به نام خدا

ساختمان داده‌ها و الگوریتم‌ها (۴۰۲۵۴)

دانشگاه صنعتی شریف — تمرین سوم

مدرس: مهدی صفرنژاد — انتشار: ۳ اردیبهشت ۱۴۰۱

درخت

سؤالات را با دقت بخوانید و روی همه آن‌ها وقت بگذارید. تمرین‌های تئوری تحویل گرفته نمی‌شوند اما از آن‌ها سؤالات کوییز مشخص می‌شود. بنابراین روی سؤالات به خوبی فکر کنید و در کلاس‌های حل تمرین مربوطه شرکت کنید.

سؤال ۱. به سوالات زیر در رابطه با درخت Red-black پاسخ دهید:

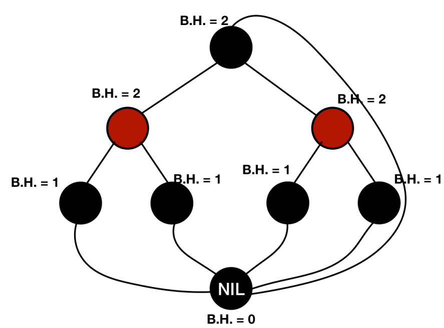
آ. در چه شرایطی بهتر است از درخت Red-black به جای AVL استفاده کنیم؟

ب. ارتفاع سیاه (black height) را برای هر گره درونی در این نوع درخت تعریف کنید. نشان دهید در هر درخت Red-black با ریشه x حداقل $n = 2^{bh(x)} - 1$ گره داخلی وجود دارد ($bh(x)$ ارتفاع سیاه گره x است.).

پاسخ:

(الف) هر دو درخت AVL و Red-Black متوازن هستند، اما در زمان اضافه کردن/حذف کردن یک گره، درخت AVL تعداد چرخش های بیشتری نیاز دارد تا دوباره متوازن شود. به همین دلیل در شرایطی که تعداد کوئری‌های زیادی برای حذف و اضافه داشته باشیم بهتر است که از درخت Red-Black استفاده کنیم. اما درخت AVL برای تعداد جست و جوی زیاد مناسب‌تر است.

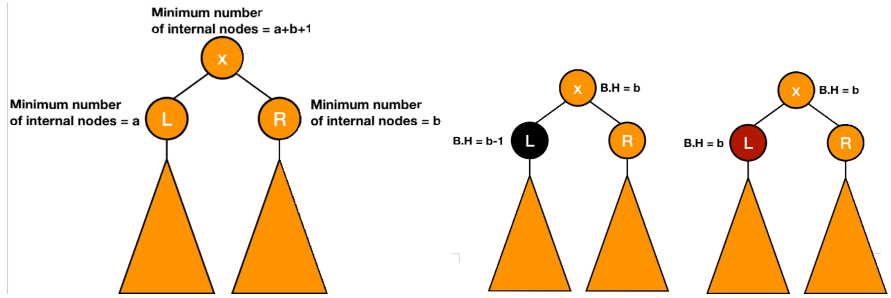
(ب) ارتفاع سیاه در یک درخت Red-Black و برای گره دلخواه x برابر تعداد گره های سیاه در یک مسیر ساده از x به یک برگ است.(توجه داشته باشید که خود گره x شمرده نمی‌شود.) برای مثال در درخت زیر ارتفاع های سیاه رئوس نشان داده شده است:



اثبات رابطه ارائه‌شده به روش استقرا است. در حالت پایه(زمانی که راس ریشه، برگ هم هست) با توجه به اینکه $bh(x) = 0$ می‌باشد میتوان دید تعداد گره‌های داخلی $n = 2^{bh(0)} - 1 = 0$ می‌باشد که مشاهده درستی است. حال فرض کنید ریشه x دو فرزند چپ و راست و ارتفاع b دارد. در این شرایط اگر رنگ یک فرزند قرمز باشد، ارتفاع سیاه آن فرزند هم b خواهد بود و در غیر این صورت $b - 1$.

با این استدلال طبق فرض استقرا می‌توان نوشت که هر گره فرزند حداقل $n = 2^{bh(root)} - 1 = 2^{b-1} - 1$ گره داخلی دارد. باید توجه داشت که اگر فرزندان راس ریشه به ترتیب دارای a و b راس داخلی باشند، راس ریشه $a + b + 1$ راس داخلی خواهد داشت.

از طرفی می‌دانیم: $a \geq 2^{bh(l)} - 1 \geq 2^{b-1} - 1$ و $b \geq 2^{bh(r)} - 1 \geq 2^{b-1} - 1$



بنابراین $n = a + b + 1 \geq (2^{b-1} - 1) + (2^{b-1} - 1) + 1 = 2^b - 1$ که حکم مدنظر ما را ثابت کرده و نشان می‌دهد تعداد گره‌های داخلی ادعا شده وجود دارند.

سؤال ۲. درخت دودویی جست‌وجو را طوری تغییر دهید تا بتوان kامین عدد را در $O(log(n))$ بدست آورد. این تغییر بر روی کدام بخش از درخت (حافظه، زمان ساخت، ...) اعمال شده است؟ مرتبه تغییر را مشخص کنید.

پاسخ:

در صورت تغییر درخت دودویی جست‌وجو به درخت بی‌نقص دودویی جست‌وجو یا $PerfectBST$ می‌توانیم عملیات پیدا کردن k امین عنصر را در $O(log(n))$ انجام دهیم. د.د.ج بی‌نقص، درختی است که تمامی گره های داخلی در آن دو فرزند دارند و تمامی گره های برگ نیز در یک ارتفاع می‌باشند. برای حل سوال ابتدا د.د.ج را با هزینه $O(n)$ به د.د.ج بی‌نقص تبدیل می‌کنیم و سپس می‌توان هر عنصر را در $O(log(n))$ پیدا کرد. برای ساخت یک د.د.ج بی‌نقص از روی یک د.د.ج ابتدا روی د.د.ج پیمایش $inorder$ انجام می‌دهیم تا دنباله اعداد به دست بیایند. توجه داشته باشید که چون این پیمایش روی یک BST انجام شده است دنباله حاصل مرتب‌شده خواهد بود. همچنین نکته قابل توجه دیگر این است که هدف، رسیدن به درخت د.د.ج بی‌نقص است و برای این موضوع امکان دارد لازم باشد تعدادی عدد به لیست اضافه کنیم. برای این امر اعداد $-\infty$ و $+\infty$ را باید به انتها و ابتدای لیست اضافه کنیم. در واقع زمانی که در جستجوی k امین عنصر بزرگ باشیم، باید $-\infty$ ها را به ابتدای دنباله اضافه کنیم و در صورت جست‌وجو برای k امین عنصر کوچک، باید $+\infty$ ها را به انتهای دنباله اضافه کنیم. حال مقدار میانی آرایه را پیدا کرده و در ریشه د.د.ج بی‌نقص قرار می‌دهیم. این کار را برای زیر دنباله راست و زیر دنباله چپ به صورت بازگشتی تکرار می‌کنیم و د.د.ج بی نقص را می‌سازیم. در د.د.ج بی‌نقص با n گره، گره میانی در جایگاه $\lceil n/2 \rceil$ قرار دارد. برای به دست آوردن k امین عنصر، در هر مرحله k را با محل گره میانی مقایسه می‌کنیم. اگر k از محل میانی کوچک‌تر بود به شاخه چپ و در غیر این صورت به شاخه راست وارد می‌شویم و در آن زیر شاخه به دنبال $(k - medianPlace)$ امین عنصر میگردیم که $medianPlace$ جایگاه عنصر میانه را نشان می‌دهد. بدین ترتیب از آن‌جایی که ارتفاع درخت از $O(log(n))$ است و در هر مرحله یا جواب یافت می‌شود یا یک لایه پایین‌تر می‌رویم می‌توانیم در زمان $O(log(n))$ به k امین عنصر دسترسی پیدا کنیم. این درخت در مسائلی که تغییرات زیادی روی خود درخت نداریم اما تعداد و انواع کوئری‌ها مختلف هستند بسیار سودمند خواهد بود!

سؤال ۳. رابطه‌ای بازگشتی برای تعداد د.د.ج های مختلف، که میتوان با اعداد $a_1 < a_2 < a_3 < ... < a_n$ ساخت را بیابید.
حال فرض کنید که یک د.د.ج ثابت داریم. رابطه‌ای بازگشتی برای تعداد دنباله‌های متفاوت از اعضای این درخت بیابید که در صورت درج آن‌ها به ترتیب، می‌توان این د.د.ج را به‌دست آورد.

پاسخ:

(الف) فرض کنید $f(n)$ تعداد د.د.ج های مختلفی باشد که میتوان با n عدد متمایز ساخت. به ازای هر $1 \leq i \leq n$ ، اگر a_i ریشه درخت باشد، همه ی اعداد $a_1, a_2, ..., a_{i-1}$ در سمت چپ این ریشه قرار میگیرند و همه ی اعداد $a_{i+1}, ..., a_n$ در سمت راست آن. حال به ازای اعداد کوچکتر، $f(i-1)$ حالت مختلف داریم و برای اعداد بزرگتر ، $f(n-i)$ حالت. پس داریم:

$$f(n) = \sum_{i=1}^{n} f(i-1)f(n-i) \rightarrow f(n) = C_n$$

که C_n همان اعداد کاتالان هستند.

(ب) مقدار $g(n)$ را تعداد دنباله‌های متفاوتی تعریف می‌کنیم که می‌توانند باعث ایجاد این د.د.ج شوند. اولین عضو این دنباله باید ریشه درخت باشد. اگر i عدد راس در سمت راست ریشه قرار داشته باشند و $n-i-1$ راس در سمت چپ، تعداد حالت های مختلف برای ایجاد زیر درحت سمت چپ برابر $g(n-i-1)$ و برای زیر درخت سمت راست $g(i)$ است. پس به $g(i)g(n-i-1)$ حالت مختلف میتوان ترتیب رئوس هر یک از زیر درخت ها را (مستقل از هم) قرار داد. پس صرفا با انتخاب کردن جای اعضای هر کدام از این دنباله‌ها در دنباله ی اصلی، ترتیبمان بدست می آید که این هم به $\binom{n-1}{i}$ حالت به دست می‌آید.
پس داریم: $g(n) = \binom{n-1}{i}g(i)g(n-i-1)$

سؤال ۴. یک د.د.ج با ارتفاع h در نظر بگیرید. نشان دهید با شروع از هر راس میتوان در $O(h+k)$ ، k عنصر بعدی آن را یافت.

پاسخ:

می‌خواهیم k عضو بعدی یک راس در پیمایش inorder را محاسبه کنیم. برای این کار فرض کنید که $x_1, x_2, ..., x_r$ دنباله رئوسی باشند که از ریشه به راس x_r میرسند.(فرض کنید راس مورد نظر x_r باشد). حال از x_r شروع می‌کنیم و پیمایش inorder را انجام می‌دهیم. متغیر s را در نظر بگیرید که در ابتدا برابر k است. در این پیمایش به ازای هر راسی که می‌بینیم مقدار s را یکی کاهش می‌دهیم. درانتهای یا حین پیمایش x_r اگر مقدار s صفر شد کار به پایان رسیده و k راسی که پیمایش کردیم جواب مسئله ماست. در غیر این صورت اولین جد x_r مانند x_t را پیدا می‌کنیم که x_{t+1} فرزند سمت چپ x_t باشد. سپس همین روند را برای x_t تکرار می‌کنیم. هر گاه مقدار s صفر شد مسئله به پایان رسیده است. میدانیم که $r \leq h$ که h ارتفاع درخت است. همچنین در هر مرحله یا در دنباله جدا بالا میرویم، یا اینکه مقدار s یکی کم می‌شود. در نتیجه حداکثر تعداد مراحل برابر $O(h+k)$ است.

سؤال ۵. آرایه‌ای شامل n عدد متمایز داریم. همچنین عدد k کوچک‌تر از n داده شده است. عددی را خوب می نامیم اگر از همه ی اعداد سمت چپ خود، و از حداقل k عدد سمت راستش بزرگتر باشد. الگوریتمی از $O(nlog(n))$ ارائه دهید که تعداد اعداد خوب در این آرایه را بیابد.

پاسخ:

با شروع از درایه ی سمت راست آرایه و حرکت به سمت چپ، درخت AVL این آرایه را میسازیم و در هر مرحله تعداد اعضای زیر درخت سمت چپ و راست را برای هر راس ذخیره میکنیم. هنگام درج کردن هر کدام از اعضای آرایه، از آن جایی که همه ی اعضای سمت راست آن در آرایه قبلا به درخت اضافه شده اند، کافی است بررسی کنیم که این عضو از چه تعداد اعضای این درخت بزرگتر است. به این ترتیب عمل میکنیم که در مراحل اضافه کردن این عضو جدید، هر گاه این عضو از یکی از راس های درخت بزرگتر بود ، تعداد کل رئوس زیر درخت سمت چپ این راس به اضافه ی خود این راس را یادداشت میکنیم. در نهایت جمع همه اعداد یادداشت شده برابر حاصل مورد نظر ما است. پس ما به ازای هر درایه تعداد درایه های سمت راست آن که از آن کوچکتر هستند را بدست آوردیم. فرض کنید رئوس ما $a_1, a_2, ..., a_n$ باشند و این تعدادی که برای راس a_i بدست آوردیم را b_i مینامیم. در هر مرحله اگر:

$$b_i \geq k \text{ و } a_i \geq max(a_{i-1}, .., a_1)$$

بود، a_i یک عدد خوب است. در هر مرحله $max(a_1, .., a_n)$ را هم آپدیت میکنیم. در کل ساخت درخت AVL و بدست آوردن b_i ها $O(nlog(n))$ زمان میبرد و پیمایش آرایه نیز $O(n)$. پس در مجموع با $O(nlog(n)$ این کار را انجام دادیم.

سؤال ۶. می‌خواهیم عمل $Tree{-}Enumerate(x, a, b)$ را بر روی زیردرخت دودویی جست‌وجو به ریشه‌ی x بنویسیم به طوری که تمام کلیدهایی را پیدا کند که مقدار آن‌ها بین a و b است. یک الگوریتم کارا از $\mathcal{O}(h + m)$ برای این کار ارائه دهید (h ارتفاع درخت و m تعداد جواب است).

پاسخ:

الگوریتم به این صورت پیاده‌سازی می‌شود:

Algorithm 1 Finding Elements On The Interval [a, b]

```
procedure Tree−Enumerate(x, a, b, Elements)
    if x.value ≤ a then
        Tree−Enumerate(x.left, a, b, Elements))
    else if x.value ≥ a then
        Tree−Enumerate(x.right, a, b, Elements)
    else
        Elements.append(x.value)
        Tree−Enumerate(x.right, a, b, Elements)
        Tree−Enumerate(x.left, a, b, Elements)
```

با توجه به کد مربوطه می‌بینیم الگوریتم از $\mathcal{O}(h + m)$ است.

سؤال ۷. داده‌ساختار «صف اولویت میانه» یا « $MeanPriorityQueue$ » شامل n عنصر مجزاست و اعمال زیر، روی این داده‌ساختار قابل اجرا می‌باشند:

- درج یک عنصر، در بدترین حالت در $\mathcal{O}(lg\, n)$
- حذف / دریافت عنصر میانه، در بدترین حالت در $\mathcal{O}(lg\, n)$

با استفاده از هرم، این داده‌ساختار را طراحی کنید و نحوه‌ی انجام اعمال فوق را دقیقاً توضیح دهید و تحلیل نمایید.

پاسخ: برای پیاده‌سازی این داده‌ساختار، از یک هرم کمینه و یک هرم بیشینه استفاده می‌کنیم. به این صورت که، کل عناصر را از نظر اندازه به دو ناحیه‌ی تقریبا مساوی بزرگتر و کوچکتر تقسیم می کنیم، به این صورت که از نظر تعداد عناصر حداکثر یکی با هم تفاوت داشته باشند. حال، ناحیه‌ی بزرگتر را درون یک هرم کمینه، و ناحیه‌ی کوچکتر را در یک هرم بیشینه قرار می دهیم. باتوجه به این پیاده‌سازی، اعمال مختلف به این صورت پیاده‌سازی می شوند:

- درج یک عنصر: آن عنصر را در هرمی که از لحاظ اندازه کوچکتر است درج می‌کنیم. اگر هم دو هرم اندازه‌شان برابر بود، فرقی نمی‌کند، در یکی درج می‌کنیم. چون درج در هرم از $\mathcal{O}(lg\, n)$ است و اندازه ی هر هرم هم تقریبا $n/2$ است، در نتیجه این عمل نیز از $\mathcal{O}(lg\, n)$ است.
- حذف عنصر میانه: می‌دانیم که عنصر میانه، یا کوچکترین عضو ناحیه‌ی بزرگتر است، و یا بزرگترین عضو ناحیه‌ی کوچکتر، و اگر هم تعداد عناصر هردو ناحیه برابر بود، می شود میانگین آن دو عضو. در نتیجه، برای پیدا کردن عنصر میانه، اگر دو هرم اندازه ی نابرابر داشتند، از هرم بزرگتر عنصر ریشه‌ی آن را خارج می‌کنیم و خروجی می‌دهیم. اگر هم دو هرم اندازه‌ی برابر داشتند، میانگین دو ریشه را خروجی می‌دهیم و عنصری را از داده‌ساختار حذف نمی‌کنیم، چون عنصر میانه در بین عناصر نیست.

سؤال ۸. فرض کنید H_1 و H_2 دو هرم بیشینه هستند که به صورت درختی (و نه با آرایه) پیاده‌سازی شده‌اند؛ بنابراین شما به ریشه‌ی هر هرم و به دو فرزند و پدر هر عنصر دست‌رسی دارید. الگوریتم $Merge{-}Heap(H_1, H_2)$ را به‌طور کامل بنویسید تا در زمان $\mathcal{O}(lg\, n)$ این دو هرم را در هم ادغام کنید و آن‌ها را به یک هرم جدید تبدیل نمایید. در صورت نیاز، در الگوریتم خود می‌توانید از اعمال تعریف شده بر روی هرم‌ها استفاده کنید. (توجه داشته باشید که ارتفاع درخت‌های H_1 و H_2 نیز از $\mathcal{O}(lg\, n)$ می‌باشد.)

پاسخ: هرم با اندازه‌ی بزرگتر را H_{max} و هرم با اندازه‌ی کوچکتر را H_{min} می‌نامیم، و ریشه‌ی هرکدام را a_{max} و a_{min} می‌نامیم. حال، اگر a_{min} بزرگتر از a_{max} بود، a_{min}را از H_{min} خارج کرده و a_{max} را در H_{min} درج می‌کنیم و بعد آن ریشه ی H_{max} را برابر a_{min} قرار می‌دهیم. سپس، H_{min} را با زیردرخت کوچکتر ریشه‌ی H_{max} ادغام می‌کنیم. حال، H_{max} بعد این تغییرات برابر نتیجه‌ی ادغام این دو درخت است. با توجه به اعمال فوق می‌بینیم اردر عمل فوق به این صورت محاسبه می‌شود:

$$f(H_1, H_2) = \mathcal{O}(height(H_{min})) + f(H_{min}, min{-}subtree(H_{max}))$$

با توجه به این معادله در می‌یابیم این عمل از $\mathcal{O}(max{-}height(H_1, H_2)$ است، که می‌شود همان $\mathcal{O}(log\, n)$.

سؤال ۹. فرض کنید که علاوه بر نشانگرهای فرزند، می‌خواهیم تعداد کل گره‌های زیردرخت یک ریشه خاص را با عنوان کلید نگهداری کنیم.

آ. تابع BSTInsert را برای نگهداری صحیح کلیدها تغییر دهید. آیا زمان اجرا تغییر می‌کند؟

ب. با داشتن مقادیر کلید، شبه‌کدی برای تابع BSTKeyLessThan(T،k) بنویسید که یک درخت T و یک عدد k را می‌گیرد و تعداد کلیدهای T را که کمتر از k هستند برمی‌گرداند. بدترین زمان اجرای این تابع چیست؟

پاسخ:

شبه‌کد به شرح زیر است.

Algorithm 2 BSTInsert(T,x)

```
x.keys = 1
if Root(T) == null then Root(T)=x
else
    y = Root(T)
    while y ≠ null do
        prev = y
        prev.keys = prev.keys + 1
        if x < y then y = Left(y)
        else y = Right(y)
    Parent(x) = prev
    if x < prev then Left(prev) = x
    else Right(prev) = x
```

زمان اجرای BSTInsert هنوز $O(h)$ است، که h ارتفاع درخت ورودی است.

Algorithm 3 BSTKeyLessThan(T,k)

```
count = 0
if x ≠ null then
    if x.value > k then
        count = BSTKeyLessThan(Left(x),k)
    else if x.value == k AND Left(x) ≠ null then count = Left(x).keys
    else count = Left(x).keys + 1 + BSTKeyLessThan(Right(x),k)
return count
```

زمان اجرای BSTKeyLessThan : میانگین حالت $T(n) = T(n/2) + O(1)$. $T(n) = O(\log n)$، که n تعداد کل گره های درخت است. بدترین حالت: $T(n) = T(n-1) + O(1).T(n) = O(n)$.

سؤال ۱۰. به سوالات زیر درباره درخت اِی‌وی‌اِل و جستجوی دودویی پاسخ دهید.

آ. فرض کنید ۷ عنصر را در یک BST درج می‌کنید. ارتفاع‌های ممکن درخت پس از درج چیست؟ (در مورد ترتیب‌های مختلف درج عناصر فکر کنید).

ب. اگر ۷ عنصر را در درخت AVL قرار دهید، ارتفاع درخت چقدر خواهد بود؟

پ. با داشتن یک درخت جستجوی دودویی، توضیح دهید که چگونه می‌توانید آن را به یک درخت AVL با حداکثر زمان $O(n\log(n))$ تبدیل کنید. بهترین زمان اجرای الگوریتم چه خواهد بود؟

ت. آیا بین ارتفاع درخت AVL و حداقل یا حداکثر تعداد گره‌های آن رابطه وجود دارد؟

پاسخ: (الف) هر ارتفاعی از ۲ (درختی که هر گره داخلی دقیقاً دو فرزند دارد) تا ۶ (یک درخت degenerate) امکان پذیر است.

(ب) یک درخت AVL با ۷ عنصر می تواند ارتفاع ۲ یا ۳ داشته باشد. نمی‌تواند ارتفاع ۴ باشد: اگر ارتفاع ۴ باشد، برای اینکه ریشه متعادل شود، یک زیردرخت باید ارتفاع ۳ و دیگری حداقل ۲ داشته باشد. یک درخت AVL با ارتفاع ۲ حداقل به ۴ گره نیاز دارد (یا ریشه درخت فرعی متعادل نخواهد شد)، بنابراین در درخت ۷ عنصری ما، ۴ عنصر برای یک زیردرخت به اضافه ریشه داریم که تنها ۲ عنصر برای زیردرخت دیگر باقی می‌ماند. اما این درخت فرعی قرار بود ارتفاع ۳ داشته باشد، بنابراین عناصر کافی برای پر کردن آن وجود ندارد. توجه داشته باشید که درخت AVL حداقل ارتفاع ممکن را برای شما تضمین نمی کند (درختان AVL با ۷ عنصر ارتفاع ۳ وجود دارند)، اما از بدترین حالت جلوگیری می کند.

(پ) از آنجایی که ما از قبل یک BST داریم، می‌توانیم یک پیمایش میان‌ترتیب روی درخت انجام دهیم تا یک آرایه مرتب‌شده از گره‌ها را بدست آوریم. اکنون می‌توانیم به سادگی همه این گره‌ها را با استفاده از چرخش‌هایی که به ما یک زمان اجرا $O(n\log(n))$ را در یک درخت AVL بازگردانیم، وارد کنیم.

(ت) رابطه ای برای حداقل تعداد گره‌ها وجود دارد و بازگشتی است:

$$
\begin{aligned}
S_{min}(h) =& 1 && \text{When } h = 0 && (۱)\\
& 2 && \text{When } h = 1 && (۲)\\
& 1 + S_{min}(h-2) + S_{min}(h-1) && \text{Otherwise} && (۳)
\end{aligned}
$$

حداکثر تعداد گره‌ها کمی ساده تر است: سطح i درخت می تواند تا 2^i گره در خود داشته باشد. با جمع کردن تمام سطوح داریم:

$$S_{max}(h) = 2^{h+1} - 1$$

سؤال ۱۱. در این سوال، در مورد درخت جستجوی سه‌گانه (TST) صحبت خواهیم کرد. درخت‌های جستجوی سه‌گانه شبیه درخت‌های جستجوی دودویی هستند، اما به جای داشتن فقط ۲ نشانگر (چپ و راست)، ۳ اشاره‌گر (چپ، وسط و راست) دارند.

درخت T یک درخت سه‌گانه است که در آن هر گره حاوی یک نقطه به شکل (x, y) برای $x, y \in Z$ است. علاوه بر این، هیچ دو نقطه‌ای در T نمی‌تواند یک مقدار x یا یک مقدار y داشته باشند. ویژگی‌های زیر برای هر گره u با کلید (x, y) در هر TST برقرار است:

- هر نقطه (x_L, y_L) در زیردرخت سمت چپ u دارای $x_L < x$ و $y_L < y$ است.
- هر نقطه (x_M, y_M) در زیردرخت وسط u دارای $x_M > x$ و $y_M < y$ است.
- هر نقطه (x_R, y_R) در زیردرخت سمت راست u دارای $x_R > x$ و $y_R > y$ است.

می‌توانید فرض کنید که برای هر گره v در درخت T، تعداد گره‌هایی را که به زیردرخت v تعلق دارند، در زمان $O(1)$ محاسبه کنید.

آ. اثبات یا رد: برای هر مجموعه‌ای از n نقطه متمایز (که در آن هیچ دو نقطه با مختصات x یا مختصات y مشترک نیستند)، یک درخت جستجوی سه‌گانه T در این n نقطه با $h(T) = O(f(n))$ وجود دارد:
- وقتی $f(n) = n$،
- وقتی $f(n) = \log n$.

ب. فرض کنید به شما یک نقطه (x', y') و یک درخت جستجوی سه‌گانه T داده شده است که شامل n نقطه و ارتفاع h است. شما می خواهید تعیین کنید که آیا (x', y') در T موجود است یا خیر.

یک الگوریتم کارآمد برای حل این سوال طراحی کنید، درستی الگوریتم را ثابت کنید و زمان اجرای آن را بر حسب n و همچنین بر حسب h تحلیل کنید.

پ. حال، فرض کنید با یک درخت جستجوی سه‌گانه T با n گره و ارتفاع h و یک نقطه (x', y')، می‌خواهید تعداد نقاط (x, y) را در درخت تعیین کنید به طوری که $x' \leq x$ و $y \leq y'$.

یک الگوریتم بازگشتی ارائه دهید که با شروع از ریشه درخت، این نقاط را جستجو می‌کند. در هر مرحله بازگشتی در یک گره u از T، الگوریتم شما در بدترین حالت به چند فرزند از u نیاز دارد؟

بدترین حالت زمان اجرا $U(h)$ الگوریتم خود را بر حسب h تحلیل کنید. بدترین حالت زمان اجرا بر حسب n چیست (زمانی که h بتواند دلخواه باشد)؟ اگر درخت کاملاً متوازن باشد و $h = \log_3 n$، زمان اجرا بر حسب n چقدر خواهد بود؟

پاسخ:

(الف) گزاره اول درست و گزاره دوم نادرست است.

اثبات گزاره اول: اگر S تمام مجموعه‌های با n نقطه باشد و (x_1, y_1) نقطه‌ای در S با کوچکترین مختصات x باشد، گره ریشه شامل (x_1, y_1) را می‌سازیم و داریم $S_R = \{(x, y) \in S : y > y_1\}$ و $S_M = \{(x, y) \in S : y < y_1\}$. توجه کنید که $S = S_R \cup S_L \cup \{(x_1, y_1)\}$ و آنها S را افراز می‌کنند. ما به صورت بازگشتی یک زیردرخت روی S_R می‌سازیم و آن را زیردرخت سمت راست ریشه قرار می‌دهیم و همچنین یک زیردرخت را روی S_M می‌سازیم و آن را به زیردرخت وسط ریشه تبدیل می‌کنیم. از آنجایی که $|S| = n$، ارتفاع این درخت واضحا $O(n)$ است. برای مشاهده اینکه ای درخت یک TST معتبر است، با فرض ما

از S_M (از S_R، همینطور) همه نقاط در زیردرخت میانی (زیردرخت راست، همینطور) ویژگی های TST را در گره ریشه برآورده می‌کنند. در واقع این عبارت در هر گره‌ای در TST که می‌سازیم درست باقی می‌ماند (می‌توان این را با استقرا روی تعداد گره‌ها ثابت کرد).

رد گزاره دوم: n نقطه با $\{(i, n-i) i \in [1, n]\}$ را در نظر بگیرید: یعنی مختصات x در حال افزایش (به i) و مختصات y در حال کاهش هستند. با توجه به ویژگی‌های TST به راحتی می توان دریافت که $(1, n-1)$ باید ریشه هر TST معتبری باشد که شامل این n نقطه است (در غیر این صورت، اگر نقطه دیگری مثل $(i, n-i)$ ریشه باشد، پس $(1, n-1)$ نمی‌تواند به هیچ یک از زیردرخت‌های آن تعلق داشته باشد. تمام $n-1$ نقطه دیگر اکنون باید به زیردرخت وسط $(1, n-1)$ تعلق داشته باشند. با تکرار این گزاره، به یک TST با ارتفاع n می‌رسیم که در آن هر گره که شامل $(i, n-i)$ دارای $(i-1, n-(i-1))$ به‌عنوان والد خود است (مگر اینکه $i = 1$) و دارای $(i+1, n-(i+1))$ به عنوان فرزند میانی آن است (مگر اینکه $i = n$).

(ب) الگوریتم: الگوریتم بازگشتی زیر با داشتن یک گره در TST کار می‌کند. فرض کنید v_L، v_M، v_R فرزندان v را نشان می‌دهند (اگر وجود نداشته باشند، NIL هستند).

Algorithm 4 Search(v)

1: **if** $[v = \text{NIL}]$ **then** return FALSE
2: **if** $[(x, y) = (x', y')]$ **then** return TRUE
3: **if** $[(x' < x) \wedge (y' < y)]$ **then** return Search(v_L)
4: **if** $[(x' > x) \wedge (y' < y)]$ **then** return Search(v_M)
5: **if** $[(x' > x) \wedge (y' > y)]$ **then** return Search(v_R)
6: **return** FALSE $\triangleright$ At this point, $x' = x$ or $y' = y$ or $(x' < x) \wedge (y' > y)$.

درستی: ابتدا، واضح است که اگر این الگوریتم (x', y') را پیدا کند، پس در TST وجود دارد (به دلیل خط ۲). اکنون ثابت می‌کنیم که اگر (x', y') در TST وجود داشته باشد، این الگوریتم آن را پیدا می‌کند.

با استقرا روی تعداد گره‌های TST ثابت می‌کنیم، با فرض اینکه (x', y') در T وجود دارد. اگر TST فقط یک گره داشته باشد، گره ریشه باید شامل (x', y') باشد، و بنابراین خط ۲ به درستی اجرا خواهد شد. حال فرض کنید که TST حاوی n نقطه است. اگر گره ریشه TST حاوی (x', y') باشد، خط ۲ به درستی اجرا خواهد شد. در غیر این صورت، یکی از نوادگان گره ریشه باید حاوی (x', y') باشد. از آنجایی که همه نقاط در TST دارای مختصات x متمایز هستند، تنها یکی از زیردرخت های گره ریشه می تواند (و باید) شامل (x', y') باشد. با توجه به ویژگی های TST، (x', y') باید یکی از شرایط ذکر شده در خطوط ۳-۵ را برآورده کند. اگر (x', y') هیچکدام از آنها را برآورده نکند، آنگاه نمی‌تواند به هیچ یک از زیردرخت‌های ریشه تعلق داشته باشد که این یک تناقض است. بنابراین با فرضیه استقرایی وقتی الگوریتم بازگشتی خود را بر روی یکی از فرزندان ریشه فراخوانی می‌کنیم، به درستی (x', y') را پیدا می‌کند (زیرا زیردرخت آن حداکثر شامل $n-1$ گره است).

زمان اجرا: در بدترین حالت، این الگوریتم از هر گره بازدید می‌کند و زمان اجرای آن $O(n)$ است. بر حسب h، این الگوریتم مسیری را از گره ریشه به گره برگ در بدترین حالت طی می کند و بنابراین $O(h)$ داریم.

(پ) الگویتم: الگوریتم بازگشتی زیر با توجه به گره TST کار می‌کند. فرض کنید v_L، v_M، v_R فرزندان v را نشان دهند (اگر وجود نداشته باشند، NIL هستند). فرض کنید c(v) تعداد گره‌های متعلق به زیردرختی باشد که در v ریشه دارند.

Algorithm 5 Count(v)

if $[v = \text{NIL}]$ **then** return 0
$(x, y) \leftarrow$ point in v
if $[(x, y) = (x', y')]$ **then** $z \leftarrow 1$
else $z \leftarrow 0$
if $[(x' \leq x) \wedge (y' \geq y)]$ **then** return Count(v_L) + Count(v_R) + c(v_M) + z
if $[(x' \leq x) \wedge (y' < y)]$ **then** return Count(v_L) + Count(v_M) + z
if $[(x' > x) \wedge (y' \geq y)]$ **then** return Count(v_M) + Count(v_R) + z
if $[(x' > x) \wedge (y' < y)]$ **then** return Count(v_M) + z

در بدترین حالت، الگوریتم در حداکثر ۲ فرزند از هر گره تکرار می شود.

زمان اجرا $U(h)$، $O(2^h)$ است زیرا ضریب انشعاب در هر گره ۲ است (یعنی حداکثر دو گره از هر گره را تکرار می‌کنیم) و ارتفاع درخت h است. (با استفاده از یک رابطه بازگشتی، می توانیم آن را به صورت $U(h) \leq 2U(h-1) + O(1)$ بنویسیم (در بدترین حالت، هر دو درخت فرعی می توانند ارتفاع $h - 1$ داشته باشند)، که منجر می شود به $U(h) = O(2^h)$.)

زمان اجرا $O(n)$ است اگر h دلخواه باشد (زیرا در بدترین حالت باید از هر گره بازدید کنیم).

اگر درخت کاملاً متوازن باشد، آنگاه $h = \log_3 n$ داریم و بدترین حالت اجرا $O(2^h) = O(2^{\log_3 n}) = O(n^{\,.log_3 2})$ (که به طور مجانبی بهتر از $O(n)$ است و کران بالای بهتری را بر حسب n به ما می دهد).

سؤال ۱۲. در یک درخت دودویی جستجوی متوازن با n عنصر، بدترین پیچیدگی زمانی برای گزارش تمام عناصر در بازه $[\,a, b\,]$ چقدر است؟ با توجه به این که تعداد عناصر گزارش شده، k تا است.

پاسخ: ابتدا چک می‌کنیم که آیا عناصر a و b در درخت موجود هستند یا خیر. سرچ یک عنصر در درخت جستجوی دودویی از اردر $O(log(n))$ می‌باشد و در نتیجه پیدا کردن عناصر a و b در این درخت از اردر $O(log(n) + log(n)) = O(log(n))$ زمان می‌برد. می‌دانیم اگر پیمایش $inorder$ را در درخت BST انجام دهیم، عددها را به صورت سورت شده به ما خروجی می‌دهد؛ پس کافی‌ست برای یافتن جواب سوال، از راس a به راس b پیمایش $inorder$ را انجام دهیم؛ پیچیدگی زمانی این پیمایش نیز $O(k)$ می‌باشد. در نهایت جواب سوال ما از اردر $O(log(n) + k)$ می‌باشد.

سؤال ۱۳. فرض کنید دو عنصر a و b از یک درخت دودویی جستجو داده شده است. الگوریتمی پیشنهاد دهید که بزرگترین عنصر در مسیر دو عنصر داده شده را بیابد. توجه داشته باشید که مسیر بین دو عدد همواره خود اعداد را هم شامل می‌شود.

(پیچیدگی زمانی باید O(n) باشد که n ارتفاع درخت است.)

پاسخ: برای یافتن بزرگ‌ترین عنصر در مسیر دو عنصر داده شده، ابتدا آن‌قدر از ریشه پایین می‌رویم تا از یک عنصر درخت، دو عدد داده شده دو طرف آن عنصر مشخص قرار گرفته باشند و مشخصا زیردرخت راست یا زیردرخت چپ آن عنصر نباشند (تا جایی

که این دو عنصر یک طرف ریشه قرار دارند، پایین می‌رویم)؛ حالا پیمایش ما به این صورت است که هرگاه در زیردرختی در حال پیمایش هستیم، اگر عنصری که در آن قرار داریم، ادامه‌ی مسیرش برای رسیدن به عنصر مدنظر ما از زیردرخت سمت چپش بگذرد، ادامه نمی‌دهیم و به زیردرخت چپ در لحظه نمی‌رویم ولی اگر مسیر ما به زیردرخت راست عنصر ما بود، ادامه می‌دهیم؛ در واقع به زیردرخت چپ نمی‌رویم در این پیمایش و نیازی به حرکت در زیردرخت‌های چپ نیست. پس بعد از رسیدن به همان راسی که دو عنصر ما دو طرفش قرار دارند، یک بار به چپ و یک بار به راست می‌رویم (با توجه به تعریف پیمایش مدنظرمان) تا بزرگ‌ترین عنصر در هر کدام از مسیرها را بیابیم. سپس ۲ عددی که به عنوان بزرگ‌ترین عددهای مسیر یافتیم را مقایسه می‌کنیم و عدد بزرگ‌تر را خروجی می‌دهیم. حالا پیچیدگی زمانی این راه‌حل را محاسبه می‌کنیم. بخش اول راه‌حل که پایین آمدن تا عنصر مشخصی از درخت است، در بدترین حالت از اردر $O(n)$ می‌باشد. بخش دوم راه‌حل که محاسبه‌ی بزرگ‌ترین عنصر مسیر از راس خاصی تا هر کدام از a و b می‌باشد، از $O(n) + O(n) = O(n)$ می‌باشد. ذخیره کردن مداوم ماکسیمم مسیر هم در بدترین حالت از اردر زمانی $O(n)$ می‌باشد. در نتیجه اردر زمانی راه‌حل $O(n)$ می‌باشد.

سؤال ۱۴. فرض کنید آرایه‌ی $arr[]$ را در اختیار دارید که شامل اعداد صحیح غیرتکراری است. آرایه‌ی $temp$ را به گونه‌ای بسازید که $temp[i]$ برابر تعداد عناصر سمت راست عنصر $arr[i]$ باشد به گونه‌ای که از این عدد کوچک‌تر باشند. توضیح دهید ساخت این آرایه با درخت AVL به چه صورت است و مرتبه‌ی زمانی آن را حساب کنید.

پاسخ: برای ساخت آرایه‌ی $temp$ با استفاده از درخت AVL، از سمت راست آرایه‌ی arr شروع به حرکت می‌کنیم و تک‌تک عناصر را داخل درخت AVL، $insert$ می‌کنیم. سپس چک می‌کنیم که جایی درخت AVL به هم نریخته باشد و ارتفاع زیردرخت راست و چپ آن، بیش از ۱ نشده باشد، که در این صورت باید $rotation$هایی انجام دهیم تا درخت AVLما اصلاح و بالانس شود. حالا چک می‌کنیم که آیا $Node$ ما زیردرخت چپ دارد یا خیر، اگر دارد همه‌ی اندازه‌ی $Node$هایی که در زیردرخت چپ آن هستند را به یک متغیری به نام $count$ اضافه می‌کنیم. سپس این $Node$ را با پدرش مقایسه می‌کنیم، اگر اندازه‌ی آن بیشتر از اندازه‌ی پدرش بود، اندازه‌ی $Node$ پدرش و اندازه‌ی تمام زیردرخت چپ آن را به $count$ اضافه می‌کنیم و این مقایسه را مدام انجام می‌دهیم با $Node$ پدربزرگ و به سمت بالا...، تا برسیم به $root$؛ در نهایت عدد $count$ نهایی، تعداد اعدادی‌ست که سمت راست عدد موردنظر ما قرار دارند و کوچک‌تر از آن هستند و همه‌ی $count$ها را به $temp$ اضافه کرده‌ایم در این راه؛ بزرگ‌ترین $count$، همان جواب مدنظر ماست. برای محاسبه‌ی پیچیدگی زمانی، پیچیدگی زمانی مراحل مختلف را محاسبه می‌کنیم. بخش $insert$ شدن عنصر اول تا عنصر آخر آرایه، n بار انجام می‌شود و در نتیجه، پیچیدگی زمانی کل آن $O(n)$ می‌باشد. حالا مراحلی که در این n بار، طی می‌شوند را بررسی می‌کنیم. از قبل می‌دانیم که $insert$ کردن و $rotate$ و بالانس کردن درخت AVL، مجموعا $O(log(n))$ زمان می‌برد. جمع کردن $count$ با زیردرخت چپ آن، از اردر $O(1)$ می‌باشد و سپس $nodeparent$ای داریم که اردر آن هم $O(1)$ می‌باشد. مسئله‌ی اصلی اینجا تعداد دفعاتی‌ست که که برای یک عنصر، پدر و پدربزرگ و عناصر بالاتر آن چک می‌شوند که با توجه به در نظر گرفتن $worstcase$ در این بخش، می‌توانیم اردر این بخش را $O(log(n))$ در نظر بگیریم. در نهایت پیچیدگی زمانی راه‌حل ما از اردر $O(nlog(n))$ می‌باشد.

سؤال ۱۵. برای عدد طبیعی $k \geq 1$، روش مرتب سازی سریع رندوم ترکیبی، الگوریتمی است که برای $n > k$ از مرتب سازی سریع رندوم و برای $n \leq k$ از مرتب سازی درجی استفاده می کند. به ازای چه مقادیری از k این الگوریتم در $O(nlogn)$ کار می کند؟

پاسخ: زیر آرایه‌ای تولید شده توسط pivot در مرتب سازی سریع رندوم را در نظر بگیرید که i عنصر دارد و $i \leq k$. می‌دانیم که پیچیدگی زمانی الگوریتم مرتب سازی درجی از $O(x^2)$ است که x تعداد عناصر آرایه می‌باشد. مشخص است که برای آرایه اولیه با

سایز n تعداد زیرآرایه‌هایی که با مرتب سازی درجی مرتب می‌شوند از اردر $\frac{n}{k}$ می‌باشد، همچنین مشخص است که اندازه هرکدام از این آرایه‌ها از اردر k است. پس در کل پیچیدگی زمانی حاصل از بخش درجی مرتب سازی به صورت زیر محاسبه می‌شود:

$$T_i = O((\frac{n}{k}) \times (k^2)) = O(nk)$$

می‌دانیم که اگر برای زیر آرایه‌های با اندازه کوچکتر از k از مرتب سازی درجی استفاده نمی‌شد، عمق بازگشتی مرتب سازی سریع $O(logk)$ مرحله بیشتر می‌شد و مانند یه مرتب سازی سریع عادی به $O(logn)$ مرحله می‌رسید. پس در این الگوریتم، عمق بازگشتی مرتب سازی سریع از اردر $O(logn - logk) = O(log(\frac{n}{k}))$ می‌باشد. در هر مرحله مرتب سازی سریع، پیچیدگی زمانی کار انجام شده از اردر خطی می‌باشد. پس در کل برای پیچیدگی زمانی بخش مرتب سازی سریع در این الگوریتم داریم:

$$T_q = O(nlog(\frac{n}{k}))$$

درنتیجه پیچیدگی زمانی کلی این الگوریتم به صورت $O(nk + nlog(\frac{n}{k}))$ می‌باشد.
در حالات $k = 1$ و $k = logn$ پیچیدگی زمانی الگوریتم به صورت $O(nlogn)$ در می‌آید.

موفق باشید