Giải Thuật Lập Trình

Nơi tổng hợp và chia sẻ những kiến thức liên quan tới giải thuật nói chung và lý thuyết khoa học máy tính nói riêng.

Red-Black tree

You are currently browsing articles tagged **Red-Black tree**.

Cây nhị phân cân bằng --- Balanced binary search tree

March 21, 2017 in Uncategorized | No comments

Xét bài toán sau:

> **Problem 1:** Giả sử ta có một tập hợp n số nguyên $S = \{k_1, k_2, \ldots, k_n\}$, tìm cách biểu diễn S bằng một cấu trúc dữ liệu sao cho 3 thao tác sau được thực hiện hiệu quả:
>
> 1. Access(S, k): Kiểm tra xem S có chứa số nguyên k hay không.
> 2. Delete(S,k): Xóa số k ra khỏi tập S.
> 3. Insert(S,k): Chèn thêm một số nguyên k vào tập S

Có 2 cách mà ta nghĩ đến ngay: (i) dùng mảng để biểu diễn S hoặc (ii) dùng danh sách liên kết (http://www.giaithuatlaptrinh.com/?p=1326) . Cách (ii) có lẽ không phù hợp vì thao tác Access(S, k) mất thời gian $O(n)$ nếu như S có n phần tử. Với cách (i), ta có thể giải quyết thao tác Access(S, k) bằng cách duy trì một mảng đã sắp xếp và sử dụng tìm kiếm nhị phân (http://www.giaithuatlaptrinh.com/?p=48) . Tuy nhiên, ta lại gặp vấn đề với hai thao tác sau.

Trong bài này, ta sẽ tìm hiểu cấu trúc cây nhị phân tìm kiếm cân bằng (blanched binary search tree) để biểu diễn S. Với cấu trúc này, ta có thể thực hiện mỗi thao tác trên trong thời gian $O(\log n)$.

Cây nhị phân tìm kiếm

Cây có thể coi là một đồ thị đơn không có chu trình. Ta thường chỉ định một nút đặc biệt trong cây, kí hiệu r, mà ta gọi là **gốc** của cây. Các nút có bậc 1 (chỉ kề với một cạnh) trong cây, ngoại trừ nút gốc, được gọi là các nút lá (leaf node). Nút không phải là nút lá ta gọi là **nút trong** của cây (internal node). Với mỗi cạnh uv nối hai nút u và v, nút nào gần gốc hơn thì ta gọi nút đó là **nút cha** của nút còn lại. Ta dùng $p[v]$ để kí hiệu

nút cha của một nút v. CHú ý, gốc r không có nút cha và ta gán $P[v] \leftarrow$ NULL. Ta gọi v là nút con của nút $P[v]$.

Một cây được gọi là **nhị phân** nếu với mỗi nút trong có **tối đa** hai nút con. Thông thường ta biểu diễn một cây nhị phân như Figure 1(a). Cây trong khoa học máy tính thường được biểu diễn ngược với cây trong tự nhiên; gốc ở trên còn lá ở dưới. Đôi khi, ta định hướng cạnh từ nút cha tới nút con (hoặc ngược lại) để quan hệ cha-con trực quan hơn. Từ đây về cuối bài, ta sử dụng khái niệm cây để ám chỉ cây nhị phân, vì ta chỉ tìm hiểu cây nhị phân trong bài này.

Ta sử dụng cây để tổ chức dữ liệu bằng cách chèn các phần tử dữ liệu vào mỗi nút của cây và mỗi nút của cây được gán một khóa. Khóa không nhất thiết phải khác nhau, tuy nhiên, để trình bày đơn giản, ta sẽ giả sử khóa của các nút đôi một khác nhau. Để hỗ trợ tìm kiếm, ta thường tổ chức dữ liệu trên cây sao cho khóa của của nút cha lớn hơn khóa của nút con trái và nhỏ hơn khóa của nút con phải. Ta gọi tính chất này là tính chất **tìm kiếm** và gọi cây nhị phân như vậy là **cây nhị phân tìm kiếm**. Figure 1(b) minh họa một cây nhị phân tìm kiếm.

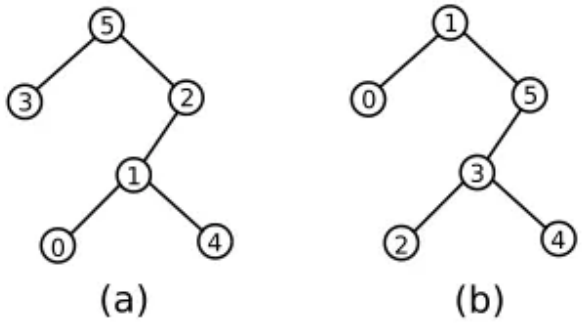


Figure 1: (a) một cây nhị phân và (b) một cây nhị phân tìm kiếm

Một số kí hiệu: Nhắc lại, ta sẽ dùng $P[v]$ để kí hiệu nút cha của v trong cây T. Ta dùng $L[v]$ và $R[v]$ lần lượt để kí hiệu nút con trái và nút con phải của nút v. Các giá trị này sẽ là NULL nếu v không có con trái (hoặc con phải). Ta sử dụng $k[v]$ để kí hiệu khóa của nút v.

Remark: Khi lập trình, ta có thể lưu con trỏ $P[v]$ cho mỗi nút v. Tuy nhiên, đa phần các cấu trúc cây nhị phân không cần đến trường này mà ta chỉ lưu con trỏ của 2 nút con $L[v]$ và $R[v]$. Việc tiết kiệm con trỏ $P[v]$ không chỉ vấn đề bộ nhớ, mà còn tiết kiệm được cả thời gian (cập nhật con trỏ này). Trong bài này, mình sẽ không sử dụng con trỏ $P[v]$ trong giả mã mà chỉ dùng nó như một kí hiệu tới nút cha (trong chứng minh).

Giả sử ta đã có một cây nhị phân tìm kiếm T để biểu diễn S, với khóa của mỗi nút là một số nguyên trong S. Sử dụng tính chất tìm kiếm (quan hệ giữa khóa của cha và hai nút con), Ta có thể thực hiện tìm kiếm như sau:

```
ACCESS(node r, key k):
    if r = NULL
        return FALSE
    if k[r] < k
        return ACCESS(R[r], k)
    if k[r] > k
        return ACCESS(L[r], k)
    return TRUE
```

Tạm thời gác lại hai thao tác chèn và xóa. Liệu thao tác Access(S, k) với biểu diễn cây nhị phân tìm kiếm như trên có hiệu quả? Figure 2 cho ta câu trả lời là không. Trong trường hợp xấu nhất, ta phải trả thời gian $O(n)$ để tìm kiếm một nút lá của cây nếu như cấu trúc cây trong giống một đường đi đơn (path) như Figure 2.

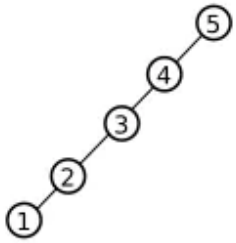


Figure 2: Một cây nhị phân tìm kiếm với độ sâu 4. Ta có thể mở rộng ra cây nhị phân tìm kiếm có độ sâu $n-1$.

Do đó, ta cần một cách tổ chức cây sao cho đường đi từ gốc của cây đến một nút lá bất kì đều không quá dài, i.e, $O(\log n)$. Từ đó ta có khái niệm cây cân bằng, là cây có độ sâu (depth) không quá lớn. Ta định nghĩa **độ sâu** của một nút u, kí hiệu $depth(u)$, là độ dài là độ dài (số cạnh) trên đường đi (duy nhất) từ gốc r đến nút u. **Độ sâu** của cây, kí hiệu $depth(T)$, được định nghĩa là độ sâu lớn nhất trong số các nút lá của cây.

$$depth(T) = \max_{v \text{ is a leaf of } T} depth(v) \qquad (1)$$

Remark: Theo định nghĩa, độ sâu của nút gốc r là $depth(r) = 0$.

Một khái niệm tương tự độ sâu là **chiều cao** (height). Chiều cao của mọi nút lá là 0 và chiều cao của một nút trong v được định nghĩa từ nút con của nó như sau:

$$height(v) = \max_{u \text{ is a child of } v} height(u) + 1 \qquad (2)$$

Chiều cao của cây, kí hiệu $height(T)$, được định nghĩa là chiều cao của nút gốc. Từ định nghĩa của chiều cao và độ sâu, ta suy ra:

$$depth(T) = height(T) \qquad (3)$$

Remark: Bạn đọc cần phân biệt rõ hai khái niệm chiều cao và độ sâu. Độ sâu được định nghĩa từ gốc xuống lá còn chiều cao được định nghĩa từ lá lên gốc.

Cây nhị phân tìm kiếm cân bằng

> **Quy ước**: Để dễ dàng trong việc phân tích cấu trúc cây, ta sẽ giả mọi nút có chứa dữ liệu (kể cả lá) có **đúng** hai nút con. Điều này có thể được đảm bảo bằng cách thêm 2 nút NULL vào mỗi nút lá, và thêm 1 nút NULL vào mỗi nút trong nếu nút trong đó có đúng 1 nút con. Xem minh họa trong Figure 3(a).

Theo quy ước trên, mọi nút lá của cây là NULL.

Có nhiều cách để định nghĩa cây cân bằng: cân bằng theo chiều cao, cân bằng theo trọng số, cân bằng theo hạng (rank), nhưng mục tiêu chung là

đảm bảo độ sâu của cây là $O(\log n)$. Mỗi cách định nghĩa cân bằng thường tương ứng với một cách thực thi riêng. Tìm một cách định nghĩa cân bằng để cân bằng giữa các yếu tố: độ sâu, bộ nhớ và tính đơn giản trong thực thi, vẫn đang là một chủ đề nghiên cứu hiện nay. Dưới đây mình giới thiệu loại cây cân bằng phổ biến: cây AVL và cây đỏ-đen (red-black tree).

Cây AVL

Cây AVL (https://en.wikipedia.org/wiki/AVL_tree) được phát triển bởi Adelson-Velsky và Landis [2], sử dụng chiều cao (height) làm tiêu chuẩn cân bằng.

Cây AVL (https://en.wikipedia.org/wiki/AVL_tree) : Mọi nút trong v trong một cây AVL đều thỏa mãn:

$$|height(L[v]) - height(R[v])| \leq 1 \qquad (4)$$

Trong bài sau ta sẽ tìm hiểu cách thức thực thi để đảm bảo tính chấ ttrong phương trình (4). Một số tính chất của cây AVL ta sẽ nghiên cứu ở đây. Đầu tiên là chiều cao của cây AVL. Theo quy ước, mọi nút lá là NULL (và có chiều cao 0). Do đó, mọi nút trong đều có chiều cao dương.

Theorem 1: Gọi T là một cây AVL với n nút trong, ta có:

$$\log_2(n+1) \leq height(T) \leq c\log_2(n+2) + b \qquad (4)$$

Với $c = 1/(\log_2(\varphi)) \approx 1.44$, $b = \frac{c}{2}\log_2 5 - 2 \approx -0.328$ và $\varphi = (1+\sqrt{5})/2 \approx 1.618$.

Chứng minh: Bằng quy nạp (chi tiế tcoi như bài tập cho bạn đọc), ta có thể chứng minh được một cây AVL chiều cao h có không quá $2^h - 1$ nút trong. Từ đó ta suy ra cận dưới.

Để chứng minh cận trên, ta cần phải xét xem một cây AVL chiều cao h có **ít nhất** bao nhiêu nút trong. Phương pháp chứng minh dưới đây được đề xuất bởi Knuth [3] (trang 453).

Gọi T_h là cây AVL chiều cao h có số nút trong ít nhất trong số các cây AVL chiều cao h. Gọi $N(h)$ là số nút của $T(h)$. Theo (4), ta có thể giả sử cây con trái của gốc r có chiều cao $h-1$ và cây con phải của gốc r có chiều cao $h-2$ (tại sao ta có thể giả sử như vậy?). Từ đó ta có công thức truy hồi:

$$N(h) = N(n-1) + N(h-2) + 1 \qquad (5)$$

và $N(0) = 0, N(1) = 1$. Nếu đặt $G(h+1) = G(h) + 1$, ta có $G(h) = G(h-1) + G(h-2)$ và $G(0) = 1, G(1) = 2$. Đây chính là dãy số Fibonacci (https://en.wikipedia.org/wiki/Fibonacci_number) , kí hiệu $F(h)$, dịch sang trái 1 đơn vị. Do đó:

$$n \geq N(h) = F(h+2) - 1 \qquad (6)$$

Bằng một vài biến đổi đại số nhỏ, ta sẽ có dpcm.

Trong thực thi cây AVL, chúng ta cần một trường (một số nguyên) để lưu trữ chiều cao của mỗi nút (khoảng 8 bít là đủ). Tùy vào ứng dụng mà số lượng bít này có thể coi là nhiều hay ít. Trong cây đỏ-đen dưới đây, mỗi nút chỉ cần đúng 1 bít để duy trì thông tin cân bằng. Do đó, so với cây AVL, cây đỏ-đen tiết kiệm được nhiều bộ nhớ lưu trữ hơn.

Cây đỏ-đen

Cây đỏ-đen được Guibas và Sedgewick [4] giới thiệu năm 1978. Trước hết ta định nghĩa cây đỏ-đen thông qua hạng (rank). Cách định nghĩa này, tuy chả liên quan gì đến cái tên đỏ-đen, nhưng cho phép chứng minh một số tính chất về chiều cao trở nên đơn giản hơn (sử dụng quy nạp là đủ). Sau đó ta sẽ liên hệ rank và màu sắc của nút.

Một cây nhị phân tìm kiếm được gọi là cây đỏ-đen nếu tồn tại một hàm $rank : V(T) \to \mathbb{R}^+$ gán cho mỗi nút của cây một số nguyên không âm sao cho với mọi nút v:

1. Nếu v không phải là nút gốc thì $rank(v) \leq rank(P[v]) \leq rank(v) + 1$.
2. Nếu nút cha của v không phải là nút gốc thì $rank(v) < rank(P^2[v])$. Ở đây $P^2[v] = P[P[v]]$ là nút ông bà (grandparent) của v.
3. Nếu v là nút lá (v = Null) thì $rank(v) = 0$ và $rank(P[v]) = 1$.

Ví dụ một cách gán $rank$ cho mỗi nút của cây ở Figure 1(b) được minh họa trong Figure 3(b)

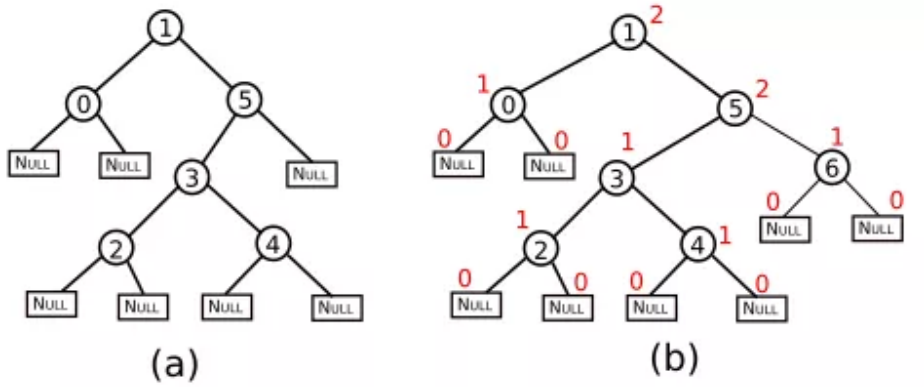


Figure 3: (a) Cây nhị phân thu được sau khi thêm các nút Null vào cây nhị phân trong Figure 1(b) để thu được cây theo quy ước. (b) Cây nhị phân với hàm $rank$. Số màu đỏ của mỗi nút tương ứng với rank của nút đó.

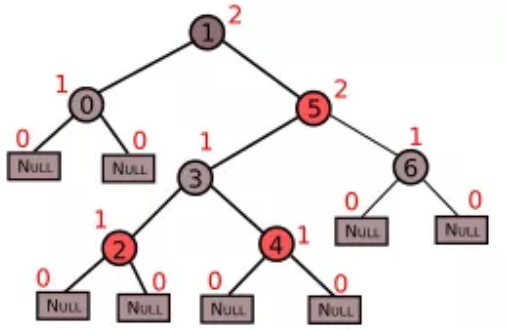
Bài tập: Chứng minh rằng không tồn tại hàm $rank(.)$ với 3 tính chất kể trên cho cây trong Figure 3(a).

Chú ý, theo quy ước, các nút lá Null sẽ có rank 0. Việc kiểm tra hàm $rank(.)$ gán như trên thỏa mãn 3 tính chất coi như bài tập cho bạn đọc.

Cây đỏ-đen (https://en.wikipedia.org/wiki/Red%E2%80%93black_tree) (red-black tree): Cây đỏ đen là cây nhị phân tìm kiếm với hàm rank thỏa mãn 3 tính chất liệt kê ở trên. Tuy nhiên, thay vì mô tả hàm rank, ta mô tả thông qua màu sắc mà ta gán cho các nút. Các nút v có $rank(P[v]) = rank(v) + 1$ được gọi là các nút đen (black) và các nút có $rank(P[v]) = rank(v)$ được gọi là các nút đỏ (red).

Từ tính chất của rank, ta suy ra các tính chất (tương đương) sau của màu sắc (xem minh họa trong Figure 4.):

1. Các nút lá (nút NULL) sẽ là các nút đen. Tính chất này được suy ra từ tính chất (3) của rank.
2. Mọi nút trong cây phả là đen hoặc đỏ. Tính chất này được suy ra từ tính chất (1) của rank.
3. Nút cha (không phải gốc) của một nút đỏ phải là một nút đen. Tính chất này được suy ra từ tính chất (2) của rank.
4. Nút gốc là một nút đen. Tính chất này chỉ là một quy ước để đảm bảo nhất quán.



Figue 4: Một cây đỏ đen được chuyển đổi từ cây nhị phân với hàm rank trong Figure 3b. Số màu đỏ trên mỗi nút là rank của nút đó.

Ngược lại, từ một cây đỏ-đen với màu sắc của mỗi nút, ta có thể suy ngược lại hàm rank bằng cách gán cho các nút lá rank 0 và rank của các nút trong khác được suy ra (bằng quy nạp) từ màu sắc và rank của các nút con của nó. Chi tiết coi như bài tập cho bạn đọc.

Tính chất màu sắc của cây đỏ đen cho phép ta chỉ dùng 1 bít để lưu trữ màu sắc tại mỗi nút, thay vì một số nguyên để lưu trữ giá trị height như trong cây AVL tree. Cách lưu trữ này tiết kiệm được rất nhiều bộ nhớ. Cấu trúc TreeMap (https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html) trong Java chính là sử dụng cây đỏ-đen.

Theorem 2: Nếu $rank(v) = k$ thì $height(v) \leq 2k$ và v có ít nhất 2^k nút con cháu (descendants).

Chứng minh: Ta chứng minh bằng phương pháp quy nạp. Dễ thấy Theorem 1 đúng cho các nút lá (có rank 0 và height 0). Với nút v có $rank(v) = k + 1$, theo (2), nút con u của v có $rank(u) = k$ hoặc $rank(u) = k - 1$. Phần còn lại của chứng minh ta sẽ xét các trường hợp, và chi tiết coi như bài tập cho bạn đọc.

Áp dụng Theorem 2 cho nút gốc r, ta suy ra nút gốc có chiều cao không quá $2 * (rank(r))$ và ít nhất $2^{rank(r)}$ nút con cháu. Do cây có tối đa n nút trong, nó có không quá $2n$ nút (tính cả các nút lá), ta suy ra:

Corollary 1: Một cây nhị phân tìm kiếm cân bằng có độ sâu không quá $2(\log n + 1)$.

Remark: Chứng minh chặt chẽ hơn ta có thể suy ra độ cao của cây là không quá $2\lfloor \log(n+1) \rfloor$. Logarithm ở đây là base 2.

Remark: So với cậy AVL, chiều cao của cây đỏ-đen nói chung là lớn hơn (xem lại Theorem 1). Tuy nhiên, cây AVL sử dụng nhiều bộ nhớ hơn. Trong bài sau ta sẽ thấy, thực thi cây đỏ-đen phức tạp hơn thực thi cây AVL rất nhiều.

Tham khảo

[1] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, 1983.
[2] G. Adelson-Velsky and E. Landis. *An Algorithm for the Organization of Information*. Proceedings of the USSR Academy of Sciences (in Russian). 146: 263–266. English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962.
[3] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition*. Addison-Wesley, 1997.
[4] L. J. Guibas and R. Sedgewick. *A Dichromatic Framework for Balanced Trees*. Proceedings of the 19th Annual Symposium on Foundations of Computer Science. pp. 8–21. 1978.

Tags: AVL-tree, binary search tree, binary tree, data structure, Red-Black tree, tutorial