هشت: درخت جستجو دودویی

ساختمان داده ها و الگوریتم

مدرس: دکتر نجمه منصوری

نگارنده: سجاد هاشمیان

۱. درخت جستجو دودویی

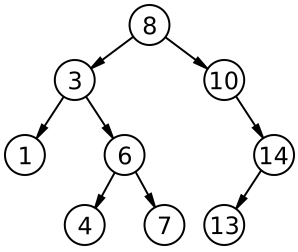
یک درخت دودویی برای ذخیره کردن داده‌هایی است که ترتیب(عملگرهای کوچکتر و بزرگتر و مساوی) برای آنها تعریف شده باشد. درخت جست و جوی دودویی این امکان را فراهم میکند که به جستوجوی یک داده، اضافه کردن آن و حذف کردن آن بپردازیم.

در واقع ایده درخت جستجو دودویی در ذخیره‌سازی گره‌هاست؛ ترتیب گره ها در درخت جست و جوی دودویی به صورتی است که در هر مقایسه نیمی از درخت باقی مانده بررسی نمیشود. بنابراین زمان جست و جوی درخت متناسب با لگاریتم تعداد عددهای ذخیره شده در درخت است.

۱.۱ تعریف

درخت جستجوی دودویی، نوعی درخت دودویی است که دارای خصوصیات زیر است:

- از تعدادی گره تشکیل شده که هر گره دارای یک کلید است. کلیدها منحصربه‌فرد هستند و در درخت کلید تکراری وجود ندارد.
- مقدار تمام کلیدهایی که در زیردرخت سمت چپ واقع شده‌اند، کوچکتر از کلید گره ریشه هستند.
- مقدار تمام کلیدهایی که در زیردرخت سمت راست واقع شده‌اند، بزرگتر از کلید گره ریشه هستند.
- زیردرخت سمت راست و زیردرخت سمت چپ خود درختان جستجوی دودویی هستند.



۱.۲ عملیات جستجو

با بررسی گره ریشه شروع می کنیم. اگر درخت تهی باشد، کلیدی که ما به دنبال آن هستیم در درخت وجود ندارد. در غیر این صورت، اگر کلید برابر با ریشه باشد، جستجو موفقیت آمیز است و گره را برمی گردانیم. اگر کلید کمتر از کلید ریشه باشد، زیر درخت سمت چپ را جستجو می کنیم. به همین ترتیب، اگر کلید بزرگتر از کلید ریشه باشد، زیردرخت سمت راست را جستجو می کنیم. این روند تکرار می شود تا زمانی که کلید پیدا شود یا زیردرخت باقی‌مانده تهی باشد. این یعنی اگر بعد از رسیدن به یک زیردرخت صفر، کلید جستجو شده پیدا نشد، آن کلید در درخت وجود ندارد.

```
def search(key, node):
    if node == None or node.key == key:
        return node
    if key < node.key:
        return search(key, node.left)
    if key > node.key:
        return search(key, node.right)
```

۱.۳ عملیات درج

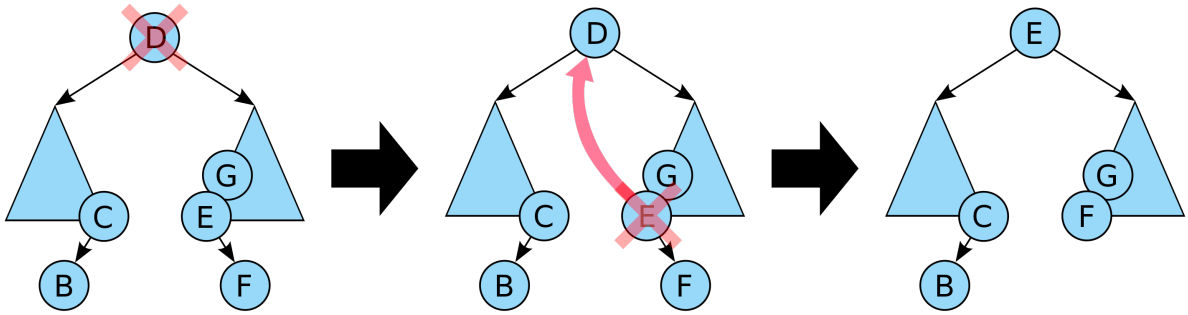
درج همانطور که جستجو شروع شد، آغاز می شود. اگر کلید برابر با ریشه نباشد، زیر درخت‌های چپ یا راست را مانند گذشته جستجو می کنیم. در نهایت، ما به یک گره خارجی خواهیم رسید و بسته به کلید این گره، کلید جدید را به عنوان فرزند راست یا چپ آن اضافه می کنیم. به عبارت دیگر، ما ریشه را بررسی می کنیم و گره جدید را اگر کلید آن کمتر از ریشه باشد در زیر شاخه سمت چپ قرار می‌دهیم، و یا اگر کلید آن بزرگتر یا برابر ریشه است در زیر درخت راست.

```
def insert(key):
    node = search(key)
    if(key == node.key):
        return
    if(key > node.key):
        node.right = key
    if(key < node.key):
        node.left = key
```

۱.۴ عملیات حذف

هنگام حذف گره از درخت جستجوی دودویی، حفظ ترتیب گره‌ها طبق تعریف درخت جستجو، امری مهم است. در واقع باتوجه به تعداد فرزندان، سه مورد احتمالی را باید در نظر گرفت:

- حذف گره بدون فرزند: به سادگی گره را از درخت جدا کنید.
- حذف گره با یک فرزند: گره را برداشته و فرزند خود را جایگزین کنید.
- حذف گره D با دو فرزند: گره قبل یا گره بعد از D را در پیمایش میان‌ترتیب انتخاب کنید. اگر E فرزند ندارد، به جای حذف D، کلید و مقدار آن را با E بازنویسی کنید، سپس E را از درخت قبلی خود حذف کنید.



با توجه به تعاریفمان از درج و حذف، شکل یک درخت جستجو دودویی کاملا وابسته به ترتیب انجام عملیات‌ها و کلید های هر عمل در هر مرحله است؛ در واقع تعریف فعلی‌مان از این عملیات ها آنچان که باید و شاید در راستا هدف ما بهینه عمل نمی‌کند، برای مثال هزنیه هر ۳ عملیات معرفی شده وابسته ارتفاع درخت است که با توجه به روش انجام این عملیات ها، بعد از تعدادی درج، درخت می‌تواند کاملا اریب شده و دیگر مزیتی نسبت به روش جستجو خطی نداشته باشد.

۲. درخت های جستجو دودویی تصادفی

همانطور که در بخشهای قبل مشاهده کردیم، مرتبه زمانی اجرای عملیات مختلف در درخت دودویی جستوجو از مرتبه $O(h)$ است. در صورتی که مطلوب ما $O(\log n)$ است. برای دستیابی به این هدف میتوانیم از ایده تصادفی کردن درخت جستوجو دودویی استفاده کنیم. هدف ما وارد کردن عناصر یک آرایه داده شده به طول n و با عناصر متمایز در یک درخت دودویی جستوجو به صورت «تصادفی» است.

۲.۱. انتخاب تصادفی یک درخت دودویی و پر کردن آن با عناصر آرایه

در این روش ابتدا از بین همه درختهای دودویی ممکن یکی را به طور کاملاً تصادفی (با توزیع یکنواخت) انتخاب میکنیم. توجه کنید که یک درخت دودویی مفروض با n راس را به طور یکتا می‌توان با مقادیر آرایه داده شده پر نمود به طوری که حاصل این عمل یک درخت جستوجو دودویی باشد. بنابراین تعداد درختهای جستوجو دودویی که متناظر با یک آرایه با n عنصر متمایز، برابر با تعداد درختهای دودویی با n گره است. همچنین تعداد درختهای دودویی با n راس برابر عدد nام کاتالان است، این یعنی:

$$h \simeq \frac{1}{n+1}\binom{2n}{n} \simeq \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

در این ر وش تصادفی، متوسط ارتفاع درخت تصادفی از مرتبه $O(\sqrt{n})$ خواهد شد و این مرتبه برای ما مطلوب نیست.

۲.۲. تشکیل درخت با درج عناصر آرایه به ترتیب تصادفی در آن

در این روش، آرایه ورودی را تصادفی خواهیم کرد. یعنی، در ابتدا از بین جایگشتهای مختلف آرایه ورودی یکی را به صورت تصادفی انتخاب می‌کنیم سپس با درج کردن متوالی عناصر آرایه تصادفی شده، درخت تصادفی مربوط به آن را میسازیم.

تعداد درخت‌های (نه لزوماً متمایز) تصادفی حاصل در این روش برابر تعداد جایگشتهای nتایی خواهد شد. با توجه به اینکه $n!$ برای $n \geq 3$ بزرگتر از عدد nام کاتالان است، توزیع حاصل یکنواخت روی همه درختهای دودویی ممکن نخواهد بود. خوشبختانه در این روش تصادفی، متوسط ارتفاع درخت تصادفی حاصل از مرتبه $O(\log n)$ خواهد شد. یعنی، اگر ترتیب درج کردن عناصر یک آرایه در یک درخت دودویی جستوجو به صورت تصادفی باشد، بهتر از این است که از ابتدا خود درخت را به صورت کاملاً تصادفی انتخاب کنیم.

دیدید که با تصادفی سازی فرآیند ساخت یک درخت جستجو دودویی می‌توان مرتبه زمانی آن را بهبود بخشید، البته اما در این روش ما مجبور به پیروی از شرایط تصادفی سازی هستیم و باید دسترسی کامل به عناصر و اجازه جابه‌جا کردن آنها را داشته باشیم کما که این اصلا چیز مطلوبی نیست و در واقع اصلا شرایط استفاده ندارد.

برای پیشگیری از این مهم، ایده دیگری وجود دارد، در واقع ما می‌توانیم با جریمه کردن عملیات های درخت جستجو برای ارتفاع آنها سعی کنیم تا برای تنظیم ارتفاع درخت پس از هر مرحله هزینه اندکی را بپردازیم، بطوری که پس از طی چند مرحله دیگر مشکل نامتناسب بود و سبک سنگین شدن زیر درخت‌ها را نداشته باشیم.

۳. درخت‌های جستجو دودویی خود متوازن کننده

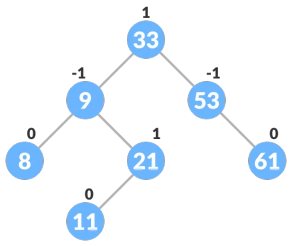
همانطور که گفتیم بیشتر عملیات‌های روی یک درخت جستجوی دودویی، زمانی مستقیما متناسب با ارتفاع درخت می‌برند. پس مطلوب است که ارتفاع را کوچک نگه داریم. یک درخت دودویی با ارتفاع h می‌تواند شامل حداکثر $2^0 + 2^1 + 2^2 + \cdots 2^h = 2^{h+1} - 1$ گره باشد. بنابراین برای درختی با n گره و ارتفاع h، $n \leq 2^{h+1} - 1$ و این یعنی برای هر درخت دودویی $h \geq \lceil \log_2(n+1) - 1 \rceil \geq \lfloor \log_2 n \rfloor$ خواهد بود.

هرچند ساده‌ترین الگوریتم برای درج عنصر در این درختان، ممکن است یک درخت با ارتفاع n را نتیجه دهد. برای مثال وقتی عناصر در ترتیب مرتب شده کلیدها درج می‌شوند، درخت به یک لیست پیوندی با n گره تبدیل می‌شود. اختلاف کارایی این دو موقعیت خیلی زیاد است. برای مثال برای $n = 10^6$ باشد حداقل ارتفاع برابر با $\lfloor \log_2(n) \rfloor = 19$ است.

ایده اصلی درختان خود متوازن در عملیات درج و حذف خواهد بود، در این روش در زمان‌های کلیدی در طی مراحل اجرای عملیات‌ها با اعمال تغییراتی روی درخت (همانند تغییر زیر درخت‌ها یا جابه‌جایی چند راس) سعی در متوازن نگه داشتن درخت دارند تا زیر درخت‌های هر راس دچار مشکل سبک-سنگینی نشوند؛ سختی این کار در کنترل هزینه سربار اعمال تغییرات است که باید به گونه ای اعمال شوند که در تعداد کمی از مراحل انجام شوند و هزینه درگیر زیادی نداشته باشند یا در غیر این صورت، هزینه محاسباتی آنها اندک و یا از زمان ثابت باشد.

۳.۲ درختان AVL

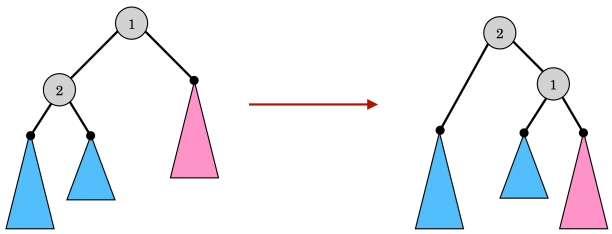
این درختان اولین ساختارداده خودمتوازن بودند که اختراع شدند. در یک درخت AVL، ایده بر تنظیم نگه داشتن حداکثر ارتفاع درخت در طی عملیات درج و حذف است، به طوری که در هر زمان، ارتفاع دو زیر درخت هر گره، حداکثر یک واحد با هم اختلاف داشته باشند، این یعنی تعداد راس‌های زیر درخت چپ و راست می‌تواند تفاوت بسیاری باهم داشته باشند اما از آنجا که اکثریت عملیات‌های یک درخت جستجو دودویی وابسته به ارتفاع درخت است، عملیات ها در بدترین حالت هم از مرتبه $O(\log n)$ زمان می‌گیرند.



به‌طور کلی برای هر راس علاوه بر اشاره‌گر ها یک مقدار BF (Balance Factor) هم نگه داری می‌شود:

$$BF(v) = h(v.right) - h(v.left).$$

تلاش بر این است که طی مراحل اجرا، برای هر گره $BF(v) \in \{1, -1, 0\}$ باشد، برای این کار نیز از عملیات چرخش برای یک گره استفاده می‌کنیم. این عملیات اینگونه تعریف می‌شود: اگر $BF(v)$ برای ما دلخواه نبود، همانند شکل با بالا آوردن ریشه زیر درخت عمیق تر، و جایگزینی آن با v و فرستادن v به زیر درخت کم عمق تر، ارتفاع درخت را اصلاح می‌کنیم.



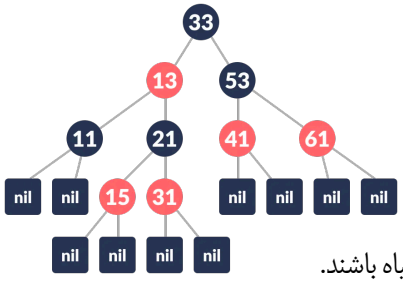
درج و حذف گره در درخت AVL

درج و حذف گره در این درختان از دو مرحله درج(حذف) گره و متوازن کردن درخت تشکیل شده است، یعنی در ابتدا همانند درختان جستجو دودویی گره‌ای را در این درخت درج(حذف) می‌کنیم، سپس اقدام به متوازن سازی درخت می‌کنیم؛ برای متوازن سازی درخت باید توجه داشت، که حذف یا درج یک گره، تنها ضریب توازن گره‌های موجود در مسیر آن گره تا ریشه را تغییر می‌دهند، این یعنی از آنجا که ارتفاع درخت تقریبا $\log n$ بوده است(خب دقیقا چقدره؟) کافیست تا برای هر کدام از آنها عملیات متعادل سازی که شرح دادیم را استفاده کرد،با توجه به مرتبه اجرایی $O(1)$ برای عملیات متوازن‌سازی یک گره، نتیجه می‌شود که حذف و درج گره در این درختان از مرتبه زمانی $O(\log n)$ خواهد بود، همانطور که شرح دادیم، این بهترین زمان ممکن برای یک درخت جستجو دودویی است، کما این که این درختان لزوما بهینه عمل نمی‌کنند.

```
def insert_node(self, root, key):
        # Find the correct location and insert the node
        if not root:
            return TreeNode(key)
        elif key < root.key:
            root.left = self.insert_node(root.left, key)
        else:
            root.right = self.insert_node(root.right, key)

        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        # Update the balance factor and balance the tree
        balanceFactor = self.getBalance(root)
        if balanceFactor > 1:
            if key < root.left.key:
                return self.rightRotate(root)
            else:
                root.left = self.leftRotate(root.left)
                return self.rightRotate(root)

        if balanceFactor < -1:
            if key > root.right.key:
                return self.leftRotate(root)
            else:
                root.right = self.rightRotate(root.right)
                return self.leftRotate(root)

        return root
```

۳.۳ درختان قرمز-سیاه

درختان قرمز سیاه، دسته مشهور دیگری از درختان خودمتوازن هستند؛ در این درختان نیز با توجه به تقریبا متوازن بودن درخت، عملیات درج و حذف از مرتبه اجرایی $O(\log n)$ خواهد بود، اما مزیت آنها نسبت به درختان AVL این است که اعمال درج و حذف با تنها یک بار پیمایش درخت از بالا به پایین و تغییر رنگ گره‌ها انجام می‌شوند.



- هر گره با یکی از دو رنگ سیاه یا قرمز رنگ آمیزی می‌شود.
- ریشه همیشه به رنگ سیاه است.
- اگر گره‌ای قرمز باشد فرزندان آن باید سیاه باشند.
- تمامی مسیرها از یک گره به برگ‌ها (گره‌های null) باید دارای تعداد مساوی گره سیاه باشند.
- تمامی برگ‌ها سیاه هستند. (برگ گره‌ای است که فرزندی نداشته باشد- همان گره‌های null).

به خاطر این که در تمام مسیرهای از ریشه به برگ، تعداد گره‌های سیاه برابر است (ویژگی ۴) و ممکن نیست دو گره قرمز پشت سر هم قرار بگیرند (ویژگی ۳)، در این نوع درخت‌ها طول بلندترین مسیر از ریشه تا برگ هیچ وقت بزرگتر از دو برابر طول کوتاه‌ترین مسیر ریشه تا برگ نمی‌شود؛ بنابراین درخت‌های قرمز-سیاه هیچ وقت دچار عدم توازن شدید نمی‌شوند و زمان اجرای لگاریتمی اعمال در این درخت‌ها تضمین شده‌است.

عملگرهای فقط خواندنی نسبت به درخت‌های دودویی ساده نیاز به تغییر ندارند ولی درج و حذف معمولی می‌تواند تاثیراتی از قبیل خارج شدن از شروط اولیه درخت‌های قرمز- سیاه داشته باشد که برای بازگردانی آن ویژگی‌ها نیاز به تغییر رنگ و چرخش درخت دارد. اگرچه این نوع درج و حذف پیچیدگی بسیاری(از لحاظ الگوریتمی و سختی پیاده‌سازی) دارد ولی بازهم نرخ جستجوی آن همچنان $O(\log n)$ می‌باشد.

درج و حذف در درختان قرمز-سیاه

با توجه به شباهت علمیات درج و حذف در این درختان تنها به شرح عملگر درج می‌پردازیم.

بنابراین می‌خواهیم یک گره (که فعلا قرمز رنگ‌شده) را در یک درخت جستجو دودویی قرمز-سیاه درج کنیم. اتفاقی که بعداً می‌افتد بستگی به شیوه رنگ آمیزی گره‌های مجاور بستگی دارد عبارت **گره عمو** به برادر گره والد اشاره دارد، همانند آنچه که در روابط خانوادگی مطرح است. توجه به موارد زیر لازم است:

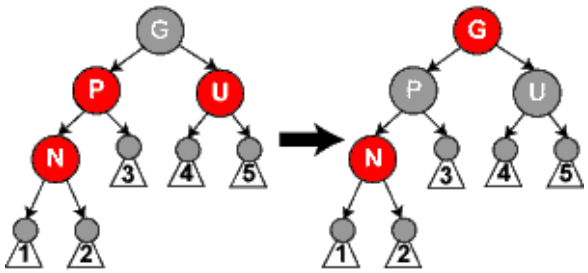
- ویژگی ۵ (همه برگ‌ها سیاه هستند (گره‌های nullهم سیاه محسوب می‌شوند)) همیشه باید حفظ شود.
- ویژگی ۳ (فرزندان گره قرمز سیاه هستند) با اضافه کردن گره قرمز و تغییر رنگ گره به سیاه یا چرخش تهدید می‌گردد.
- ویژگی ۴ (همه مسیرها باید دارای تعداد مساوی گره سیاه باشند) با اضافه کردن یک گره سیاه تغییر رنگ قرمز به سیاه یا چرخش تهدید می‌شود.

توجه کنید برچسب‌های N برای گره اضافه شده، P برای والد گره اضافه شده،G برای پدربزرگ گره اضافه شده و U برای عموی گره اضافه شده مورد استفاده می‌شود.

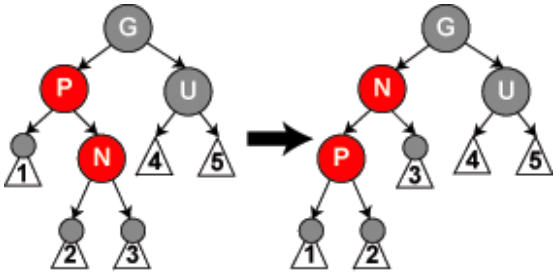
حالت ۱: گره جدید N ریشهه درخت باشد. در این حالت با تغییر رنگ به سیاه (بخاطر ویژگی ریشه باید سیاه باشد) و در هر مسیر باید یگ گره سیاه وجود داشته باشد اعتبار درخت را حفظ می‌کند.

حالت ۲: اگر گره والد سیاه باشد هیچ کدام از دو ویژگی (هردو فرزند قرمز، سیاه هستند و تعداد راه‌های از هر مسیر به برگ دارای تعداد یکسان گره سیاه دارند) مورد تهدید واقع نمی‌شوند ولی فقط وجود دوگره قرمز اعتبار درخت را زیر سؤال می‌برد (حالت ۳).

حالت ۳: اگر هردو گره والد و عمو قرمز باشد، هردو به رنگ سیاه در می‌آیند و پدر آن‌ها قرمز در می‌آید. (برابر ویژگی ۴). گره قرمز جدید دارای والد سیاه است. تا آن زمان که هر راه از والد /عمو که به‌طور قطع از پدر بزرگ نیز عبور می‌کند همچنان معتبر می‌باشد. با این وجود G(پدر والد و عمو و پدربزرگ N) باید بررسی شود که آیا ریشه‌است یا خیر (تا به رنگ سیاه درآید (ویژگی ۲) این کار را می‌توان با یک متد بازگشتی انجام داد. حتی می‌توان آن را به صورت یک حلقه نوشت که البته ثابت می‌شود این حلقه دارای تعداد مشخصی انجام عملیات چرخش است.

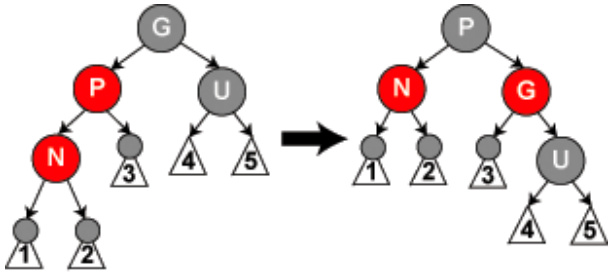


حالت ۴: اگر والد (p) قرمز باشد ولی عمو سیاه باشد همچنان گره جدید سمت راست p است و p فرزند سمت چپ G است. در این حالت، یک دوران به چپ که نقش گره جدید N را با والدش؛ P تغییر می‌دهد، قابلیت اجرا پیدا می‌کند که در آن والد قبلی با فرزندش جابجا می‌شود سپس والد قبلی که اکنون دوباره برچسب‌گذاری شده با ویژگی همه مسیرها دارای تعدا مساوی گره سیاه هستند؛ سر و کار دارد. چون بر طبق ویژگی ۴ (هر دو فرزند گره قرمز سیاه هستند) همچنان مورد تهدید بی اعتبارشدن درخت همراه است. دوران باعث می‌گردد تا در بعضی از مسیرها (از جمله مسیری که از زیر درخت برچسب‌گذاری شده)عبور می‌نماید از گره جدیدی که قبلاً مطرح نبوده‌است، عبور می‌کند ولی چون این گره مانند گره قبلی قرمز است، ویژگی ۴ با دوران اعتبار درخت را از بین نمی‌برد.
به همین ترتیب اگر والد (p) قرمز باشد ولی عمو سیاه باشد همچنان گره جدید سمت چپ p است و p فرزند سمت راست G است. در این حالت، یک دوران به راست روی والد(p) انجام می‌شود.



حالت ۵: والد قرمز است ولی عمو سیاه می‌باشد و گره جدید N فرزند چپ والد P است و P فرزند چپ پدربزرگ(G)است. در این حالت رو ی پدر بزرگ(g) دوران به راست اجرا می‌گردد. درختی که نتیجه می‌شود دارای والد ی به نام P برای هردو گره G و N است .G، به عنوان ریشه، سیاه است و این تا آن زمان که فرزندش قرمز باشد پابرجاست. با تغییر رینگ P و G درخت نهایی حاصل می‌گردد که همهٔ ویژگی‌های فوق از جمله تعداد مساوی گره سیاه در هر مسیر (به این شکل که تمامی مسیرهایی که از این سه گره عبور می‌کند که قبلاً از G عبور می‌کرده و اکنون از P عبور می‌کند) پابرجاست.

به همین ترتیب اگر والد قرمز است ولی عمو سیاه می‌باشد و گره جدید N فرزند راست والد P است و P فرزند راست پدربزرگ(G)است. در این حالت رو ی پدر بزرگ(g) دوران به چپ اجرا می‌گردد.



۴. سوالات برنامه نویسی

1. HackerEarth, Monk and his Friends
2. HackerEarth, Distinct Count
3. ACMSGU::180. Inversions
4. HackerEarth, B-Sequence

۵. برای مطالعه بیشتر

1. Sleator, Daniel Dominic, and Robert Endre Tarjan. "Self-adjusting binary search trees." Journal of the ACM (JACM) 32.3 (1985): 652-686.
2. Reed, Bruce. "The height of a random binary search tree." Journal of the ACM (JACM) 50.3 (2003): 306-332.
3. Knuth, Donald E. "Optimum binary search trees." Acta informatica 1.1 (1971): 14-25.
4. Devroye, Luc. "A note on the height of binary search trees." Journal of the ACM (JACM) 33.3 (1986): 489-498.
5. Daniel, Chengwen Chris Wang Jonathan Derryberry, and Dominic Sleator. "O (log log n)-competitive dynamic binary search trees." Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms. Vol. 122. SIAM, 2006.