ساختمان‌داده‌ها
بهار ۱۴۰۰
استاد: حسین بومری
گردآورنده:علی سرائر

دانشگاه صنعتی شریف
دانشکدهٔ ریاضی

جلسهٔ هجدهم

مسئله‌ای برای علاقه‌مندان

دو heap را چگونه می‌توان ادغام کرد؟مرتبهٔ زمانی آن چیست اگر
الف) heap را به صورت آرایه پیاده سازی کرده باشیم.
ب) heap را به صورت درختی پیاده کرده باشیم.

حال به سراغ درس می‌رویم.

جستجو:

ابتدا مرتبهٔ زمانی تعدادی از اعمال داده ساختارها را بررسی می‌کنیم.

Sorted Array

- پیدا کردن i امین عنصر
از آنجا که آرایه مرتب شده است واضح است که این عمل در $O(1)$ انجام‌پذیر است.

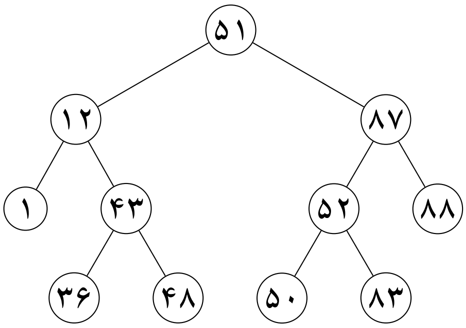
- hasElement & findElement
مرتبهٔ زمانی این دو نیز با استفاده از BinarySearch از $O(log\, n)$ است.

- delete & insert این دو عمل از $O(n)$ انجام‌پذیر هستند.(چون برای پاک کردن یا اضافه کردن یک عنصر، در بیشترین حالت باید همهٔ اعضای دیگر آرایه شیفت داده شوند تا آرایه سورت شده باقی بماند.)

به جای Sorted Array می‌توان از درخت‌های جستجوی دودویی استفاده کرد.

Binary Search Tree

اعضای درخت را طوری می‌چینیم که تمام اعدادی که بچه‌های سمت راست هر نود هستند، از آن بزرگ‌تر باشند و تمام اعدادی که بچه‌های چپ آن نود هستند، از آن کوچک‌تر باشند(به عنوان مثال بچهٔ چپ بچهٔ راست یک نود، از آن نود کوچک‌تر است. همین‌طور است برای تمام بچه‌های بچهٔ چپ این نود) شکل زیر یک مثال از درخت جستجوی دودویی است.



حال برای اعمال درخت یک تحلیل زمانی اولیه می‌دهیم.

تحلیل مرتبه زمانی

- hasElement & findElement
واضح است که برای پیدا یک عنصر یا اینکه عنصر به خصوصی در BST وجود دارد یا خیر، باید ارتفاع درخت را طی کنیم(وقتی به برگ رسیدیم و پیدا نشد یعنی عنصر وجود ندارد). بنابراین مرتبهٔ زمانی از $o(H)$ است که H ارتفاع درخت است.[۱]

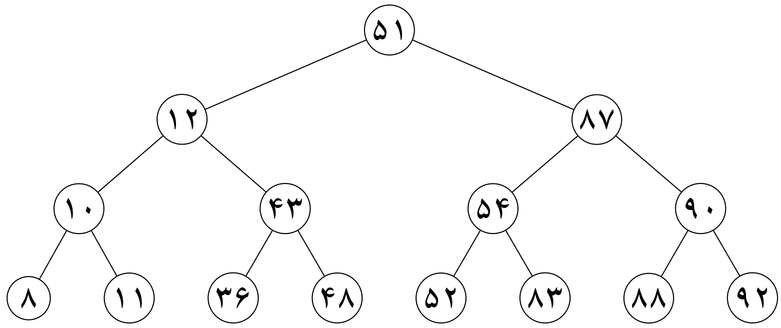
- delete
سه حالت مختلف داریم:

  – اگر عنصری که قرار است حذف شود برگ باشد که مشکلی نیست، فقط کافی است نود اشاره‌گر مربوط به بچهٔ پدر این نود را null کنیم و کار تمام است. بنابراین مرتبهٔ زمانی این کار، مرتبهٔ زمانی یافتن عنصر مربوطه است.
$$T(n) = O(Search) + O(۱)$$

  – اگر نود مربوط فقط یک بچه داشته باشد، کافی است بچهٔ این نود را به عنوان بچهٔ پدر نودی که می‌خواهیم پاک کنیم قرار دهیم. بنابراین مرتبهٔ زمانی این کار نیز مرتبهٔ زمانی یافتن عنصر مربوطه است.
$$T(n) = O(Search) + O(۱)$$

  – نود مربوط دو بچه داشته باشد.
برای این حالت مثال زیر را در نظر بگیرید. فرض کنید می‌خواهیم عدد ۱۲ را از درخت زیر حذف کنیم. دو کار می‌توانیم انجام دهیم. یا باید برویم مینیمم سمت راست را پیدا کنیم و یا باید ماکسیمم سمت چپ را پیدا کنیم. فرض کنید می‌خواهیم مینیمم سمت راست را پیدا کنیم. بنابراین ابتدا سمت راست می‌رویم و روی نود عدد ۴۳ قرار می‌گیریم. حال از اینجا آن‌قدر چپ می‌رویم تا به یک برگ برسیم و یا به یک نودی برسیم که فقط بچه سمت راست دارد و مینیمم را پیدا کنیم(واضح است که این نود همان مینیمم است). در مثالی که زدیم مینیمم همان عدد ۳۶ می‌شود. حالا این نود مربوط به مینیمم را با نودی که می‌خواستیم پاک کنیم جایگذاری می‌کنیم و به این ترتیب عنصر مطلوب پاک شده است. دقت کنید که نود مینیمم فقط یک بچه، آن هم بچه راست بیشتر نمی‌تواند داشته باشد، که اگر این بچه را داشته باشد، قبل از جابجا کردن این نود با عنصری که می‌خواستیم پاک کنیم، کافی است این بچهٔ راست مینیمم را به پدر مینیمم وصل کنیم.

[۱] دقت کنید که مینیمم ارتفاع درخت $log\,n$ و ماکسیمم n است. به این صورت که در حالت مینیمم درخت متوازن است(در مورد متوازن بودن درخت در جلسات آینده بیشتر صحبت می‌شود.) و حالت ماکسیمم این است که همهٔ عناصر پشت سر هم وصل شده اند(هر نود فقط یک بچه داشته باشد)



واضح است که مرتبهٔ زمانی این عمل در بدترین حالت همان $O(H)$ است.
یک نکته: دقت کنید که گفتیم در درخت جستجوی دودویی تمام بچه‌های سمت راست یک نود، از آن نود بزرگ‌تر و تمام بچه‌های سمت چپش از آن کوچک‌ترند. اگر به شکل هر درخت، مثل شکل بالا دقت کنید، می‌توانید به راحتی نتیجه بگیرید که اگر تمام عناصر درخت را روی یک خط افقی زیر آن تصویر کنید، به یک آرایهٔ مرتب شده دست خواهید یافت. به این صورت که اعداد به شکل زیر تصویر می‌شوند

۸, ۱۰, ۱۱, ۱۲, ۳۶, ۴۳, ۴۸, ۵۱, ۵۲, ۵۴, ۸۳, ۸۷, ۸۸, ۹۰, ۹۲

سؤال: چند درخت دودویی مختلف می‌توانیم داشته باشیم؟
جواب: از ریشه شروع می‌کنیم. اگر k عنصر سمت چپ این نود داشته باشیم و تعداد اعضای درخت n تا باشند، بنابراین $n-k$ تا عضو سمت راست این نود داریم. اگر فرض کنیم تعداد روش‌های مختلفی که می‌توان ریشه را این نود به خصوص قرار داد $S(n)$ است، می‌توان به صورت بازگشتی تعداد مربوط به بچهٔ راست و بچهٔ چپ را به صورت جداگانه به دست آورد و در هم ضرب کرد.

$$S(n) = \sum_{k=۱}^{n} S(k-۱) * S(n-k)$$
$$S(۰) = ۱$$

فرمول بالا به فرمول کاتالان معروف است، و جواب آن برابر با مقدار زیر است

$$S(n) = \frac{۱}{n+۱}\binom{۲n}{n}$$

– getSorted در این تابع می‌خواهیم اعداد درخت را به صورت sortedArray بگیریم.
می‌توانیم این‌کار را به صورت بازگشتی انجام دهیم.به این صورت که از ریشه شروع می‌کنیم، و به بچهٔ سمت راست و چپش دستور می‌دهیم که آرایه سورت شدهٔ خودشان و تمام بچه‌هایشان را پس دهند. آن‌ها نیز روی بچه‌های چپ و راستشان همین عمل را تکرار می‌کنند. مرتبهٔ زمانی این الگوریتم نیز به صورت زیر محاسبه می‌شود. (فرض می‌کنیم هر نود k تا بچه در چپ و $n-k$ بچه در راست داشته باشد)

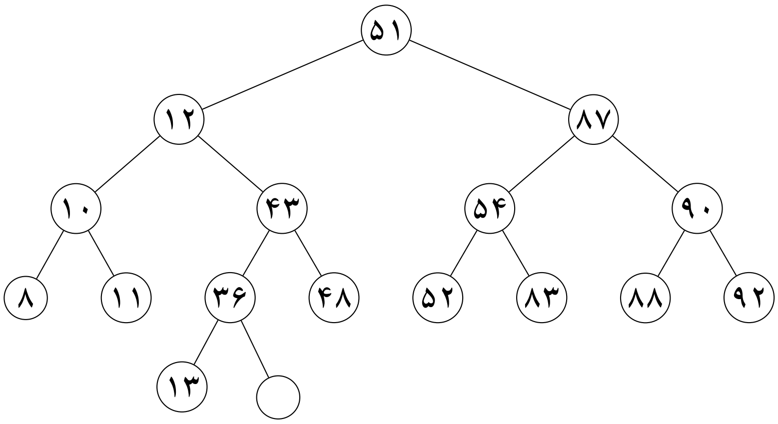
$$T(n) = T(n-k) + T(k-۱) + O(۱)$$
$$\rightarrow T(n) = O(n)$$

که $O(۱)$ بالا از این آمده که خود عددی که روی آن هستیم(که در مرحلهٔ اول ریشه است) را در آرایه، در جای خود قرار دهیم.

- insert

در این تابع می‌خواهیم یک عنصر داده شده را در جای مناسب در درخت درج کنیم. برای این‌کار از ریشه شروع می‌کنیم و این عنصر را با عناصر درخت مقایسه می‌کنیم. مثلا اگر از ریشه کمتر بود به سمت بچه چپش می‌رویم

و باز با این نود عنصر را مقایسه می‌کنیم. همین کار را ادامه می‌دهیم تا به جایی برسیم که نودی وجود نداشته باشد. به عنوان مثال دوباره شکل بالای صفحهٔ ۳ را در نظر بگیرید. فرض کنید می‌خواهیم عدد ۱۳ را در آن درج کنیم. ابتدا آن را با ۵۱ مقایسه می‌کنیم. چون ۱۳ از ۵۱ کمتر است آن را با بچهٔ چپش مقایسه می‌کنیم. چون ۱۳ از ۱۲ بیشتر است سپس آن را با بچهٔ سمت راست ۱۲ مقایسه می‌کنیم. جال ۱۳ از ۴۳ کم‌تر است پس به سمت چپ می‌رویم و پس آن چون ۱۳ از ۳۶ کوچک‌تر است باید آن را با بچهٔ چپ ۳۶ مقایسه کنیم. اما واضح است که ۳۶ بچه سمت چپ ندارد. همینجا می‌ایستیم و عدد ۱۳ را درج می‌کنیم. درخت پس از insert کردن عدد ۱۳ به صورت زیر درمی‌آید.



از این توضیحات واضح است که مرتبهٔ زمانی این عمل نیز از $O(H)$ است.

- upperBound & lowerBound

تابع upperBound کوچک‌ترین عددی که در درخت از عنصر داده شده در تابع بزرگ‌تر است را برمی‌گرداند و تابع lowerBound نیز بزرگ‌ترین عدد که در درخت از عنصر داده شده در تابع کوچک‌تر است را برمی‌گرداند. در مورد مرتبهٔ زمانی این دو خودتان فکر کنید.

ساخت درخت متوازن

اگر به مرتبهٔ زمانی‌های توابع بالا دقت کنیم، می‌بینیم که هر چه ارتفاع درخت کمتر باشد، الگوریتم‌ها بهینه می‌شوند. بنابراین بهترین حالت این است که درخت ما متوازن باشد(به عبارت دیگر داشته باشیم $H = log\, n$) حال باید سعی کنیم درخت را به شکل متوازن بسازیم.
یک ایده می‌تواند این باشد که که میانه را به عنوان ریشه در نظر بگیریم. واضح است که انتخاب میانه به عنوان ریشه بسیار خوب است؛ چرا که نصف اعداد سمت راست آن و نصف دیگر سمت چپ قرار خواهند گرفت. همین کار را برای بچه‌های راست و چپ ریشه به صورت بازگشتی حساب کنیم. بنابراین باید میانه را هر دفعه برای اعداد راست و چپ به دست آوریم.

محاسبه مرتبهٔ زمانی

مرتبهٔ زمانی این شکل ساختن درخت متوازن به صورت زیر می‌شود

$$T(n) = ۲T(\frac{n}{۲}) + O(n)$$
$$\rightarrow T(n) = n\, log\, n$$

که در محاسبهٔ مرتبهٔ زمانی فوق برای پیدا کردن میانه، از همان الگوریتم median of medians که در جلسات گذشته گفته شده استفاده کردیم تا میانه هر n عدد را در $O(n)$ به دست آوریم.

توجه: روش دیگر می‌توانست این باشد که ابتدا اعداد را سورت کنیم و پس از آن می‌توان میانه را از $O(۱)$ پیدا کرد. بنابراین

$$T(n) = ۲T(\frac{n}{۲}) + O(۱)$$
$$\rightarrow T(n) = O(n)$$

اما برای سورت کردن اعداد باز هم باید حداقل $O(n\,log\,n)$ زمان مصرف کنیم(پس عملا این کار با ایدهٔ قبلی فرق چندانی ندارد).

محاسبهٔ مرتبهٔ حافظه

هر عنصری سه مقدار دارد. یکی ارزش داخل خودش، یکی نود بچهٔ راستش و دیگری نود بچهٔ چپش. بنابراین مرتبهٔ حافظه از $O(n)$ است.

مرتبهٔ زمانی توابع برای درخت متوازن

- nextElement
برای پیدا کردن عنصر بزرگ بعد از یک عنصر خاص،دو حالت ممکن است پیش بیاید.

  - عنصر ورودی بچهٔ راست داشته باشد.
یدبه بچهٔ سمت راست این عنصر برویم و پس از آن تا جایی که ممکن است به بچه‌های سمت چپ بریم. آخرین عنصر کوچک ترین عدد ممکن است که از عنصر مورد نظر ما بزرگ‌تر است که همان nextElement می‌شود. واضح است که مرتبهٔ زمانی این عمل از $O(H)$ است که برای درخت متوازن از $O(log\,n)$ می‌شود.
  - عنصر ورودی، که آن را a می‌نامیم، بچهٔ راست نداشته باشد.
اگر این عنصر فرزند چپ یک نودی بود، به این معنی است که این عنصر از پدرش کوچک‌تر است. بنابراین یک کاندیدا برای nextElement پدر این عنصر است. حالا اگر پدر a بچهٔ چپ یک عنصر باشد، که آن را g می‌نامیم، به این معنی است که g قطعا، طبق تعریف درخت دودویی جست‌وجو، از a کوچک‌تر است. بنابراین پدر a همان nextElement می‌شود. و اگر پدر a بچهٔ راست یک عنصر باشد، که آن را l می‌نامیم، به این معنی است که باز هم طبق تعریف این درخت، l از a کوچک‌تر است و بازم هم پدر a همان nextElement است. مرتبهٔ زمانی این حالت نیز واضح است از $O(۱)$ است.

  پس در بدترین حالت از دوحالت بالا، مرتبهٔ زمانی این تابع از $O(log\,n)$ می‌شود.

- ith element
اگر در درخت، به ازای هر عنصری تعداد بچه‌هایش را داشته باشیم، مرتبهٔ زمانی از $O(H) = O(log\,n)$ می‌شود. به این صورت که اگر i از تعداد بچه‌های یک سمت بیشتر بود که به آن سمت نمی‌رویم. اگر کم‌تر بود به آن سمت می‌رویم و به این نود دستور می‌دهیم که $i - ۲$ امین عنصر بزرگ را برگرداند. این کار را به صورت بازگشتی تکرار می‌کنیم تا i امین عنصر بزرگ را پیدا کنیم.

توجه: در روش‌هایی که برای ساخت درخت متوازن گفت یک مشکلی وجود دارد. اگر داده ساختارمان استاتیک باشد و تغییر نکند، یا به عبارت دیگر از توابعی مانند insert و یا delete استفاده نشود، روش ساختی که با استفاده از میانه‌ها گفتیم روش خوبی است. اما اگر مثلا میانهٔ درخت را پاک کنیم، بهینه بودن ارتفاع درخت از بین می‌رود.

باید الگوریتم یا الگوریتم‌هایی معرفی کنیم که توازن خود را به صورت دینامیک نگه دارند، یا به اصطلاح Self Balancing BST باشند.

در جلسهٔ بعدی در مورد دو روش بهینهٔ ساخت درخت متوازن صحبت می‌کنیم. که عبارتند از

- Red Black Tree
- AVL Tree