



# هشت: درخت جستجو دودویی

ساختمان داده ها و الگوریتم

مدرس: دکتر نجمه منصوری

نگارنده: سجاد هاشمیان

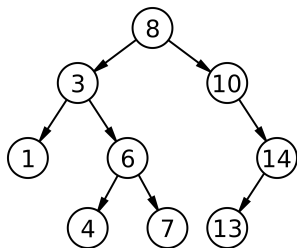
## ۱. درخت جستجو دودویی

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 جستجوی یک داده، اضافه کردن آن و حذف کردن آن بپردازیم. در واقع ایده درخت جستجو دودویی در ذخیره‌سازی گره‌هاست؛ ترتیب گره‌ها 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 داده‌های 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 منحصر به فرد 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 می‌کنیم. 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 برمی‌گردانیم. اگر کلید کمتر از کلید ریشه باشد، زیر درخت سمت چپ را جستجو می‌کنیم. به همین ترتیب، اگر کلید بزرگتر از کلید ریشه باشد، زیردرخت سمت راست را جستجو می‌کنیم. این روند تکرار می‌شود 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

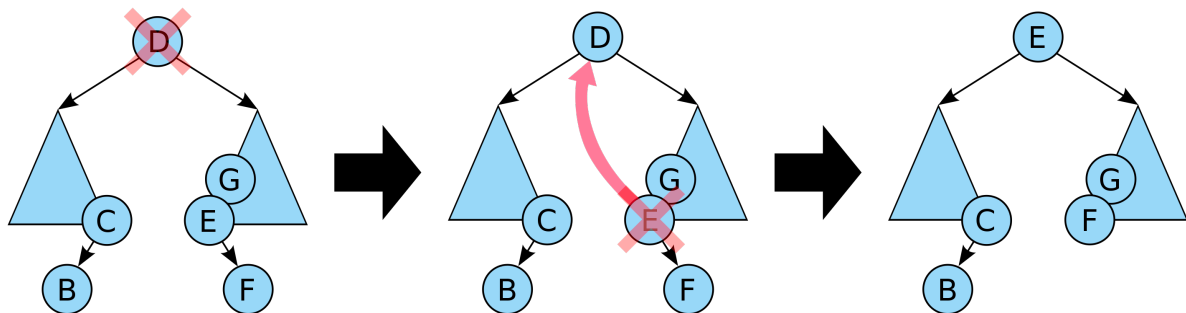
درج همانطور که جستجو شروع شد، آغاز می شود. اگر کلید برابر با ریشه نباشد، زیر درخت‌های چپ یا راست را مانند گذشته جستجو می کنیم. در نهایت، ما به یک گره خارجی خواهیم رسید و بسته به کلید این گره، کلید جدید را به عنوان فرزند راست یا چپ آن اضافه می کنیم. به عبارت دیگر، ما ریشه را بررسی می کنیم و گره جدید را اگر کلید آن کمتر از ریشه باشد در زیر شاخه سمت چپ قرار می دهیم، 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 کاملاً وابسته به ترتیب انجام عملیات‌ها و کلید‌های هر عمل در هر مرحله است؛ در واقع تعریف فعلی‌مان از این عملیات‌ها آنچنان که باید و شاید در راستا هدف ما بهینه عمل نمی‌کند، برای مثال هزینه هر ۳ عملیات معرفی شده وابسته ارتفاع درخت است که با توجه به روش انجام این عملیات‌ها، بعد از تعدادی درج، درخت می‌تواند کاملاً 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 جستجو 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 جستجو دودویی استفاده کنیم. هدف ما وارد کردن عناصر یک آرایه داده شده به طول  $n$  و با عناصر متمایز در یک درخت دودویی جستجو 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 می توان با مقادیر آرایه داده شده پر نمود به طوری که حاصل این عمل یک درخت جستجو دودویی باشد. بنابراین تعداد درختهای جستجو 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  $n$ ام کاتالان است، این یعنی:

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

در این روش 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 می کنیم 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  $n$ تایی خواهد شد. با توجه به اینکه  $n!$  برای  $n \geq 3$  بزرگتر از عدد  $n$ ام 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 جستجو 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 می توان 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 جابه جا کردن آنها را داشته باشیم کما که این اصلاً چیز مطلوبی نیست و در واقع اصلاً شرایط استفاده ندارد.

برای پیشگیری از این مهم، ایده دیگری وجود دارد، در واقع ما می توانیم 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 درخت ها را نداشته باشیم.

### ۳. درخت‌های جستجو دودویی خود متوازن کننده

همانطور که گفتیم بیشتر عملیات‌های روی یک درخت جستجوی دودویی، زمانی مستقیماً متناسب با ارتفاع درخت می‌برند. پس مطلوب است که ارتفاع را کوچک نگه داریم. یک درخت دودویی با ارتفاع  $h$  می‌تواند شامل حداکثر  $2^0 + 2^1 + 2^2 + \dots + 2^h = 2^{h+1} - 1$  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 چرخش 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 اضافه شوند که یا در تعداد کمی از مراحل انجام شوند و هزینه درگیر زیادی نداشته باشند و در غیر این صورت، هزینه محاسباتی آنها اندک و یا از زمان ثابت باشد.

از ساختمان داده‌های مشهوری که از این روش کمک می‌گیرند می‌توان به درختان AVL، قرمز-سیاه، ۲-۳ و درختان عمومی خودمتوازن وزن دار اشاره کرد.

### 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.