoay liên kết màu đỏ sang trái và cập nhật liên kết gốc cây. Kết quả trong cả hai trường hợp là biểu diễn màu đỏ-đen của một 3 nút duy nhất, với hai khóa, một liên kết màu đỏ nghiêng về bên trái và chiều cao màu đen là 1.

Insert into a 2-node at the bottom: Chúng ta chèn các khóa vào BST đỏ-đen như thường lệ vào BST, thêm một nút mới ở phía dưới (tuân theo thứ tự), nhưng luôn kết nối với nút cha của nó bằng một liên kết màu đỏ. Nếu nút cha là 2 nút, thì hai trường hợp giống nhau vừa thảo luận là có hiệu lực. Nếu nút mới được gắn vào liên kết bên trái, nút cha chỉ đơn giản trở thành nút 3; nếu nó được gắn vào liên kết bên phải, chúng ta có nút 3 nghiêng theo hướng sai, nhưng một phép quay trái hoàn thành công việc.

Insert into a tree with two keys (in a 3-node): Trường hợp này giảm xuống còn ba trường hợp con: khóa mới nhỏ hơn cả hai khóa trong cây, giữa chúng, hoặc lớn hơn cả hai. Mỗi trường hợp giới thiệu một nút có hai liên kết màu đỏ được kết nối với nó; mục tiêu của chúng ta là sửa tình trạng này.

\*Trường hợp đơn giản nhất trong ba trường hợp là khi khóa mới lớn hơn

hai khóa trong cây và do đó được gắn vào liên kết ngoài cùng bên phải của

3 nút, tạo thành một cây cân bằng với khóa ở giữa tại gốc,

được kết nối bằng các liên kết màu đỏ đến các nút chứa một khóa nhỏ hơn và một

khóa lớn hơn. Nếu chúng ta lật màu của hai liên kết đó từ đỏ sang đen, thì chúng ta

có một cây cân bằng có chiều cao 2 với ba nút, chính xác là những gì chúng ta

cần để duy trì sự tương ứng 1-1 của chúng ta với các cây 2-3. Hai

trường hợp khác cuối cùng sẽ giảm xuống trường hợp này.

\*Nếu khóa mới nhỏ hơn hai khóa trong cây và nằm trên liên kết bên trái, thì chúng ta có hai

liên kết màu đỏ liên tiếp, cả hai đều nghiêng về bên trái, mà chúng ta có thể giảm xuống trường hợp trước đó (khóa giữa ở gốc, được kết nối với các khóa khác bằng hai liên kết màu đỏ) bằng cách xoay liên kết trên cùng sang bên phải.

\*Nếu khóa mới nằm giữa hai khóa trong cây, chúng ta lại có hai liên kết màu đỏ liên tiếp, một liên kết nghiêng phải bên dưới một liên kết nghiêng trái, chúng ta có thể giảm xuống trường hợp trước (hai liên kết màu đỏ liên tiếp, sang trái) bằng cách xoay trái liên kết dưới cùng.

Tóm lại, chúng ta đạt được kết quả mong muốn bằng cách thực hiện không,

một hoặc hai lần xoay tiếp theo là lật màu của

hai con của gốc. Cũng như với 2-3 cây, hãy chắc chắn rằng

bạn hiểu các phép biến đổi này, vì chúng là chìa khóa cho

động lực học của cây đỏ-đen.

Flipping colors: Để đảo ngược màu của hai con màu đỏ của một nút, chúng ta sử dụng phương thức flipColors(), được hiển thị ở bên trái. Ngoài việc đảo ngược màu của các con từ đỏ sang đen, chúng ta cũng đảo ngược màu của cha mẹ từ đen sang đỏ.Một đặc điểm cực kỳ quan trọng của hoạt động này là, giống như phép quay, đây là phép biến đổi cục bộ giúp duy trì sự cân bằng màu đen hoàn hảo trong cây. Hơn nữa, quy ước này ngay lập tức dẫn chúng ta đến một triển khai đầy đủ, như chúng tôi mô tả tiếp theo.

Keeping the root black: Trong trường hợp vừa xem xét (chèn vào một nút 3 đơn), lật màu sẽ tô màu gốc thành màu đỏ. Điều này cũng có thể xảy ra ở những cây lớn hơn. Nói một cách chính xác, một gốc màu đỏ ngụ ý rằng gốc là một phần của nút 3, nhưng không phải vậy, vì vậy chúng ta tô màu gốc đen sau mỗi lần chèn. Lưu ý rằng chiều cao màu đen của cây tăng lên 1 bất cứ khi nào màu của màu của gốc được lật từ đen sang đỏ.

Implementation: Vì các hoạt động cân bằng được thực hiện trên đường lên cây từ điểm chèn, nên việc triển khai chúng rất dễ dàng trong triển khai đệ quy tiêu chuẩn của chúng tôi: chúng tôi chỉ thực hiện chúng sau các lệnh gọi đệ quy, như được hiển thị trong Thuật toán 3.4. Ba hoạt động được liệt kê trong đoạn trước, mỗi hoạt động có thể được thực hiện bằng một câu lệnh if duy nhất để kiểm tra màu của hai nút trong cây. Mặc dù nó liên quan đến một lượng nhỏ mã, nhưng việc triển khai này sẽ khá khó hiểu nếu không có hai lớp trừu tượng mà chúng tôi đã phát triển (2-3 cây và BST đỏ-đen) để triển khai nó. Với chi phí kiểm tra ba đến năm màu nút (và có thể thực hiện một hoặc hai phép xoay hoặc lật màu khi thử nghiệm thành công), chúng tôi có được các BST có sự cân bằng gần như hoàn hảo.

Deletion: Vì put() trong Thuật toán 3.4 đã là một trong những phương pháp phức tạp nhất mà chúng tôi xem xét trong cuốn sách này, và việc triển khai deleteMin(), deleteMax() và delete() cho BST đỏ-đen phức tạp hơn một chút, nên chúng tôi hoãn việc triển khai đầy đủ của chúng cho các bài tập. Tuy nhiên, cách tiếp cận cơ bản vẫn đáng để nghiên cứu. Để mô tả, chúng ta bắt đầu bằng cách quay lại 2-3 cây. Cũng như với việc chèn, chúng ta có thể xác định một chuỗi các phép biến đổi cục bộ cho phép chúng ta xóa một nút trong khi vẫn duy trì sự cân bằng hoàn hảo. Quá trình này phức tạp hơn một chút so với việc chèn, vì chúng ta thực hiện các phép biến đổi cả trên đường xuống đường tìm kiếm,khi chúng ta giới thiệu 4 nút tạm thời (để cho phép xóa một nút), và cũng trên đường lên đường tìm kiếm, nơi chúng ta chia bất kỳ 4 nút còn lại nào (theo cùng cách như khi chèn).

Delete the minimum: Để khởi động lần thứ hai cho việc xóa, chúng ta xem xét hoạt động

xóa giá trị tối thiểu khỏi cây 2-3. Ý tưởng cơ bản dựa trên quan sát rằng chúng ta có thể dễ dàng xóa khóa khỏi 3 nút ở dưới cùng của cây, nhưng không phải khỏi 2 nút. Xóa khóa khỏi 2 nút sẽ khiến nút không còn khóa; điều tự nhiên cần làm là thay thế nút bằng liên kết null, nhưng hoạt động đó sẽ vi phạm điều kiện cân bằng hoàn hảo. Vì vậy, chúng ta áp dụng cách tiếp cận sau: để đảm bảo rằng chúng ta không kết thúc ở một nút 2, chúng ta thực hiện các phép biến đổi thích hợp trên đường xuống cây để bảo toàn bất biến rằng nút hiện tại không phải là 2 nút (có thể là nút 3 hoặc nút 4 tạm thời).

Delete: Các phép biến đổi tương tự dọc theo đường dẫn tìm kiếm vừa được mô tả để xóa minimum có hiệu quả để đảm bảo rằng nút hiện tại không phải là nút 2 trong quá trình tìm kiếm cho bất kỳ khóa nào. Nếu khóa tìm kiếm ở dưới cùng, chúng ta có thể chỉ cần xóa nó. Nếu khóa không ở dưới cùng, thì chúng ta phải trao đổi nó với nút kế nhiệm của nó như trong BST thông thường. Sau đó, vì nút hiện tại không phải là nút 2, chúng ta đã giảm vấn đề xuống còn xóa minimum trong một cây con có gốc không phải là nút 2 và chúng ta có thể sử dụng quy trình vừa được mô tả cho cây con đó. Sau khi xóa, như thường lệ, chúng ta chia bất kỳ nút 4 nào còn lại trên đường dẫn tìm kiếm trên đường lên cây.

Properties of red-black BSTs: Nghiên cứu các đặc tính của BST đỏ-đen là vấn đề kiểm tra sự tương ứng với 2-3 cây và sau đó áp dụng phân tích của 2-3 cây. Kết quả cuối cùng là tất cả các hoạt động bảng ký hiệu trong BST đỏ-đen đều được đảm bảo là logarit theo kích thước của cây (trừ tìm kiếm phạm vi, ngoài ra tốn thời gian tỷ lệ thuận với số lượng khóa được trả về). Chúng tôi lặp lại và nhấn mạnh điểm này vì tầm quan trọng của nó.

Ordered symbol-table API: Một trong những tính năng hấp dẫn nhất của BST đỏ-đen là mã phức tạp bị giới hạn ở các phương thức put() (và xóa). Mã của chúng tôi cho các truy vấn tối thiểu/tối đa, chọn, xếp hạng, sàn, trần và phạm vi trong BST chuẩn có thể được sử dụng mà không cần bất kỳ thay đổi nào, vì nó hoạt động trên BST và không cần tham chiếu đến màu nút. Thuật toán 3.4, cùng với các phương thức này (và các phương thức xóa), dẫn đến việc triển khai hoàn chỉnh API bảng ký hiệu có thứ tự của chúng tôi. Hơn nữa, tất cả các phương thức đều được hưởng lợi từ sự cân bằng gần như hoàn hảo trong cây vì tất cả chúng đều yêu cầu thời gian tỷ lệ thuận với chiều cao của cây, nhiều nhất là như vậy. Do đó, Đề xuất G, kết hợp với Đề xuất E, đủ để thiết lập một đảm bảo hiệu suất logarit cho tất cả chúng.