جزوه جلسه اول داده ساختارها و الگوریتم

۲۸ شهریور ۱۴۰۰

فهرست مطالب

۱ آشنایی با الگوریتم ها

موضوع کلی درس درمورد روندهای بهینه برای حل مسائل با مقیاس نسبتا بزرگ در کنار تحلیل و طراحی این روندها است.
مسائل با مقیاس بزرگ با داده هایی نسبتا بزرگ نیز سروکار دارند. اما منظور از داده های بزرگ(Big Data) چیست؟ داده ها هنگامی Big Data تلقی میشوند که در یک کامپیوتر جا نشوند و برای ذخیره سازی و استفاده از آنها مجبور به استفاده از چند کامپیوتر یا سرور باشیم.
در مورد بهینگی یک الگوریتم میتوان به مواردی مانند پایین بودن زمان اجرا، کم بودن حافظه و منابع مصرفی، درستی الگوریتم برای داده های مختلف، کلی بودن الگوریتم، ساده بودن راه حل، خلاقانه بودن الگوریتم و مقیاس پذیر بودن آن اشاره کرد. منظور از مقیاس پذیری یک الگوریتم چیست؟ الگوریتمی را تصور کنید که برای ۵۰۰۰ ورودی به درستی کار میکند. حال اگر تعداد ورودی ها به ۱۰۰۰۰ تا افزایش یافت در اینصورت زمان اجرای الگوریتم و منابع مصرفی به چه مقدار تغییر میکنند؟ در این مثال اگر زمان اجرا و حافظه مورد استفاده دو برابر شوند، الگوریتم مقیاس پذیر محسوب میشود اما اگر الگوریتم برای ۱۰۰۰۰ داده اصلا کار نکند الگوریتم مقیاس پذیر محسوب نمیشود.
مفهوم بعدی Test of Time است. Test of Time به این معنیست که کاربردی بودن یک الگوریتم تنها توسط زمان ثابت میشود. یکی از دلایلی که امروزه از الگوریتم های چندین دهه قبل استفاده میکنیم همین مفهوم است.

۲ مثال های اولیه از الگوریتم های ساده

۱.۲ مسئله پیدا کردن قله یک آرایه یک بعدی

آرایه ای از اعداد به طول n را در نظر بگیرید. عضوی از آرایه را قله مینامیم اگر از دو عضو (برای عناصر ابتدا و انتها یک عضو) همسایه خود کوچکتر نباشد. آرایه مدنظر از ابتدا در دسترس نیست و برای فهمیدن هر عضو آن باید آنرا بپرسیم. هدف این است که با کمترین تعداد پرسش یک قله پیدا کنیم.
ساده ترین راه برای پیدا کردن قله پرسیدن تمامی اعضا و پیدا کردن قله است. برای اینکار نیاز به n پرسش داریم. راه حل دیگر این است که از یک سمت شروع به پرسش کنیم تا به یک قله برسیم. در این حالت نیز بد بدترین حالت نیاز به n پرسش داریم.
یکی از الگوریتم های بهینه برای پیدا کردن قله این است که سه عضو از میانه آرایه را بپرسیم. اگر عضو وسطی از بین این سه عضو قله بود که مسئله حل میشود. اگر این سه عضو به صورت صعودی (نزولی) مرتب شده بودند به سمت راست (چپ) حرکت کرده و مجددا سوال میپرسیم. اگر قله پیدا شد که الگوریتم به پایان میرسد و در غیر اینصورت به حرکت و پرسش روی اعضای آرایه ادامه میدهیم. این الگوریتم یا با پیدا یک قله قبل

از رسیدن به انتهای آرایه به پایان میرسد یا عضو انتهایی آرایه خود قله میشود.
حالت دیگری که بعد از پرسش سه عضو میانی با آن مواجه میشویم، حالتی است که عضو وسط از دو عضو دیگر کوچکتر باشد که در این حالت به یکی از دو سمت چپ یا راست حرکت میکنیم و مانند حالت صعودی/نزولی به پیدا کردن قله میپردازیم. در این الگوریتم حداکثر تعداد پرسش ها حدودا $n \div 2$ است.
اما بهینه ترین راه، نصف کردن آرایه است. در هر مرحله سه عضو میانی را میپرسیم که یا قله را پیدا کردیم یا بسته به حالت سه عضو که پرسیدیم یکی از دو نصفه چپ یا راست را انتخاب میکنیم و مجددا با پرسیدن سه عضو میانی به صورت بازگشتی مسئله را حل میکنیم. در اینصورت تعداد سوالهایی که میپرسیم در بدترین حالت به تعداد $3log_2n$ است. بدیهی است که برای nهای بزرگ داریم: $n \div 2 >> 3log_2n$

جزوه جلسه دوم داده ساختارها و الگوریتم

۳۰ شهریور ۱۴۰۰

فهرست مطالب

۱ ادامه مثال حل شده در جلسه قبل

در ادامه مسئله پیدا کردن قله، یک آرایه دو بعدی را در نظر میگیرم و عضوی را پیدا میکنیم که از تمامی همسایه های خود در هر ۴ جهت کوچکتر نباشد.
یکی از راه حل ها پیدا کردن قله در هر سطر از جدول و ریختن آنها در یک آرایه دیگر و پیدا کردن مجدد قله در آرایه یک بعدی جدید است. طبق مطالب جلسه قبل، تعداد پرسش ها در این راه برابر است با $n3log_2n + 3log_2n$. اما با یک مثال نقض میتوان نادرست بودن این راه حل را اثبات کرد.
راه حل دیگر، پیدا کردن عضو ماکسیمم در کل جدول است. بدیهی است که برای پیدا کردن عضو ماکسیمم باید تمامی اعضای آرایه پرسیده شوند(اگر حتی یکی پرسیده نشود احتمال داره همان عضو ماکسیمم باشد). پس این راه حل جواب درست را ارائه میدهد اما مشکل آن پیچیدگی زمانی بالای آن است(n^2).
یک راه حل بهینه برای مسئله شرح داده شده است. عضو ماکسیمم را در سطر وسط پیدا میکنیم. این عضو یا از دو عضو بالایی و پایینی خود بزرگتر است و قله را پیدا کردیم و یا از حداقل یکی از اعضای بالایی و پایینی کوچکتر است. پس عضو بزرگتر را انتخاب میکنیم و قله را در نیمه مربوط به آن عضو جستجو میکنیم. در این روش تعداد پرسش ها برابر است با $nlog_2n$(n پرسش در هر مرحله و log_2n مرحله در کل).
یک الگوریتم بهینه دیگر نصف کردن جدول در هر مرحله با پیدا کردن ماکسیمم هر سطر یا ستون است. مانند الگوریتم قبل در سطر وسط ماکسیمم را پیدا میکنیم و با توجه به اینکه عضوی که پیدا میکنیم قله است یا خیر، جدول را نصف میکنیم. اما این بار در ستون جدید که طول آن $n/2$] است مجدد عضو ماکسیمم را پیدا میکنیم و اگر قله پیدا نشده بود جدول جدید، جدولی به ابعاد $n/2$ خواهد بود که قله را در آن جدول پیدا خواهیم کرد. در این الگوریتم حداکثر تعداد پرسش های مورد نیاز برابر میشود با:

$$n + n/2 + n/2 + n/4 + n/4 + n/8 + ... \cong 3n$$

۲ الگوریتم چیست؟

به طور کلی یک دستور آشپزی یا یک دستورالعمل کشاورزی یک الگوریتم به حساب می‌آیند. در دنیای ریاضیات، ضرب اعداد، تجزیه اعداد، الگوریتم اقلیدس برای پیدا کردن ب.م.م اعداد، غربال اراتستن برای پیدا کردن اعداد اول و یا حذف گاوسی برای حل چند معادله چند مجهولی نمونه هایی از الگوریتم هستند.
اما تعریف دقیق الگوریتم: یک الگوریتم توسط یک شبه کد نوشته یا توصیف میشود که این شبه کد در مدل محاسباتی اجرا میشود. برای مثال میتوان یک برنامه(الگوریتم) را درنظر گرفت که توسط یک زبان برنامه نویسی(شبه کد) نوشته شده و روی یک کامپیوتر(مدل محاسبات) اجرا میشود.

یک الگوریتم باید قوانین شبه کد را رعایت کند و این بدین معنی است که برای توصیف الگوریتم از چه دستورهایی میتوان استفاده کرد و از چه دستورهایی نمیتوان، زیرا این شبه کد در یک مدل محاسبات اجرا خواهد شد.

۳ مدل های محاسبات

مدل محاسباتی معادل یک کامپیوتر در دنیای ریاضیات است که خود به چندین دسته تقسیم میشود.

۱.۳ ماشین تورینگ

ماشینی ساده که برای اثبات قابل اجرا بودن یا نبودن یک کار توسط کامپیوتر به کار گرفته میشد.

۲.۳ ماشین دسترسی تصادفی(Random Access Machine)

هر ماشین دسترسی تصادفی میتواند سه عمل را انجام دهد:
۱. تعداد ثابتی خانه از حافظه بارگذاری کند
۲. تعداد ثابتی عملیات روی آنها انجام دهد
۳. تعداد ثابتی خانه را در حافظه دخیره سازی کند
این سه عمل یک مرحله نامیده میشود.
حال توضیحتی راجع به خانه های حافظه: هر خانه از حافظه یک کلمه(Word) نامیده میشود. به عبارت دیگر هر ورد اندازه خانه های یک حافظه را نشان میدهد که حداقل آن به اندازه $clog_2n$ است که c یک ضریب ثابت است. آرایه ای به طول n را درنظر میگیریم. هر عضو آرایه طبق مطالب ذکر شده، اندازه‌اش $clog_2n$بیت میباشد و همانطور که پیداست این اندازه ثابت نیست. بدیهی است که با log_2n بیت میتوان حداکثر عدد n را نشان داد.
در این ماشین، زمان انجام الگوریتم برابر است با تعداد مراحل انجام شده و میزان حافظه نیز برابر است با تعداد کلمات مصرفی حافظه است.
مثال های دیگر از مدل های محاسباتی:

۳.۳ مدل مقایسه

۴.۳ ماشین اشاره‌گر

۵.۳ ماشین پایتون

یک مدل محاسبات را میتوان به راحتی تعریف کرد. یک کد نوشته شده به زبان پایتون را میتوان در یک مدل محاسبات خاص مدل کرد. هر دستور پایتون در یک یا چند مرحله در مدل محاسباتی انجام میشود. از جمله قابلیت های ماشین پایتون میتوان به موارد زیر اشاره کرد:

۱.۵.۳ list

هر لیست یک آرایه از ابجکت های مختلف در زبان پایتون است. با دستور append میتوان یک عضو را در زمان ثابت به انتهای لیست اضافه کرد.
مرتب سازی یک لیست با n عضو در پایتون در مرتبه زمانی $nlog_2n$ انجام میگیرد.

۲.۵.۳ dictionary

دیکشنری مجموعه‌ای کلیدها(Key) و مقادیر(Value) است که الگوریتم های درهم سازی روی آنها پیاده میشوند.

۳.۵.۳ long

این اصطلاح به اعداد بزرگ نسبت داده میشود که در یک word جا نمیگیرند.

جزوه جلسه سوم داده ساختارها و الگوریتم

۴ مهر ۱۴۰۰

فهرست مطالب

۱ مرتب سازی و کاربرد های آن

مرتب سازی به فرایندی اطلاق میشود که در آن ورودی یک آرایه از اشیا قابل مقایسه به سایز n است و حروجی نیز جایگشتی از همان آرایه است که در عین حال بر اساس یک ویژگی مرتب شده است.

کاربردهای مرتب سازی

۱. آماده سازی برای جست‌وجوی دودویی
۲. آماده سازی برای پیدا کردن عناصر تکراری و تعداد تکرار آنها در آرایه
۳. دستورانی مانند ls در سیستم عامل که با پارامتر های ورودی مختلف مرتب سازی را انجام میدهد
۴. پایگاه داده
۵. مرور اتفاقات رخ داده در یک یا چند فایل(log)
۶. رندر سه بعدی برای انیمیشن سازی
۷. الگوریتم های فشرده سازی فایل
۸. الگوریتم های هندسی مانند Convex Hull

۲ مرتب سازی درجی یا **Insertion Sort**

در این شیوه مرتب سازی نشانگر را از عضو اول تا عضو آخر حرکت میدهیم و در هر مرحله مطمئن میشویم اعضای قبل نشانگر مرتب هستند.
Pseudocode for Insertion Sort:

for i ← 1 to n-1
. insert A[i] into sub-array A[0,...,i-1]
. which is already sorted by pairwise swaps.

۳ تحلیل زمانی

در تحلیل زمانی یک الگوریتم سه حالت را در نظر میگیریم:
۱. بهترین حالت
۲. بدترین حالت
۳. حالت میانگین

برای تحلیل زمانی یک الگوریتم، بدترین حالت را برای تعدادی ورودی خاص در نظر میگیریم که الگوریتم بدترین عملکرد خود را نشان میدهد.
در تحلیل زمانی باید عملکرد الگوریتم را برای داده های بزرگ(nهای بزرگ) را نیز در نظر گرفت.
تابع $T(n)$ را زمان اجرای الگوریتم در بدترین حالت تعریف میکنیم. حال برخی نمادها را معرفی میکنیم. ($f(n)$ یک تابع برحسب n است.)

۱. $\Theta(f(n))$
۲. $O(f(n))$
۳. $o(f(n))$
۴. $\Omega(f(n))$
۵. $\omega(f(n))$

حال به تعریف دقیق علائم ذکر شده میپردازیم:

۱. $if \quad \lim_{n\to\infty} T(n)/f(n) = c \quad (c \in \mathbb{R}) \quad then \quad T(n) = \Theta(f(n))$
۲. $if \quad \lim_{n\to\infty} T(n)/f(n) = c \quad or \quad 0 \quad (c \in \mathbb{R}) \quad then \quad T(n) = O(f(n))$
۳. $if \quad \lim_{n\to\infty} T(n)/f(n) = 0 \quad then \quad T(n) = o(f(n))$
۴. $if \quad \lim_{n\to\infty} T(n)/f(n) = 0 \quad or \quad \infty \quad then \quad T(n) = \Omega(f(n))$
۵. $if \quad \lim_{n\to\infty} T(n)/f(n) = \infty \quad then \quad T(n) = \omega(f(n))$

$\Theta(1)$ نشانگر عدد ثابت است که این عدد میتواند 10^{80}(تعداد کل الکترون های جهان!) باشد.

مثال۱

$$(20n)^7 = 20^7 * n^7 = \Theta(n^7) = O(n^7) = O(n^8) = \Omega(n^7) = \Omega(n^6) = o(n^8) = \omega(n^6)$$

مثال۲

$$5^{log_2 3} n^3 + 10^{80} n^2 + 0.001 n^{3.1} + 402555 = \Theta(n^{3.1}) \neq \Theta(n^4)$$

نکته این است که قادر به تغییر و رند کردن توان ها حتی به صورت جزئی نیستیم و همچنین اگر عبارت لگاریتمی در توان نباشد، مبنای آن در محاسبات تفاوتی ایجاد نمیکند.

مثال۳

$log\binom{n}{n/2} = ???$

برای حل ابتدا با تقریب استرلینگ آشنا میشویم: $n! \approx \sqrt{2\pi n}(n/e)^n$

$$log\binom{n}{n/2} = log(\frac{n!}{(n/2)!(n/2)!}) = log(\frac{\sqrt{2\pi n}(n/e)^n}{(\sqrt{2\pi n/2}(\frac{n/2}{e})^{n/2})^2}) = log(\frac{1}{\sqrt{2\pi}\sqrt{n}/2(1/2)^n}) =$$
$$log(\frac{2^n}{\sqrt{n}} * c) = n - \frac{1}{2}logn = \Theta(n)$$

جزوه جلسه چهارم داده ساختارها و الگوریتم

۶ مهر ۱۴۰۰

فهرست مطالب

۱ مرتبه زمانی الگوریتم Insertion Sort

اگر مرتبه زمانی اجرای الگوریتم مرتب سازی درجی را با $T(n)$ نشان دهیم، میخواهیم ثابت کنیم: $T(n) = \Theta(n^2)$
در ریاضیات برای اثبات برابری a و b میتوانیم نشان دهیم: $a \geq b \quad and \quad b \geq a$
حال معادلا برای اثبات برابری $T(n) = \Theta(n)$ میتوانیم نشان دهیم:
$T(n) = O(n) \quad and \quad T(n) = \Omega(n)$

ابتدا کد مربوط به مرتب سازی درجی:

```
1. for i in ragne (1,n)
2.     while i>0 and A[i] < A[i-1]
3.         A[i], A[i-1] = A[i-1], A[i]
4.         i-=1
```

۱. $T(n) = O(n^2)$
باید نشان دهیم زمان اجرای الگوریتم کمتر یا مساوی n^2 میباشد. خطوط سه و چهار در زمان ثابت $\mathrm{O}(1)$ اجرا میشوند. حلقه خط دوم حداکثر n بار اجرا میشود و همچنین حلقه خط اول نیز حداکثر n بار پیمایش میشود. پس اردر زمانی اجرای برنامه $O(n^2)$ میباشد .

۲. $T(n) = \Omega(n^2)$
این بار نشان میدهیم زمان اجرای الگوریتم بیشتر یا مساوی n^2 است. پس بدترین حالت را در نظر میگیریم؛ یعنی با فرض اینکه میخواهیم آرایه به صورت صعودی مرتب شود، آرایه ای را در نظر میگیریم که به صورت نزولی مرتب شده است. $([n, n-1, ..., 3, 2, 1])$
حال محاسبات را به صورت دقیق انجام میدهیم. فرض کنیم خطوط ۳ و ۴ در c واحد زمانی اجرا میشوند. هر حلقه خط دوم، دقیقا i مرحله انجام میشود. پس هر مرحله از حلقه خط یک ic واحد زمانی طول میکشد. پس کل حلقه $\sum_{n=1}^{n-1} ic$ واحد زمانی به طول می انجامد.
$\sum_{n=1}^{n-1} ic = \binom{n}{2} c = \frac{n(n-1)}{2} c = \frac{n^2}{2} c - \frac{n}{2} c = \Omega(n^2)$

پس از دو بخش فوق نتیجه میشود: $T(n) = \Theta(n)$

۲ مرتب سازی درجی دودویی یا Binary Insertion Sort

در مرتب سازی درجی، جایگاه یک عضو را با swap های متوالی در یک زیر-آرایه مرتب پیدا میکنیم و در آن جایگاه قرار میدهیم. اما در مرتب سازی دودویی، ابتدا با جست‌وجوی

دودویی جایگاه عدد را پیدا میکنیم و با swap های متوالی آنرا در جایگاه خود قرار میدهیم. هنگامی که عمل مقایسه پیچیده شود(مانند مقایسه کردن اشیا مختلف) و هزینه زمانی آن دیگر $O(1)$ نباشد، مرتب سازی باینری بهتر است زیرا تعداد مقایسه ها برای پیدا کردن جایگاه کمتر میشود. به‌طور کلی:
اگر هر مقایسه از مرتبه $O(1)$ باشد:

Insertion Sort: $\Theta(n^2)$
Binary Insertion Sort: $\Theta(n^2)$

اگر هر مقایسه از مرتبه $O(k)$ باشد:

Insertion Sort: $\Theta(n^2k)$
Binary Insertion Sort: $\Theta(nLognk + n^2)$

$nlognk$ مربوط به مقایسه ها و n^2 مربوط به swap هاست.

۳ رویکرد حل مسئله تقسیم و حل یا Divide and Conquer

در این رویکرد حل مسئله، مسئله اصلی به زیر مسئله های کوچکتر تقسیم شده و پس از حل کردن زیر مسئله ها، آنها را باهم ادغام میکنیم. به طور کلی این رویکرد از سه بخش تشکیل شده است:
۱. تقسیم (Divide)
۲. حل (Solve)
۳. ادغام (Merge)
هنگامی که مسئله را با این رویکرد حل میکنیم، مرتبه زمانی آن به صورت زیر محاسبه میشود:
$T(n)$ = زمان تقسیم + زمان حل + زمان ادغام

۴ مرتب سازی ادغامی یا Merge Sort

با استفاده از رویکرد تقسیم و حل میتوان مرتب سازی ادغامی را معرفی کرد. سورس کد مرتب سازی اذغامی به زبان های مختلف در لینک زیر موجود است.
https://www.geeksforgeeks.org/merge-sort/
با توجه به کد، میتوان مرتبه زمانی را برای الگوریتم فوق به صورت زیر تعریف کرد:
۱. زمان تقسیم: $O(1)$
۲. زمان حل: $2T(n/2)$
۳. زمان ادغام: $O(n)$

$$T(n) = O(1) + 2T(n/2) + O(n) = 2T(n/2) + O(n)$$

حال برای پیدا کردن یک رابطه صریح برای $T(n)$ از درخت بازگشتی استفاده میکنیم و

فرض میکنیم $O(n) = cn$
رابطه فوق برای $T(n)$ از دو بخش cn و $2T(n/2)$ تشکیل شده است. پس ریشه درخت cn است و دو خوشه متصل به آن، هریک $T(n/2)$ هستند. به صورت بازگشتی میتوان این مراحل را برای $T(n/2)$، $T(n/4)$ و... انجام داد.
$T(n) = 2T(n/2) + cn = 2(2T(n/4) + cn/2) + cn = ...$
پس ارتفاع درخت برابر $logn$ میباشد و که مجموع برگ ها در هر مرحله برابر cn میباشد.
پس داریم: $T(n) = cn * logn = \Theta(nlogn)$

۵ حل مسئله به روش درخت بازگشتی

در حالت کلی فرض کنیم رابطه مربوط به $T(n)$ به صورت زیر تعریف شده است:
$T(n) = aT(n/b) + f(n)$ f(n) تابعی از n میباشد :
ریشه از مرتبه $f(n)$ میباشد و برگ های متصل به آن به تعداد aتا برگ از مرتبه $f(n/b)$ است. به طور کلی هر برگ از مرتبه $f(n/b^k)$ به تعداد a فرزند از مرتبه $f(n/b^{k+1})$ دارد.
طبق توضیحات فوق واضح است که درخت بازگشتی ذکر شده دارای $L = n^{log_b a}$ برگ و ارتفاع $log_b n$ میباشد.
به کمک قضیه اصلی(Master Theorem) یک درخت بازگشتی در سه حالت قابل حل است.

قضیه اصلی یا Master Theorem

قضیه اصلی با توجه به سه حالت ممکن برای وضعیت مجموع برگ های با یک ارتفاع میتواند رابطه صریح برای $T(n)$ ارائه دهد.
۱. $f(n) = O(L^{1-\epsilon})$: حالتی که در آن، برگ های با بیشترین ارتفاع غالب هستند.
در این حالت داریم: $T(n) = \Theta(L)$
۲. $f(n) = \Theta(L(logn)^k)$: حالتی که در آن، مجموع برگ ها در هر ارتفاعی برابر است.
در این حالت داریم: $T(n) = \Theta(L(logn)^{k+1})$
۳. $f(n) = \Omega(L^{1+\epsilon})$: حالتی که در آن، ریشه غالب است
در این حالت داریم: $T(n) = \Theta(f(n))$
نکته این که قصیه اصلی برای حالتی غیر این سه حالت، جوابی ارائه نمیدهد.

۶ تحلیل زمانی الگوریتم ها برای nهای بزرگ

در تحلیل زمانی الگوریتم های مختلف، علاوه بر مرتبه بزرگی آنها (Θ, Ω, O) باید به ضرایب موجود در رابطه مربوط به $T(n)$ و همچنین رابطه صریح آن نیز توجه کرد. زیرا

برای مقادیری از n رفتار توابع $T(n)$ مربوط به الگوریتم های مختلف تغییر میکند.

مثال

۱. مرتب سازی درجی در زبان C++ در زمان $0.01n^2\mu s$ انجام میشود.
۲. مرتب سازی درجی در زبان Pytho در زمان $0.2n^2\mu s$ انجام میشود.
۳. مرتب سازی ادغامی در زبان Python در زمان $2.2n^2\mu s$ انجام میشود.
در مقایسه الگوریتم های اول و سوم، برای $n \geq 400$ مرتب سازی ادغامی پایتون عملکرد بهتری دارد. همچنین برای $n \geq 67$ استفاده از مرتب سازی ادغامی پایتون بهتر از مرتب سازی درجی در همان زبان است.

جزوه جلسه پنجم داده ساختارها و الگوریتم

۱۱ مهر ۱۴۰۰

فهرست مطالب

۱ مقایسه الگوریتم های مرتب سازی(درجی و ادغامی)

۱. زمان اجرا در بدترین حالت و حالت میانگین:
مرتب سازی ادغامی در زمان $O(nlogn)$ آرایه به طول n را مرتب میکند در حالی که همان آرایه در زمان $O(n^2)$ توسط الگوریتم درجی مرتب میشود.

۲. زمان اجرا در بهترین حالت:
در بهترین حالت (یک آرایه مرتب به عنوان ورودی به الگوریتم داده شود) الگوریتم ادغامی یک آرایه را در زمان $O(nlogn)$ مرتب میکند که این زمان برای الگوریتم درجی برابر $O(n)$ است.

۳. حافظه مصرفی (حافظه اضافی به جز حافظه مخصوص آرایه):
مرتب سازی درجی به صورت in-place انجام میشود و به همین دلیل حافظه اضافی آن برای ایندکس های مخصوص پیمایش آرایه و swap عناصر است. با این تفاسیر حافظه مورد نیاز از مرتبه $O(1)$ است. اما مرتب سازی ادغامی نیازمند یک حافظه اضافی به دلیل تقسیم آرایه به زیر آرایه های کوچک و همچنین ادغام آنهاست. از این رو حافظه مورد نیاز در این الگوریتم $O(n)$ است. توجه شود که حافظه از مرتبه $O(n)$ کافی است و در هر مرحله از الگوریتم که به صرت بازگشتی اجرا میشود نیاز به حافظه جدید نداریم. برای کم کردن حافظه میتوان از الگوریتم های کمکی دیگر مانند External Sort استفاده کرد اما اینکار موجب افزایش زمان اجرای مرتب سازی به اندازه ۲ یا ۳ برابر میشود و حافظه مصرفی از $O(n)$ به $O(1)$ میرسد.

۲ الگوریتم های مرتب سازی مورد استفاده در زبان های برنامه سازی

۱. C/C++
برای داده های پایه(Primitive Data) و اشیا(Objects) از الگوریتم مرتب سازی Quick Sort استفاده میشود.

۲. Python
در این زبان برای هر دوی داده های پایه و اشیا از الگوریتم مرتب سازی Tim Sort استفاده میشود.

۳. Java
این زبان برای مرتب سازی داده های Primitive از الگوریتم Quick Sort و برای Objectها

از الگوریتم Tim Sort استفاده میکند.

۱.۲ Tim Sort

این الگوریتم در سال ۲۰۰۲ توسط Tim Peter در فضای صنعتی و توسط زبان Python پیاده سازی شده است. یک آرایه طی مراحل زیر توسط این الگوریتم که خود ترکیبی از الگوریتم های مرتب سازی درجی دودویی و ادغامی است Sort میشود:

۱. قسمت های صعودی و نزولی در آرایه پیدا میشوند.
۲. قسمت های نزولی برعکس شده و صعودی میشوند.
۳. قسمت های صعودی باهم ادغام میشوند.
۴. اگر مجموع طول ۲ قسمت ادغام شده کمتر از ۶۴ باشد با الگوریتم درجی و در غیر اینصورت با الگوریتم ادغامی مرتب میشوند.

۳ داده ساختارها یا Data Structures

دو مفهوم ابتدایی در مورد ساختمان های داده شرح داده شده است:

۱. Interface/Abstract Data Type: این اصطلاح که با نام واسط نیز شناخته میشود مربوط به مشخصات داده ساختار و ویژگی های آن است؛ یعنی چه داده هایی نگه داری میشوند و چه عملیاتی روی آنها انجام میشود. به عبارت دیگر این اصطلاح همان صورت مسئله است.
۲. Data Structure: این اصطلاح مربوط به پیاده سازی داده ساختار با ویژگی های واسط است. در این قسمت درمورد چگونگی نگه داری داده ها و الگوریتم هایی که روی داده ها پیاده میشوند صحبت میشود. این اصطلاح همان راه حل مسئله است.
هر داده ساختار یه واسط مخصوص به خود را دارد که با توجه به ویژگی های همان واسط، داده ساختار موردنظر پیاده سازی میشود.

اینترفیس های مورد بحث در این درس به دو بخش تقسیم میشوند:
۱. واسط های دنباله ای: این واسط ها برای داده ساختار هایی که ترتیب آنها مهم است تعریف میشود.
۲. واسط های مجموعه ای: این واسط بر خلاف واسط های دنباله ای، برای داده هایی است که ترتیب ذخیره سازی آنها مهم نیست.
واسط های دنباله ای، خود به دو دسته ۱.واسط های دنباله ای ایستا و ۲.واسط های دنباله ای پویا تقسیم میشوند.

۱.۳ واسط های دنباله ای ایستا

این واسط ها برای نگه داری دنباله ای به طول n به کار میرود که عملیات زیر نیز در آن تعریف شده است:
len(): طول دنباله را برمیگرداند که عدد ثابت n است.
seq-iter(): کل دنباله را برمیگرداند
right()/left(): عضو اول/آخر را برمیگرداند
at(i): عنصر iام را برمیگرداند
set-at(i, x): عنصر x را در جایگاه iام قرار میدهد

۲.۳ واسط های دنباله ای پویا

این واسط ها همانند واسط های ایستا هستند با این تفاوت که تعداد اعضای دنباله ای که میخواهیم آنرا ذخیره کنیم فیکس نیست. تمام عملیات موجود برای واسط ایستا برای این واسط نیز تعریف میشود بعلاوه:
insert-at(i, x): عنصر x را در جایگاه iام وارد میکند.
insert-right(x)/insert-left(x): عنصر x را به ابتدا/انتهای دنباله وارد میکند.
delete-at(i)/delete-right()/delete-left(): عنصر اول/آخر/iام را حذف میکند.
تمامی عملیات فوق در یک آرایه به راحتی انجام میشود اما در یک لیست پیوندی بعضی عملیات به سادگی آرایه نیست.

لیست پیوندی: یک لیست پیوندی شامل تعدادی شی است که هر شی به شی بعدی خود در حافظه اشاره میکند. در لیست پیوندی دسترسی به عضو اول یا آخر مانند آرایه در زمان ثابت $O(1)$ انجام میشود اما عملیاتی مانند at(i) در زمان ثابت انجام نمیشوند، زیرا دسترسی مستقیم به آنها موجود نیست و باید از عضو اول یا آخر به آن رسید.

۳.۳ برخی واسط های مهم

۱.۳.۳ واسط **Stack**

اشیا به ترتیب وارد stack میشوند و روی هم قرار گرفته و در هر لحظه تنها به بالاترین عضو دسترسی داریم(Last In First Out). عملیاتی که در این واسط قابل انجام است عبارتند از:

۱. top(): معادل عمل right() در آرایه است و اجازه دسترسی به بالاترین عضو را میدهد.
۲. push(): معادل عمل insert-right() در آرایه است که یک عضو جدید را در بالای استک قرار میدهد.
۳. pop(): معادل عمل delete-right() در آرایه است که بالاترین عضو را از استک حذف میکند.

۲.۳.۳ واسط **Queue** یا صف یکطرفه

عناصر ورودی به ترتیب ورود، از صف خارج میشوند(First In First Out). عملیات تعریف شده در صف یک طرفه:

۱. enqueue(): معادل عمل insert-left() در آرایه است که یک عنصر به صف اضافه میکند.
۲. dequeue(): معادل عمل delete-right() در آرایه است که قدیمی ترین عضو را خارج میکند.

۳.۳.۳ واسط **Deque** یا صف دوطرفه

اشیا از هر دو طرف میتوانند وارد و یا خارج شوند و محدودیتی از این بابت وجود ندارد. عملیاتی که برای Deque تعریف شده اند:(همانند عملیات آرایه ها)

۱. delete-right()
۲. delete-left()
۳. insert-right()
۴. insert-left()

جزوه جلسه ششم داده ساختارها و الگوریتم

۱۱ مهر ۱۴۰۰

فهرست مطالب

۱ پیاده سازی داده ساختار های Stack، Queue و Deque

۱. استفاده از لیست پیوندی:
اینترفیس های ذکر شده برای داده ساختار های بالا به راحتی توسط لیست پیوندی قابل پیاده سازی هستند اما در مواردی مانند دسترسی به عنصر iام داده ها در زمان ثابت $O(1)$ امکان پذیر نیست.
۲. آرایه با سایز متغیر:
تغییر طول آرایه به منزله allocate کردن یک فضای جدید و انتقال آرایه قبلی به آرایه جدید است که این کار هزینه زمانی زیادی دارد. از طرفی سایز یک آرایه به هر میزان نمیتواند بزرگ باشد و مطلوب این است که برای یک آرایه به طول n فضای مصرفی برابر $O(n)$ باشد.
حال شیوه پیاده سازی یک داده ساختار با اینترفیس های ذکر شده (در جلسه قبل) را ذکر میکنیم. داده ساختار های vector در زبان c++، list در python و array list در java اینگونه پیاده سازی شده اند:
- به جای آرایه ای به طول n، آرایه ای به طول $\Theta(n)$ میگیریم.
- هر وقت آرایه پر شد، آرایه جدید به طول ۲ برابر آرایه قبل میگیریم و آرایه قبل را در ابتدای آرایه جدید کپی میکنیم.
پس با تفاسیر فوق اگر سایز آرایه قبل از درج توانی از ۲ باشد، درج عضو جدید در زمان $O(n)$ و درغیر اینصورت در زمان $O(1)$ انجام میشود. برای تحلیل زمانی عملیات درج نیز داریم: (اگر فرض کنیم $O(1) = 1$)

$$1 + 2 + 4 + 1 + 8 + 1 + 1 + 1 + 16 + 1 + 1 + ... \geq 5n = \Theta(n)$$

با تحلیل سرشکن میتوان نتیجه گرفت:

$$\Theta(n)/n = O(1)$$

پس میتوان ادها کرد هزینه درج بصورت سرشکن برابر $O(1)$ است.
- برای اینکه حافظه مصرفی $O(n)$ باقی بماند، وقتی اعضای آرایه به اندازه $1/4$ طول آن شدند، آرایه جدید به طول نصف آرایه میگیریم و اعضا را به ارایه جدید منتقل میکنیم.
بدیهی ایست که اگر مقدار یک چهارم، برابر یک دوم میشد، هنگامی که نصف آرایه پر بود، درج یک عضو و سپس حذف آن به دفعات زیاد، هزینه زمانی بالایی در پی داشت.
مجددا مانند تحلیل سرشکن برای درج میتوان نشان داد هزینه زمانی حذف عضو به صورت سرشکن برابر $O(1)$ میباشد

۲ واسط های مجموعه ای

یک واسط مجموعه ای، مجموعه ای مانند S را نگهداری میکند که هر عضو(شئ) از مجموعه یک کلید نیز دارد.

۱.۲ اینترفیس مجموعه‌ای ساده یا ایستا

۱. find-by-key(key): شی با کلید key را درصورت وجود برمیگرداند.
۲. iter(): اشیا را با تریتبی دلخواه برمیگرداند.

۲.۲ اینترفیس مجموعه‌ای پویا

تمام عملیات واسط مجموعه‌ای ایستا بعلاوه:
۱. insert(key, value): شئ value با کلید key را وارد مجموعه میکند. اگر شئ با کلید key وجود داشت، حذف میشود.
۲. delete-by-key(key): شئ با کلید key را در صورت وجود پاک میکند.

۳.۲ اینترفیس مجموعه‌ای اشیا مرتب

تمام عملیات واسط مجموعه ای ایستا بعلاوه:
۱. find-next(key): شئ x عضو S را با حداقل کلید بزرگتر از key را برمیگرداند.
۲. find-prev(key): شئ x عضو S با حداکثر کلید کوچکتر از key را برمیگرداند.
۳. find-next$(-\infty)$ = find-min()
۴. find-prev(∞) = find-max()
۵. ordered-iter(): مجموعه S را به ترتیب کلیدها برمیگرداند.

۴.۲ اینترفیس مجموعه‌ای پویا و مرتب

تمام عملیات واسط ایستای مرتب و پویا بعلاوه:
۱. delete-min(): حذف شئ x با کوچکترین کلید
۲. delete-max(): حذف شئ x با بزرگترین کلید

۳ واسط صف اولویت یا Priority Queue

در این داده ساختار اشیا به ترتیب اولویت در صف قرار میگیرند و داده با بیشترین اولویت در دسترس است. عملیات تعریف شده در این واسط به شرح زیر هستند:

۱. len(): تعداد اشیا را برمیگرداند.
۲. insert(key, value): شئ value را با کلید key وارد صف میکند.
۳. find-max()
۴. delete-max()

اگر صف اولویت را با آرایه عادی پیاده سازی کنیم، در اینصورت مرتبه زمانی عملیات به ترتیب زمان $O(1)$، $O(1)$(بصورت سرشکن)، $O(n)$ و $O(n)$ است.
اما اگر این پیاده سازی با آرایه مرتب انجام شود به ترتیب داریم: $O(1)$، $O(n)$، $O(1)$ و $O(1)$(بصورت سرشکن).

جزوه جلسه هفتم داده ساختارها و الگوریتم

۲۰ مهر ۱۴۰۰

فهرست مطالب

۱ درخت (Tree)

یک درخت، گرافی است که شامل چندین راس و یال است به نحوی که دور در گراف وجود ندارد و همچنین گراف همبند است. درخت ها به دو گروه تقسیم میشوند:
۱. درخت ریشه دار: درختی که یک راس به عنوان ریشه انتخاب شده و بقیه رئوس نسبت به آن اولویت پیدا میکنند. به رئوس موجود در آخرین سطح، برگ گفته میشود.
۲. درخت بدون ریشه: درختی که تمام رئوس در یک سطح هستند و هیچ راسی نسبت به راس دیگر اولویت ندارد.
در یک درخته ریشه دار، فرزندان یک راس (رئوس متصل به آن راس) میتوانند مرتب یا غیر مرتب باشند. در یک درخت مرتب تمامی عناصر در یک سطح به ترتیب از چپ به راست تکمیل هستند.

۱.۱ برخی تعاریف مهم درخت

زیر درخت یک راس: یک درخت مستقل به ریشه آن راس و سلسله مراتب فرزندانش
ارتفاع یک راس: طول بلندترین مسیر (به سمت پایین) موجود از آن راس به یک برگ
عمق یک راس: طول مسیر آن راس تا ریشه درخت

۲.۱ درخت دودویی

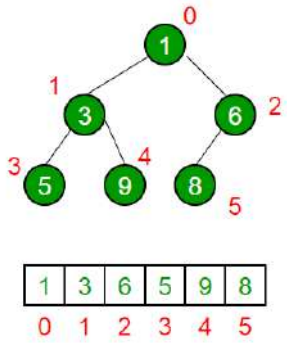
درختی که در آن هر راس حداکثر به دو راس دیگر متصل باشند (حداکثر تعداد فرزندان هر راس برابر دو است)
درخت دودویی کامل: درختی که در آن تمام رئوس سطوح مختلف (به جر سطح آخر یا برگ ها) به صورت مرتب پر شده اند.
درخت دودویی تقریبا کامل: درختی که در آن در سطح آخر، عناصر از ابتدا تا یک جا پر هستند.
حال میتوان هرم دودویی یا Binary Heap را معرفی کرد.

۳.۱ هرم دودویی یا Binary Heap

هرم دودویی داده ساختاری است که از درخت دودویی شکل گرفته به طوری که یک درخت دودویی نسبتا کامل است و بسته به بیشینه یا کمینه بودن هر راس از فرزندان خود بزرگتر یا کوچکتر است. به کمک هرم دودویی میتوان صف اولویت را به نحوی پیدا کرد که مرتبه زمانی عملیات صف اولویت به شکل زیر باشند:

1. len(): $O(1)$
2. insert(k, v): $O(logn)$
3. find _max(): $O(1)$
4. delete _max(): $O(logn)$

برای پیاده سازی صف اولویت با هرم دودویی کافی است اعضای هیپ را مانند شکل زیر در یک آرایه ذخیره سازی کرد.



شکل ۱: هرم دودویی و شیوه ذخیره سازی آن در آرایه

هنگامی که به شیوه بالا ذخیره سازی انجام میشود دسترسی به پدر یک راس و فرزندانش به راحتی انجام میشود:

$\text{parent(i)} = \lfloor \frac{i-1}{2} \rfloor$
$\text{left \_child(i)} = 2i + 1$
$\text{right \_child(i)} = 2i + 2$

هر هرم دودویی، دو خاصیت دارد که میتوان اثبات کرد معادل یکدیگر هستند:

۱. هر راس از رئوس موجود در زیر درخت خود بزرگتر است.
۲. هر راس از فرزندان خود بزرگتر است.

۴.۱ مرتب سازی با صف اولویت

اگر Q یک صف اولویت باشد به کمک آن میتوان یک آرایه را به شکل زیر مرتب کرد:

```
1.  def max_pq _sort(A):
2.      n = len(A)
3.      Q = <>
4.      for v in A:
5.          Q.insert(v)
6.      for i in range(n)
7.          A[n-1-i] = Q.delete _max()
```

در کد فوق nبار عمل insert و سپس nبار عمل delete انجام گرفته. پس مرتبه زمانی آن برابر با $O(nlogn)$ میباشد.

۵.۱ پیاده سازی عملیات تعریف شده برای صف اولویت

۱.۵.۱ **len()**

```
1.  def len():
2.      return len(Q)
```

۲.۵.۱ **find _max()**

```
1.  def find _max():
2.      return Q[0]
```

۳.۵.۱ **insert(Q, v)**

```
1.  def insert(Q, v):
2.      Q.append(v)
3.      max _heapify _up(Q, len(Q) - 1)
4.
5.  def max _heapify _up(Q, i):
6.      if i > 0 and Q[i] > Q[parent(i):
7.          Q[0], Q[parent(i)] = Q[parent(i)], Q[0]
8.      max _heapify _up(Q, parent(i)
```

نکته: تا ارتفاع h از یک هرم دودویی حداکثر $2^{h+1} - 1$ و حداقل 2^h راس وجود دارد. همچنین عمق راس iام برابر است با $\lfloor log((i+1)/n) \rfloor$.

۴.۵.۱ delete _max()

```
1.    def delete _max():
2.       Q[0], Q[len(Q) - 1] = Q[len(Q) - 1], Q[0]
3.       result = Q.pop()
4.       max _heapify _down(Q, 0)
5.       return result
6.
7.    def max _heapify _down(Q, i):
8:       best = max(i, right _child(i), left _child(i))
9:       if best ≠ i:
10:          Q[i], Q[best] = Q[best], Q[i]
11:          max _heapify _down(Q, best)
```

برای حذف یک عنصر دلخواه نیز آنرا با آخرین برگ swap کرده و سپس برای قرار دادن عناصر در جایگاه درستشان از max _heapify _down استفاده میکنیم.

۶.۱ راه های ساخت یک Binary Heap

۱.۶.۱ درج تک تک عناصر

عضو جدید را به عنوان یک برگ وارد میکنیم و با max _heapify _up آنرا در جایگاه خود قرار میدهیم. پس کل عملیات در زمان $O(nlogn)$ انجام میشود.

۲.۶.۱ صدا کردن تابع max _heapify _down برای تمام عناصر

با فرض انجام این عملیات برای یک عنصر و پایین اوردن آن در زمان ثابت c، با جمع زدن زمان برای هر راس در بدترین حالت داریم:

$$n/2 * c * 0 + n/4 * c * 1 + n/8 * c * 2 + ... \approx cn = O(n)$$

اولین جمله مربوط به برگ ها، جمله بعدی مربوط به راس ها با ارتفاع ۱ و ...

جزوه جلسه هشتم داده ساختارها و الگوریتم

۲۵ مهر ۱۴۰۰

فهرست مطالب

۱ درخت دودویی جست و جو یا Binary Search Tree

مبحث را با یک مثال معروف شروع میکنیم:

۱.۱ مسئله رزرو باند فرودگاه

در یک فرودگاه، مسئول برج مراقبت باید عملیات زیر را انجام دهد:
۱. به مدت زمان t قبل و بعد فرود هر هواپیما، نباید هیچ هواپیمایی در باند فرودگاه باشد.
۲. در صورت ارضا شدن مورد اول، هواپیما به لیست رزرو اضافه میشود.
۳. بعد از فرود موفقیت آمیز، هواپیما از لیست انتظار خط میخورد.

عملیات فوق، دقیقا معادل عملیات تعریف شده در واسط مجموعه ای مرتب پویا هستند؛ به نحوی که مورد اول همان find _next(A) و find _prev(A)، مورد دوم معادل insert(A) و مورد سوم نیز delete(A) میباشد.

۲.۱ مقایسه پیچیدگی زمانی مسئله برای داده ساختارهای مختلف

اگر مسئله را با یک آرایه معمولی حل کنیم اردر های زمانی به شکل زیر خواهند بود:
۱. insert(A): $O(1)$
۲. delete(A): $O(n)$
۳. find(A): $O(n)$
۴. find _next(A)/ find _prev(A): $O(n)$

برای پیاده سازی با یک آرایه مرتب، مرتبه های زمانی به شکل زیر تغییر میکنند:
۱. insert(A): $O(n)$
۲. delete(A): $O(n)$
۳. find(A): $O(logn)$
۴. find _next(A)/ find _prev(A): $O(logn)$

در گام آخر اگر مسئله با یک درخت دودویی جست و جو مدل شود، مرتبه های زمانی زیر را خواهیم داشت:
۱. insert(A): $O(logn)$
۲. delete(A): $O(logn)$
۳. find(A): $O(logn)$
۴. find _next(A)/ find _prev(A): $O(logn)$
در اصل، مرتبه زمانی تمامی عملیات به اندازه $O(height())$ یا ارتفاع درخت است و

حداکثر ارتفاع نیز میتواند n-1 باشد ولی به صورت متوازن، به اندازه $logn$ میباشد.
حال به تعریف درخت دودویی جست و جو برگردیم:

۳.۱ تعریف داده ساختار Binary Search Tree

درخت دودویی جست و جو، درختی دودویی است که ترتیب ندارد و کامل نیست. همچنین هر راس آن، بزرگتر مساوی زیر درخت چپ خود و همچنین کوچکتر مساوی زیر درخت راست خود میباشد.
توجه شود که خاصیت فوق را نمیتوان به صورت راسی (هر راس، بزرگتر مساوی فرزند چپ خود و کوچکتر مساوی فرزند راست خود) برای هر راس بیان کرد.

۴.۱ متد های مختلف پیمایش یک درخت دودویی جست و جو

در هر شیوه، از ریشه شروع کرده و به ترتیب ذکر شده شروع به پیمایش میکنیم:

pre-order: ابتدا خود راس، سپس درخت چپ و بعد از آن، درخت راست راس پیمایش میشود.
post-order: در این متد ترتیب پیمایش به ترتیب درخت چپ، درخت راست و خود راس میباشد.
in-order: در هر راسی که هستیم، ابتدا درخت چپ، سپس خود راس و درنهایت درخت سمت چپ پیمایش میشود.
در این متد، درخت به صورت مرتب و صعودی پیمایش میشود و میتوان ادعا کرد برای هر درخت دودویی، آن درخت BST است اگر و تنها اگر پیمایش in-order آن مرتب شده باشد.

۵.۱ تعریف عملیات تعریف شده برای Binary Search Tree

۱.۵.۱ find(node, k)

این عملیات مانند جست و جوی دودویی (Binary Search) میباشد. از ریشه شروع به پیمایش میکنیم. اگر k بزرگتر از ریشه بود جست و جو را در زیر درخت راست و در صورت کوچک بودن در زیر درخت چپ انجام میدهیم.

```
1. def find(node, k):
2.     if node.key == k:
3.         return node
4.     elif k < node.key and node.left != None:
5.         return find(node.left, k)
```

```
6.    elif k > node.key and node.right != None:
7.       return find(node.right, k)
7.    return None
```

همانگونه ک بیان شد، تابع find(node, k) دقیقا عملیات Binary Search را در آرایه in-order انجام میدهد.

۲.۵.۱ **insert(node, k)**

```
1. def insert(node, k):
2.    if k <= node:
3.       if node.left != None:
4.          insert(node.left, k)
5.       else:
6.          node.left = new _node(key = k, parent = node, left = None, right = None)
7.    else:
8.       #Same on right side
```

۳.۵.۱ **delete(node)**

```
1. def find _min(node):
2.    if node.left != None:
3.       return find _min(node.left)
4.    return node
```

```
1. def delete(node):
2.    if node.left != None and node.right != None:
3.       succ = find _min (node.right)
4.       node.key = succ.key
5.       delete(succ)
6.    elif node.left != None:
7.       # replace left child
8.    elif node.right != None:
9.       # replace right child
10.    else:
11.       # replace parent
```

۴.۵.۱ **find _next(node)**

```
1. def if _right _child(node):
2.     return node.parent
```

```
1. def successor(node):
2.     if node.right != None:
3.         return find _min(node.right)
4.     while is _rigth _child(node) != None:
5.         node = node.right
6.     return node.parent
```

کد تابع find _prev(node) نیز مانند کد بالا نوشته میشود.

جزوه جلسه نهم داده ساختارها و الگوریتم

۲۳ مهر ۱۴۰۰

فهرست مطالب

۱ مقدمات

همانگونه که در جلسه قبل گفته شد، اردر زمانی عملیات مختلف برابر $O(height)$ میباشد که ارتفاع میتواند تا n-1 بزرگتر شود. اما هدف در درخت دودویی جست و جو این است که ارتفاع را کمتر کرده و به یک درخت متوازن برسیم که در این حالت ارتفاع به $O(logn)$ میرسد.
به طور کلی، هر درخت با ارتفاع لگاریتمی متوازن است و بالعکس.

۲ درخت دودویی جست و جوی AVL

این ساختار که در سال ۱۹۶۲ معرفی شده است، درختی متوازن است که اصلی ترین ویژگی آن، اختلاف ارتفاع فرزندان هر راس است؛ به این صورت که اختلاف ارتفاع فرزند چپ و راست هر راس، حداکثر یک واحد اختلاف دارد.
هر راس که یک فرزند داشته باشد، فرزند غایب آن none نامیده میشود و ارتفاع آن -1 در نظر گرفته میشود.

۱.۲ ارتفاع درخت AVL

ادعا میکنیم ارتفاع درخت حداکثر برابر $2log_2n$ میباشد. برای اثبات این ادعا N_h را حداقل تعداد رئوس در ارتفاع h مینامیم. اگر ارتفاع فرزند چپ و راست راس مورد نظر را با h_1 و h_2 نشان دهیم برای زوج مرتب (h_1, h_2) داریم:

1. $(h_1, h_2) = (h-1, h-1)$
2. $(h_1, h_2) = (h-1, h-2)$
3. $(h_1, h_2) = (h-2, h-1)$

بدیهی است که در حالت اول نمیتوان به دنبال مینیمم تعداد رئوس گشت. حالت های دوم و سوم معادل هم هستند پس با درنظر گرفتن ارتفاع های $(h-1, h-2)$ داریم:

$$N_h = N_{h-1} + N_{h-2} + 1$$

بدیهی است $N_{h-1} \geq N_{h-2}$ پس:

$$N_h \geq 2N_{h-2} \geq 4N_{h-4} \geq 8N_{h-8} \geq ... \implies N_h \geq 2^{h/2}$$

میدانیم:

$$n \geq N_h \implies n \geq 2^{h/2} \implies log_2 n \geq h/2 \implies h \leq 2log_2 n$$

پس ارتفاع درخت متوارن AVL حداکثر برابر $2log_2n$ میباشد.

۱.۱.۲ محاسبه دقیق تر ارتفاع درخت!

اگر جمله iام دنباله فیبوناتچی را با f_i نشان دهیم، ادعا میکنیم:

$$N_h = f_{n+3} - 1$$

درستی رابطه فوق را نیز میتوان تحقیق کرد:

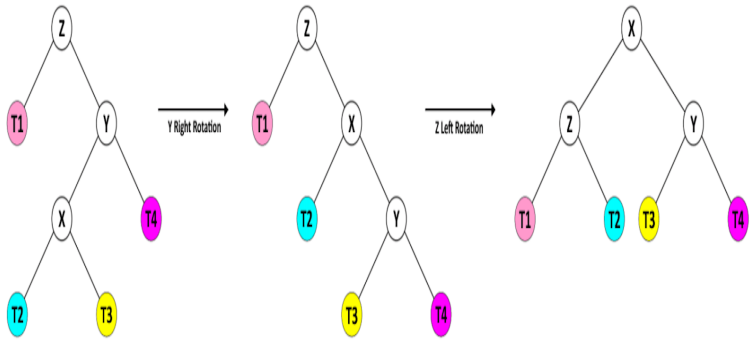
$$N_h = N_{h-1} + N_{h-2} + 1 = (f_{n+2} - 1) + (f_{n+1} - 1) + 1 = f_{n+3} - 1$$

با انجام محاسبات، میتوان کران بالای بهتری برای ارتفاع درخت ارائه داد:

$$h \leq 1.440 * log_2(n+1)$$

۲.۲ متوازن نگه داشتن درخت AVL

بعد از عملیاتی مانند درج و حذف یک کلید، ممکن است ارتفاع بعضی رئوس تغییر کرده و ویژگی درخت AVL یعنی اختلاف ارتفاع فرزندان چپ و راست برقرار نباشد. در این حالت، بسته به حالت درخت، با یک یا چند دوران میتوان شرط ارتفاع هارا برقرار کرد. دو نوع دوران وجود دارد که در شکل زیر هردوی آنها آمده است.



شکل ۱: از چپ به راست: دوران راست روی راس Y، دوران چپ روی راس Z

۳.۲ عملیات تعریف شده برای درخت AVL

کد مربوط به find _next() و find _prev() مانند یک درخت دودویی جست و جو معمولی پیاده سازی میشود.
اما برای عملیات insert() و delete() نیاز است بعد از درج یا حذف یک راس، در صورت لزوم با انجام دوران ارتفاع رئوس به هم ریخته را درست کنیم. برای این کار، از پایین (برگ ها)، اولین راسی را درنظر میگیریم که ارتفاع آن به هم ریخته است(X). پس بعد از انجام عملیات درج یا حذف، اگر ارتفاعش را k+2 در نظر بگیریم، ارتفاع فرزند راست را k+1 و ارتفاع فرزند چپ را k-1 فرض میکنیم.
حال اگر فرزند راست (Y) را درنظر بگیریم، برای ارتفاع فرزند راست(h_1) و فرزند چپ(h_2) آن سه حالت میتوان متصور شد:

1. $(h_1, h_2) = (k, k)$

این حالت بعد از درج یک راس ناممکن است و تنها بعد از حذف یک راس میتواند اتفاق بیافتد.

2. $(h_1, h_2) = (k, k-1)$

در این حالت، با یک دوران چپ گرد روی راس Y میتوان مشکل را حل کرد و بعد از دوران نیز ارتفاع چپ و راست باهم برابر میشوند. در این حالت، بعد از انجام دوران، نیاز به بررسی رئوس بالاتر نیست.

3. $(h_1, h_2) = (k-1, k)$

در این حالت، ابتدا یک دوران راست گرد روی Y انجام میدهیم و سپس، یک دوران چپ گرد روی X انجام میدهیم.
بعد این مراحل، مجدد به بالا حرکت میکنیم و اگر راس دیگری وجود داشت که ارتفاع آن به هم ریخته بود، عملیات بالا را با توجه به ارتفاع چپ و راست آن انجام میدهیم.

۴.۲ مرتب سازی با درخت AVL

میدانیم in-order درخت AVL مرتب شده است. با توجه به این نکته میتوان کد زیر را برای مرتب سازی ارائه داد:

```
1.  def AVl_sort(A):
2.      T = AVL()
3.      for i in A:
4.          T.insert(i)
5.      return in _order _traversal(T)
```

جزوه جلسه ده داده ساختارها و الگوریتم

۴ آبان ۱۴۰۰

فهرست مطالب

۱ درخت جست و جوی متوازن قرمز و سیاه (Red-Black) (Balanced Binary Search Tree)

هر درخت قرمز و سیاه ۴ ویژگی اصلی دارد:
۱. هر راس به یک رنگ قرمز یا سیاه میباشد.
۲. معمولا ریشه و برگ ها به رنگ سیاه هستند.
۳. اگر راسی قرمز باشد، والدش حتما سیاه است.
۴. هر مسیری از ریشه به برگ ها، از تعدادی مشخصی راس سیاه میگذرد که به آن تعداد، سیاه ارتفاع درخت (Black Height) میگویند.
هر راس در این ساختار، اگر دو فرزند نداشته باشد، برای تکمیل درخت راس NIL به عنوان فرزند میپذیرد تا تعداد فرزندانش ۲تا شود. رئوس NIL به رنگ سیاه هستند ولی در شمارش سیاه ارتفاع محاسبه نمیشوند. همچنین این رئوس در محاسبه ارتفاع درخت، درنظر گرفته نمیشوند.
نکته دیگر در مقایسه درخت RB و AVL حافظه اضافی مورد نیاز برای ذخیره سازی اطلاعات میباشد. در درخت های AVL هر راس علاوه بر کلید، ارتفاع خود را نیز ذخیره میکرد که این عدد را میتوان یک int در نظر گرفت. اما در درخت RB هر راس، تنها نیاز داره رنگ خود را در یک بیت ذخیره کند که این مورد یک نقطه قوت برای BR درنظر گرفته میشود.

۱.۱ ارتفاع درخت Red-Black

اگر ارتفاه درخت را با h نشان دهیم، ادعا میکنیم:

$h \leq 2log(n+1)$
برای اثبات ابتدا رئوس قرمز را در رئوس سیاه درج میکنیم؛ بدین صورت که کلید رئوس قرمز را در والد سیاهشان درج میکنیم. در اینصورت،، هر راس سیاه یک، دو یا سه کلید را شامل میشود. در درخت جدید به وجود آمده که تمام رئوس سیاه هستند، هر راس، دو یا سه یا چهار فرزند دارد. ارتفاع این درخت برابر سیاه ارتفاع درخت اولیه میباشد.
حال اگر سیاه ارتفاع را با bh نشان دهیم، بنابر ویژگی سوم درخت قرمز سیاه داریم:

$bh \geq h/2$
میدانیم در درخت جدید حداقل تعداد فرزندان ۲تاست، پس:

$n+1 \geq 2^{bh} \implies bh \leq log(n+1)$
$bh \leq log(n+1)$ and $h/2 \leq bh \implies h \leq 2log(n+1)$

۲.۱ عملیات تعریف شده برای RB Tree

عملیات find _next() و find _prev() به سادگی و مانند درخت AVL انجام میشود.
برای عملیات درج و حذف نیز باید بعد از درج و حذف، رنگ رئوس تغییر کند. توجه شود که عملیات حذف به دلیل پیچیدگی بیش از حد، بررسی نمیشود.

۱.۲.۱ insert(key, value)

ابتدا راس را درج میکنیم و سپس رنگ آنرا قرمز میکنیم. دراینصورت لزوما ویژگی سوم برقرار نیست؛ برای حل این مشکل به رنگ والد و عموی راس درج شده نگاه میکنیم. اگر هردو قرمز باشند، رنگ هردو را با رنگ پدر بزرگ عوض میکنیم. اگر مشکل رنگ ها حل نشده بود، همین کار را برای رئوس پدر بزرگ به بالا مجدد انجام میدهیم.
اگر به مرحله ای رسیدیم که نتوانستیم رنگ هارا تغییر دهیم، از دوران استفاده میکنیم. منطق دروان و تغییر رنگ، همانند منطق دوران و تغییر کلید در درخت AVL میباشد.

۳.۱ نکاتی کلی درمورد درخت های RB و AVL

۱. درخت های RB میتوانند ارتفاع بیشتری نسبت به درخت های AVL داشته باشند اما همانطور که در ابتدا ذکر شد، حافظه کمتری مصرف میکنند.
۲. هرچه ارتفاع درخت کمتر باشد، عملیات جست و جو سریعتر و درج کند تر انجام میشود و بالعکس
۳. in-order درخت قرمز و سیاه مانند درخت AVL مرتب شده میباشد.

جزوه جلسه یازدهم داده ساختارها و الگوریتم

۹ آبان ۱۴۰۰

فهرست مطالب

۱ مرتب سازی سریع یا Quick Sort

این شیوه مرتب سازی در سال ۱۹۶۲ معرفی شد و هم اکنون، در زبان های Java و C++ به عنوان الگوریتم مرتب سازی پیش فرض استفاده میشود.
حافظه مورد نیاز برای این مرتب سازی، به اندازه $O(1)$ برای ذخیره سازی آرایه اصلی و حافظه‌ای از مرتبه $logn$ برای اجرای الگوریتم به شیوه بازگشتی در stack میباشد.
این مرتب سازی اگر به شیوه درست پیاده سازی شود، به اندازه ۲ برابر سریع تر از مرتب سازی ادغامی (Merge Sort) است. (لازم به ذکر است مرتبه زمانی این الگوریتم برابر $O(nlogn)$ میباشد.)

۱.۱ مقایسه Quick Sort با Merge Sort

مرتب سازی سریع نیز مانند مرتب سازی ادغامی، بر اساس الگوریتم تقسیم و حل (Divide and Conquer) کار میکند. در مرتب سازی سریع، تمرکز اصلی روی مرحله تقسیم و در مرتب سازی ادغامی روی ادغام است. مرحله حل نیز در هر دو به دلیل کوچک شدن زیر مسئله ها، بسیار بدیهی و ساده است.
توجه شود چون میتوان Quick Sort را به صورت in-place پیاده سازی کرد، پس نیازی به مرحله ادغام در این متذ نیست و در هر مرحله از الگوریتم، آرایه مرتب میشود و نیازی به ادغام زیر ارایه ها نیست.

۱.۱.۱ مقایسه کوچک بین Quick Sort و Tim Sort!

مرتب سازی Tim سریعتر از مرتب سازی سریع است (ضریب کوچکتری از $nlogn$ میباشد) و همچنین برای آرایه هایی که تقریبا-مرتب هستند، استفاده از Tim Sort بسیار بهتر است؛ زیرا این متد مرتب سازی به initial state آرایه بستگی دارد.

۲.۱ مراحل مرتب سازی Quick Sort

۰. انتخاب عنصر لولا (pivot) به نام x از آرایه ورودی
انتخاب pivot میتواند خروجی یک الگوریتم خاص باشد یا یک عنصر تصادفی از آرایه (مانند عضو اول یا آخر یا وسط)

۱. تقسیم آرایه ورودی به سه قسمت L: اعضای کوچکتر از x، E: اعضایی که برابر با x هستند، G: عناصری که از x بزرگتر هستند.

۲. مرتب سازی L و G به صورت بازگشتی
مرحله تقسیم تا جایی پیش میرود که L و G حداکثر یک عضو داشته باشند که در این

حالت مسئله حل شده میشود.

۳. ادغام
همانگونه که ذکر شد، این مرحله نیاز نیست.
مرحله یک، با یک پیمایش ارایه انجام میشود و ارایه به سه زیر آرایه تقسیم میشود (این قسمت میتواند به صورت in-place نیز پیاده سازی شود و حافظه اضافی نگیرد)
اما نکته اصلی این الگوریتم، انتخاب عنصر لولا میباشد. بسته به اینکه این عنصر چگونه انتخاب مبشود، نسخه های مختاف Quick Sort معرفی میشود:

۳.۱ نسخه های مختلف مرتب سازی سریع

۱.۳.۱ مرتب سازی سریع پایه‌ای

عنصر اول آرایه به عنوان pivot درنظر گرفته میشود. در بدترین حالت (یک آرایه مرتب شده به صورت صعودی یا نزولی) زمان اجرا به $O(n^2)$ میرسد.
برای بهبود عملکرد این حالت، میتوان از درهم سازی آرایه در ابتدا و قبل از انتخاب pivot بهره برد. با انجام این کار، دیگر نمیتوان یک حالت را به عنوان بدترین حالت درنظر گرفت و اردر زمانی بدست آمده برای حالت میانگین یا Average Case میباشد.

۲.۳.۱ مرتب سازی سریع با انتخاب هوشمندانه لولا

در این نسخه، با یک الگوریتم خطی ($\Theta(n)$)، میانه عناصر در ارایه پیدا میشود و به عنوان لولا انتخاب میشود.
در بدترین حالت، مرتبه زمانی برابر میشود با:
$T(n) = \Theta(n) + 2T(n/2)$
که طبق Master Theory داریم:
$T(n) = \Theta(nlogn)$
اما نکته قابل توجه، ضریب بالا در این شیوه پیاده سازی است. ضریب $nlogn$ در این حالت ۲ برابر مرتب سازی ادغامی است و نتیجتا سرعت آن نیز نصف سرعت Merge Sort میشود.
الگوریتمی که برای پیدا کردن میانه استفاده میشود، میانه میانه ها یا Median of Medians نام دارد.

۳.۳.۱ مراحل الگوریتم میانه میانه ها

این الگوریتم به صورت کلی پیاده سازی میشود و میتواند برای هر k، kامین عضو آرایه را برگرداند.

۱. آرایه ورودی(A) را به ستون های حداکثر ۵ عضوی افراز میکنیم. (زمان: $\Theta(1)$)

۲. هر ستون ۵تایی را مرتب میکنیم (زمان: $\Theta(n)$)

۳. میانه سطر وسط ستون هارا با همین الگوریتم پیدا میکنیم و آنرا x مینامیم (زمان: $T(n/5)$)
عنصر x پیدا شده حتما از سی درصد عناصر بزرگتر و از سی درصد عناصر کوچکتر است.

۴. عناصر A را به سه مجموعه L، E ، G تقسیم میکنیم (زمان: $\Theta(n)$)

۵. بر اساس مقدار k، در یکی از سه مجموعه به صورت بازگشتی به دنبال عنصر مورد نظر میگردیم (زمان: $T(7n/10)$)
بدیهی است در هر مرحله از مسیر بازگشتی، k مقدار جدیدی میگیرد؛ چون اندازه زیر ارایه ها تغییر میکند.
همچنین ضریب ۷/۱۰ در مرتبه زمانی به این دلیل است که حداکثر ۷۰ درصد اعضا میتوانند در یک مجموعه که در آن جست و جو را انجام میدهیم قرار داشته باشند.

همانگونه که ذکر شد، حداقل سی درصد و حداکثر هفتاد درصد اعضا در L و G قرار دارند. با محاسبه دقیق میتوان به این نتیجه رسید که حداکثر تفاضل تعداد اعضای L و G برابر ۶ میباشد.

۴.۳.۱ زمان اجرای الگوریتم

$$T(n) = \Theta(n) + T(n/5) + T(7n/10)$$

از طرفی داریم:

$$T(7n/10) + T(n/5) = T(9n/10) < T(n)$$

بنابر قضیه اصلی، به دلیل کوچک شده برگ ها نسبت به ریشه، مرتب زمانی کل، برابر مرتبه زمانی ریشه میشود.

$$T(n) = \Theta(n)$$

درنظر داشته باشیم که ضریب n در این حالت بزرگ است.

۵.۳.۱ مرتب سازی سریع تصادفی

قبل از توضیح این بخش، کمی درباره خود الگوریتم های تصادفی صحبت میکنیم

۶.۳.۱ الگوریتم های تصادفی

الگوریتم هایی که یک عدد تصادفی r بین ۱ تا R تولید کرده و بر اساس این مقدار، الگوریتم اجرا میشود. برای انتخاب تعداد بیشتری عدد تصادفی، باید عدد R را بسیار بزرگ در نظر گرفت.
ترجیح بر این است که از اعداد و الگوریتم ها تصادفی زیاد استفاده نکنیم.
بنابراین اجراهای مختلف الگوریتم روی ورودی های یکسان میتواند متفاوت باشد. این تفاوت یا در تعداد مراحل اجرا (کند یا سریع بودن الگوریتم) یا در خروجی (خروجی های صحیح یا بد و یا گاها اشتباه) نمود پیدا میکند. (باید احتمال تولید خروجی اشتباه در الگوریتم تصادفی بسیار کم باشد).
الگوریتم های تصادفی به سه دسته تقسیم میشوند:

۱. الگوریتم های مونت کارلو یا احتمالا درست: درستی خروجی در این الگوریتم ها احتمالی است اما همیشه سریع هستند؛ به دیگر بیان، زمان اجرا کم است ولی خروجی نامعلوم است.
مانند الگوریتم بررسی اول بودن یک عدد، بررسی درست بودن ضرب ماتریس ها، شبیه سازی فرایند های فیزیکی، 3D Rendering

۲. الگوریتم های لاس وگاس یا احتمالا سریع: برخلاف الگوریتم های مونت کارلو، خروجی الگوریتم همیشه درست است ولی سریع بودن آن احتمالی است و معلوم نیست.
مانند مرتب سازی سریع تصادفی

۳. الگوریتم های اتلانتیک یا احتمالا درست و احتمالا صحیح: در این الگوریتم ها درستی و سرعت اجرای الگوریتم هردو احتمالی و نامعلوم هستند

جزوه جلسه دوازدهم داده ساختارها و الگوریتم

۱۱ آبان ۱۴۰۰

فهرست مطالب

۱ مرتب سازی سریع تصادفی

در این نسخه، عنصر لولا در هر مرحله، به صورت تصادفی و با احتمال مساوی بین عناصر آرایه(A) انتخاب میشود.
به دیگر بیان، عدد r بین ۰ و $\mathrm{len(A)}$ - 1 انتخاب شده و $\mathrm{A[r]}$ به عنوان عنصر لولا انتخاب میگردد. این کار در عین سادگی بسیار کارامد نیز هست. همچنین فرض میشود انتخاب عدد رندوم در $O(1)$ انجام میگیرد.

۱.۱ مرتبه زمانی مرتب سازی سریع تصادفی

ادعا میکنیم امید ریاضی (میانگین) زمان اجرا برای ورودی های مختلف برابر $O(nlogn)$ است.
حالت خاص این الگوریتم یعنی مرتب سازی سریع تصادفی وسواسی را در نظر گرفته و ادعای خود را برای آن اثبات میکنیم. در این الگوریتم، بعد از انتخاب عنصر لولا و تشکلی L و G، اگر اندازه هرکدام از آنها حداقل $\mathrm{len(A)}/4$ نبود، لولای دیگری را انتخاب میکنیم.
پس اگر لولا در آرایه مرتب شده A در ۲ قسمت میانی باشد (بعد از تقسیم آرایه به ۴ قسمت) الگوریتم ادامه پیدا میکند و درغیر اینصورت لولای دیگری انتخاب میشود. با این تفاسیر، احتمال انخاب لولای مناسب برابر $1/2$ است و در نتیجه، امید ریاضی تعداد تکرار انتخاب لولا با احتمال $1/2$ درستی لولا، طبق توزیع هندسی برابر ۲ است.
طبق توضیحات فوق داریم:

$$\mathrm{T(n)} = \#\mathrm{iterations}.\Theta(n) + \mathrm{T(n/4)} + \mathrm{T(3n/4)}$$

با اعمال امید ریاضی نیز داریم:

$$\mathrm{E(T(n))} = \mathrm{E}(\#\mathrm{iterations}).\Theta(n) + \mathrm{E(T(n/4))} + \mathrm{E(T(3n/4))}$$

میدانیم:

$$\mathrm{E}(\#\mathrm{iteration}) = 2$$

پس در نهایت مرتبه زمانی در بدترین حالت برابر است با:

$$\mathrm{E(T(n))} = \Theta(nlogn)$$

۲ کران پایین الگوریتم های مختلف مرتب سازی

برای پیدا کردن کران پایین مرتب سازی، نیاز به یک مدل محاسباتی داریم؛ زیرا از جلسات اول میدانیم هر الگوریتم در یک مدل محاسبات اجرا میشود.
مدل محاسبه خود را، مدل مقایسه انتخاب میکنیم. در این مدل، عناصر واسط هایی

هستند که تنها میتوانیم آنها را باهم مقایسه کنیم و دسترسی مستقیم به مقادیر آنها نداریم.
زمان اجرای الگوریتم در این مدل، تعداد کل مقایسه هاست. همچنین از عملیات دیگر به جز مقایسه برای محاسبه کران پایین صرف نظر میکنیم؛ هرچند این صرف نظر برای اثبات ما مشکلی ایجاد نمیکند.
تمام مرتب سازی هایی که تا به حال دیدیم به جز الگوریتم Binary Insertion Sort با این مدل محاسباتی سازگار هستند (تنها از مقایسه عناصر برای مرتب سازی استفاده میشود)

۱.۲ درخت تصمیم

این درخت، با درخت هایی قبلی که خواندیم متفاوت است و یک مفهوم انتزاعی است و برای اثبات های ریاضیاتی کاربرد دارد. در عمل، درختی رسم نمیکنیم. همچنین درخت تصمیم منحصر به مدل مقایسه نیست و در دیگر مدل های محاسباتی نیز وجود دارد.
برای هر الگوریتم مبتی بر مقایسه با اندازه ورودی فیکس n، یک درخت تصمیم معادل آن نسبت داده میشود. در این درخت تمام مقایسه های ممکن برای عناصر مختلف و نتایج ممکن برای هر مقایسه بصورت یکجا به تصویر کشیده میشود. رئوس داخلی مقایسه ها و برگ ها نتایج ممکن هستند.

۱.۱.۲ مقایسه درخت تصمیم و الگوریتم در مدل مقایسه

۱. هر راس داخلی در درخت ≡ یک تصمیم دودویی در الگوریتم
۲. برگ های درخت ≡ یک جواب پیدا شده برای الگوریتم
۳. هر مسیر از ریشه به برگ در درخت ≡ یک بار اجرای الگوریتم
۴. طول مسیر از ریشه به برگ در درخت ≡ زمان یک بار اجرای الگوریتم (به ازای یک ورودی خاص)
۵. ارتفاع درخت ≡ طول بزرگترین مسیر از ریشه به برگ ≡ زمان اجرای الگوریتم در بدترین حالت

۲.۱.۲ مسئله کران پایین جست و جو

با درخت تصمیم، میخواهیم اثبات کنیم برای n عنصر پیش پردازش شده، پیدا کردن یک عنصر در مدل مقایسه، حداکثر به زمان $O(nlogn)$ نیاز دارد. دو مورد را در نظر داریم:
۱. درخت تصمیم، یک درخت باینری است
۲. درخت تصمیم متناظر با مسئله حداقل n برگ دارد. این تعداد ممکن است بیشتر هم بشود زیرا ممکن است عنصری که دنبال آن میگردیم در آرایه تکرار شده باشد.
با توجه به دو بند بالا، ارتفاع درخت برابر $\Omega(logn)$ است که معادل زمان اجرای الگوریتم

در بدترین حالت است.
در نتیجه، جست و جوی دودویی روی آرایه و درخت های دودویی جست و جوی متوازن (AVL و R-B) بهینه هستند.

۲.۲ کران پایین مرتب سازی

به مانند مسئله قبل، دو بند زیر را داریم:
۱. درخت تصمیم تشکیل شده، دودویی است.
۲. حداقل تعداد برگ ها برابر $n!$ است. زیرا برای هر آرایه ورودی، باید تمام حالت های ممکن را درنظر بگیریم.
پس ارتفاع درخت حداکثر $logn!$ ($\Omega(logn!)$) میباشد.
در ادامه ثابت میکنیم:

$$\Theta(logn!) = \Theta(nlogn)$$

این کار را به دو روش انجام میدهیم:

۱. تبدیل ضرب به جمع در لگاریتم

$$log(n!) = \sum_{i=1}^{n} logi > (n/2)log(n/2) = \Omega(nlogn)$$

۲. تقریب استرلینگ

$$log(n!) = log(\sqrt{2\pi n}(n/e)^n) = nlogn - nlog_e n + O(logn) = \Theta(nlogn)$$

در نهایت میتوان ادعا کرد مقادیر $nlogn$ و $log(n!)$ بصورت اردری باهم برابرند و اختلاف اندکی دارند.
از تقریب استرلینگ نیز میتوان نتیجه گرفت: $O(n!) = O(n^2)$
همچنین طبق درخت تصمیم میتوان ادعا کرد با تعداد تقریبی $log(n!)$ مقایسه میتوان آرایه را مرتب کرد.
درنهایت با توجه به نتیجه ای که از کران پایین مرتب سازی گرفتیم، میتوانیم ادعا کنیم الگوریتم های مرتب سازی ادغامی، هرمی، AVL، R-B و سریع با انتخاب هوشمندانه لولا (به صورت مجانبی و اردری در مدل مقایسه) بهینه هستند.

جزوه جلسه سیزدهم داده ساختارها و الگوریتم

۱۶ ابان ۱۴۰۰

فهرست مطالب

۱ الگوریتم های مرتب سازی خطی

در این بخش از درس، به مطالعه الگوریتم هایی میپردازیم که زمان اجرای آنها کمتر از $O(nlogn)$ میباشد و همچنین در مدل مقایسه (مدل محاسباتی که در جلسه قبل ذکر شد) نیستند. توجه داریم که $logn$ خود مقداری بسیار کوچک در مقایسه با n میباشد و پیدا کردن چنین الگوریتم هایی صرفا از منظر تئوری ارزشمند هستند.
فرض اضافه ای که میکنیم، این است که کلید تمام عناصری که با هم مقایسه میشوند مقداری صحیح و نامنفی است (در مدل مقایسه این فرض را نداشتیم). این فرض معقول است؛ زیرا کلید هایی که با آنها سروکار داریم همواره عدد نیستند و میتوان برای آنها، معادل عددی (یک عدد صحیح نامنفی) درنظر گرفت.
حال فرض میکنیم کلید ها، اعداد صحیح بین 0 تا k-1 هستند.(حالتی که کلید ها منفی هستند نیز معادل همین فرض است). به بیان دیگر برای n شئ (معادلا n کلید) داریم: $k = O(n)$. یعنی کران بالا برای شماره کلید ها، برابر تعداد کلیدها منهای یک است. همچنین فرض میشود هر کلید در یک کلمه (word) جا میشود.

۱.۱ مرتب سازی شمارشی یا Counting Sort

در این الگوریتم برای هر کلید بین 0 تا k-1، یک لیست درنظر میگیریم و هر عنصر را به لیست مربوط به خود اضافه میکنیم. درنهایت با یک پیمایش روی لیست اصلی، محتوای هر لیست که عناصر ما هستند را خروجی میدهیم. کد پایتون زیر شهود بهتری از این الگوریتم ارائه میدهد.

```
1.  def counting _sort (A):
2.      u = 1 + max([x.key for x in A])    #O(n)
3.      D = [[] for i in range(u)]          # O(u)
4.      for x in A:                         # O(n)
5.          D[x.key].append()
6.      i = 0
7.      for chain in D:                     # O(u)
8.          for x in chain:
9.              A[i] = x
10.             i += 1
```

خطوط ده و یازده در مجموع n بار و حلقه خط هفت نیز u بار اجرا میشوند؛ پس میتوان گفت مرتبه زمانی اجرای حلقه خط هفت برابر $O(n+u)$ است.
مرتبه زمانی اجرای الگوریتم برابر $O(n+k)$ (در کد بالا $O(n+u)$) میباشد.
مرتب سازی شمارشی برای k های کوچک قابل استفاده است که این نکته از ضعف های این الگوریتم است. اما به دلیل پایه ای برای الگوریتم های دیگر است؛ ارزشمند و مهم برای یادگیری است.

۲.۱ مرتب سازی مبنایی یا Radix Sort

اساس کار این مرتب سازی، مرتب سازی شمارشی است و عیب آن (کار کردن برای k های کوچک) را مرتفع نموده است و این الگوریتم برای زمانی که k یک چندجمله ای از n باشد ($k = n^{O(1)}$) در زمان خطی کار میکند.
در این الگوریتم فرض میکنیم کلیدها در مبنای b هستند.
d را نیز تعداد ارقام k در مبنای b درنظر میگیریم. ($b^d \simeq k \iff d = log_b k$)
در مرتب سازی از رقم کم ارزش کلید ها شروع به مرتب سازی کرده و به رقم پرارزش میرسیم. هر مرحله از مرتب سازی توسط مرتب سازی شمارشی انجام میشود. نکته این که از هر مرتب سازی پایدار (مرتب سازی که ترتیب عناصر با کلید یکسان قبل و بعد از مرتب سازی یکسان باشد) میتوان به جای counting sort استفاده کرد.
پس با تفاسیر فوق، میتوان زمان اجرای الگوریتم مرتب سازی پایه ای را ارائه کرد. به تعداد d بار مرتب سازی شمارشی اجرا میشود که در این الگوریتم زمان اجرای مرتب سازی شمارش برابر $O(n + b)$ است (هرکلید بین 0 تا b-1 قرار دارد).
پس مرتبه زمانی این الگوریتم برابر است با $O(d(n + b))$.
زمان اجرا در حالت کلی خطی نیست و برای خطی شدن کافی است d از مرتبه $O(1)$ باشد؛ بدین منظور b را برابر n قرار میدهیم و داریم:

$$d = log_b n^{O(1)} = O(1) \implies O((n + b)d) = O((n + n).O(1)) = O(n)$$

کد پایتون مربوط به Radix Sort نیز در شکل زیر آمده است.

```
def radix_sort(A):
    "Sort A assuming items have non-negative keys"
    n = len(A)
    u = 1 + max([x.key for x in A])            # O(n) find maximum key
    c = 1 + (u.bit_length() // n.bit_length())
    class Obj: pass
    D = [Obj() for a in A]
    for i in range(n):                         # O(nc) make digit tuples
        D[i].digits = []
        D[i].item = A[i]
        high = A[i].key
        for j in range(c):                     # O(c) make digit tuple
            high, low = divmod(high, n)
            D[i].digits.append(low)
    for i in range(c):                         # O(nc) sort each digit
        for j in range(n):                     # O(n) assign key i to tuples
            D[j].key = D[j].digits[i]
        counting_sort(D)                       # O(n) sort on digit i
    for i in range(n):                         # O(n) output to A
        A[i] = D[i].item
```

۳.۱ نکاتی چند در مورد مرتب سازی های خطی

از سوالات حل نشده در مورد مرتب سازی مبنایی میتوان به مرتب سازی برای k های بزرگتر از $n^{O(1)}$ در زمان خطی اشاره کرد.
همچنین در مدل word RAM سریعترین الگوریتم مرتب سازی در زمان $O(n\sqrt{loglogn})$ و با احتمال بالا (مرتب سازی Integer Sort که یک مرتب سازی تصادفی است) میباشد.

جزوه جلسه چهاردهم داده ساختارها و الگوریتم

۱۸ آبان ۱۴۰۰

فهرست مطالب

۱ درهم سازی یا Hash

جدول درهم سازی یکی از پرکاربردترین داده ساختار ها در علوم و مهندسی کامپیوتر است. این داده ساختار در پایتون به نام Dictionary و در جاوا به نام Hash Map شناخته میشود. به دلیل پرکاربرد بودن جدول درهم سازی، خود مسئله درهم سازی نیز پرکاربرد و مهم است.

۱.۱ کاربرد های درهم سازی

۱. زبان های برنامه نویسی: هم در نوشتن کامپایلر (برای مثال keyword های یک زبان را نمیتوان به عنوان اسم متغیر انتخاب کرد و این محدودیت توسط جدول درهم سازی اعمال میشود) و هم به عنوان داده ساختار آماده در خود زبان

۲. پایگاه های داده یا DataBase: در دیتابیس های جدید مانند SQL که داده ها به صورت جدولی ذخیره سازی میشوند، با در دست داشتن یک کلید میتوان به داده های متناطر با شئ آن کلید دسترسی پیدا کرد

۳. مسیریابی شبکه توسط router: مسیر یابی برای یک IP خاص توسط الگوریتم های درهم سازی انجام میگیرد

۴. سرورهای شبکه: عملیات اختصاص port برای یک برنامه توسط جدول درهم سازی انجام میگیرد

۵. حافظه مجازی: هنگام پر شدن حافظه RAM، قسمت هایی از آن رو Hard Drive نوشته میشوند که این عملیات به کمک درهم سازی انجام میگیرد

۶. جست و جوی رشته و زیر رشته در سرویس هایی مانند Google و Grep

۷. پیدا کردن تشابه DNA

۸. همگام سازی فایل ها و شاخه ها (Branch): در فضاهای ابری مانند Dropbox و دستورهایی مانند rsync در Linux، تغییر یک فایل رو یک کامپیوتر موجب تغییر فایل در دیگر کامپیوتر های همگام میشود

۹. رمزنگاری یا Cryptography

۱۰. زنجیره بلوکی یا Block Chain

۲.۱ توضیحات مسئله درهم سازی یا interface

دنبال داده ساختاری هستیم که بر روی تعدادی شئ که هرکدام دارای یک کلید منحصر به فرد هستند عملیات زیر را انجام دهد.
۱. insert: یک کلید میگیرد و آنرا در داده ساختار درج میکند. به دلیل منحصر بودن کلیدها، درصورت وجود یک شئ با کلید مدنظر، مقدار آن در داده ساختار آپدیت میشود؛ یعنی مقدار جدید جایگزین مقدار قبلی میشود.
۲. delete: یک کلیذ میگیرد و درصورت وجود آن در داده ساختار، آنرا حذف میکند.
۳. search: کلید را به عنوان ورودی میگیرد و در صورت وجود کلید در داده ساختار، شئ متناظر با آنرا برمیگرداند.
از دانسته های قبلی میدانیم بهترین داده ساختار برای پیاده سازی این عملیات درخت های دودویی جست و جوی متوازن (مانند AVL یا R-B) است که هر کدام از عملیات بالا را در زمان $O(logn)$ انجام میدهند. اما میتوان این زمان را به $O(1)$ در حالت میانگین کاهش داد.

۳.۱ چند نکته درمورد درهم سازی و Dictionary پایتون

درهم سازی در مدل مقایسه بیان نمیشود، زیرا در اینصورت مرتبه زمانی کمتر از $logn$ نمیشد.
همچنین در واسط درهم سازی نیازی به find _next()/prev() نداریم. البته در مدل word RAM دو عملیات ذکر شده را میتوان در زمانی کمتر از $O(1)$ پیاده سازی کرد.
جدول درهم سازی در پایتون Dictionary نام دارد که عملیات زیر روی آن اجرا میشود:

d = {'key1': 5, 'key2': 10}
d['key1'] → 5 , d[5] → KeyError
'key2' in d → True , 10 in d → False
d.item() → returns all dictionary (key and value)

۴.۱ پیاده سازی جدول درهم سازی به روش Direct Access Array

در این شیوه پیاده سازی، از کلید ها به عنوان اندیس های آرایه برای ذخیره سازی اشیا استفاده میشود. در این شیوه پیاده سازی، عملیات سه گانه ذکر شده (insert، delete و search) هرسه در زمان $O(1)$ انجام میشوند.
اما از محدودیت های این روش میتوان به:
۱. حافظه زیاد: هنگامی که کلیدها بزرگ شوند، اندازه آرایه نیز بزرگتر میشود
۲. cast کردن کلیدها به عدد: از انجا که کلید ها به عنوان اندیس نیز استفاده میشوند

باید مقادیر صحیح و نامنفی داشته باشند.
راه حل مشکل اول Hash و راه حل مشکل دوم Pre Hahs نام دارد.

۱.۴.۱ پیش درهم سازی یا **Pre Hash**

از آنجا که تمام اطلاعات در کامپیوتر به صورت صفر و یک نگه داری میشوند، پس میتوان برای هر داده یک نمایش عددی نیز داشت. تابع Pre Hash در زبان های مختلف پیاده سازی شده است (تابع hash در پایتون) که ورودی آن یک کلید و خروجی آن یک عدد صحیح و نامنفی یکتا است. همچنین کلید ها نیز باید غیر قابل تغییر (immutable) باشند.

۲.۴.۱ درهم سازی یا **Hash**

ایده حل مشکل ذکر شده، تقلیل مجموعه کلید هاست. اگر مجموعه کلید های اولیه را u بنامیم، میخواهیم u را به یک مجموعه کوچکتر با m عضو کاهش دهیم.
h را تابع درهم سازی مینامیم و داریم:

$h : u \to \{0, 1, 2, ..., m-2, m-1\}$

در حالت ایده آل داریم:

$m \approx n(= size(u))$

پس با حل دو مشکل ذکر شده، در پیاده سازی به روش Direct Access Array مراحل زیر را طی میکنیم: (x یک کلید از u میباشد)

$x \xrightarrow{PreHash} x' \xrightarrow{Hash} h(x') \to x \text{ is in index } h(x') \text{ of array}$

حال حالتی را در نظر میگیریم که دو کلید x' و y' بعد از اعمال تابع h، خروجی یکسان داشته باشند ($h(x') = h(y')$) یعنی هردوی کلیدها در یک خانه از آرایه دخیره بشوند. این حالت برخورد یا Collision نام دارد.
ساده ترین راه حل برای این مشکل، استفاده از زنجیر (chain یا linked list) است؛ یعنی هر خانه از آرایه یک زنجیر است که اعضای آن، عناصر با $h(x)$ های یکسان هستند.
درنهایت یا حل این مشکل مرتبه زمانی عملیات مربوط را ارائه میکنیم:

1. insert: $O(1)$

(در صورتی که نیاز به بررسی عدم وجود عنصر با کلیذ تکراری داشته باشیم مرتبه زمانی به $O(n)$ میرسد.)

2. delete: $O(n)$
3. search: $O(n)$

در بدترین حالت، همه اعضا در یک زنجیر ذخیره میشوند و مرتبه زمانی حذف و جست و جو به $O(n)$ میرسد.

جزوه جلسه پانزدهم داده ساختارها و الگوریتم

۲۳ آبان ۱۴۰۰

فهرست مطالب

۱ درهم سازی با فرض درهم سازی یکنواخت ساده

در این حالت فرض میکنیم تابع درهم سازی، هر کلید را با احتمال مساوی و مستقل از بقیه کلیدها به یکی از خانه های جدول درهم سازی می نگارد (map میکند). احتمال ذکر شده در این حالت برابر $1/\mathrm{m}$ میباشد که m اندازه جدول میباشد.
تابع درهم سازی که طراحی میکنیم، هر کلید را به صورت مجزا به یک خانه مپ میکند و عملیات هر کلید، تاثیری رو عملیات متناظر با کلید های دیگر ندارد. به همین دلیل احتمال نگاشت، مستقل است.
همچنین چنین فرضی در عمل به صورت صد در صدی عملی نمیشود و با ارائه روش هایی میتوان به فرض ارائه شده بسیار نزدیک شد. (مانند درهم سازی سراسری)

۱.۱ ضریب بارگذاری یا α

برای سادگی محاسبات، ضریب بارگذاری را تعریف میکنیم:

$\alpha = \frac{n}{m}$

در این رابطه، n تعداد اشیا و m اندازه جدول میباشد.
با درنظر گرفتن درستی فرض ما، میانگین اندازه زنجیر در هر خانه به صورت میانگین برابر α میشود.
حال میتوان مرتبه زمانی عملیات مربوط به جدول درهم سازی را بررسی کرد

۱.۱.۱ search(key)

میانگین زمان جست و جو در این حالت برابر $O(1+\alpha)$ میباشد. همچنین اگر $m = \omega(n)$ باشد، آنگاه زمان جست و جو به $O(1)$ میرسد.

۱.۱.۲ insert(key, value), delete(key)

مطلوب است که زمان این عملیات نیز برابر $O(1)$ باشد. در ابتدا، درکی از میزان بزرگی m نداریم؛ پس میتوانم:
۱. یک جدول بسیار بزرگ در نظر بگیریم که در این صورت میتواند بخش بزرگی از آن استفاده نشده باقی بماند و حافظه اضافی زیاده توسط الگوریتم اشغال شود.
۲. برخلاف حالت اول، m را کوچک در نظر گرفت که در این صورت α بزرگ میشود و مرتبه زمانی جست و جو افزایش می یابد.
بهترین رویکرد، استفاده از آرایه پویا است؛ به این صورت که هنگام پر شدن جدول، جدولی جدید به اندازه ۲ برابر قبلی و هنگام خالش شدن $3/4$ جدول، جدولی جدید با اندازه نصف جدول قبلی میگیریم و تابع درهم سازی جدیدی در نظر گرفته و عناصر را

تحت آن تابع، به خانه های جدول جدید نگاشت میکنیم.
در نتیجه، هزینه زمانی درج و حذف نیز به صورت سرشکن برابر $O(1)$ میشود.

۲ توابع درهم سازی متداول

۱.۲ روش تقسیم

$h(k) = k \, mod \, m$
این روش در عمل وقتی مفید است که m عددی اول باشد و به توان های اعداد کوچک مخصوصا ۲ و ۱۰ نزدیک نباشد.
برای مثال اگر $m = 10$ باشد، آنگاه k های زوج در خانه های زوج جدول و k های فرد در خانه های فرد جدول قرار میگیرند که به دلیل قابل پیشبینی بودن تابع Hash نامطلوب است.
همچنین از اعداد اول بزرگ به فرم $2^m - 1$ (مانند $2^{31} - 1$) نیز استفاده نمیکنیم؛ زیرا کلید هایی با مقادیر کم در تعیین $h(k)$ تاثیر بیشتری میگذارند.
از معایب این روش میتوان به زمان بر بودن پیدا کردن یک عدد اول در هربار تغییر اندازه جدول اشاره کرد. همچنین عملگر باقی مانده (mod) زمان بیشتری نسبت به ضرب یا جمع برای انجام میخواهد.

۲.۲ روش ضرب

$h(k) = [(a.k) \, mod \, 2^w] >> (w-r)$
a یک عدد تصادفی در بازه $[2^{w-1}, 2^w]$ و فرد است که ترجیحا به دو سر بازه نزدیک نیست و w برابر تعداد ارقام k و a در مبنای دو میباشد و در نهایت $m = 2^r$
a و k هردو w بیتی هستند؛ پس حاصل ضرب آنها حداکثر 2w بیتی خواهد شد. در عملیات باقی مانده گیری، تنها w بیت کم ارزش حاصل ضرب باقی می ماند و در نهایت با شیفت به راست، w-r بیت کم ارزش دیگر هم از بین رفته و تنها r بیت باقی میماند.
در این روش به دنبال m نمیگردیم و همچنین محاسبه آن نیز راحت است. محاسبه باقی مانده نیز به روش معمول انجام نمیشود و زمان بر نیست (مثلا میتوان حاصل ضرب را با یک رشته بیتی که w بیت با ارزش آن صفر و w بیت کم ارزش آن یک است and منطقی کرد).
در این روش عدد تصادفی a در ابتدا یک بار انتخاب میشود و به هنگام تغییر جدول، تغییر نمیکند. همچنین غیر قابل پیشبینی بودن a از نقاط قوت این روش است که باعث میشود ورودی بد برای تابع تولید نشود.

۳.۲ درهم سازی سراسری یا Universal Hash

$h(k) = [(a.k + b) \, mod \, P] \, mod \, m$
P یک عدد اول بزرگ (بزرگتر از اندازه مجموعه U(مجموعه جهانی کلیدها)) و a و b نیز دو عدد تصادفی صحیح در بازه $[0, P-1]$ میباشند. عدد P نیز به دلیل تعریف آن، یکبار در ابتدا محاسبه میشود و تغییر نمیکند.
به دلیل اینکه این روش به مقادیر تصادفی a و b بستگی دارد، به خانواده از توابع درهم سازی معروف است؛ گویی بسته به مقادیر a و b یک تابع درهم سازی جدید انتخاب میشود.
اثبات میشود برای دو کلید مجزای k_1 و k_2 فرض درهم سازی یکنواخت ساده برقرار است؛ به عبارت دیگر:
$\forall k_1, k_2 \text{ and } k_1 \neq k_2 : P_{a,b}[h(k_1) = h(k_2)] = 1/m$
از انجا که این احتمال برای هر کلید بیان شد، پس احتمال روی تصادفی بودن a و b است.

۳ سرفصل مطالب جلسات آینده

۱.۳ ادامه روش های مقابله با برخورد

۱. استفاده از زنجیر

۲. Open Addressing یا تلاش های متوالی با چند تابع درهم سازی برای $h(k)$ به نحوی که خانه $h(k)$ام جدول خالی باشد.
در این روش به دنبال کاهش تعداد تلاش یا برای پیدا کردن جایگاه خالی هستیم. یکی از اقداماتی که در این راستا انجام میدهیم ۲ برابر کردن اندازه جدول به هنگام پر شدن نصف خانه های آن است.
در عمل، این روش بیشتر از روش استفاده از زنجیر استفاده میشود.

۳. Perfect Hashing

۴. درهم سازی کوکو

جزوه جلسه شانزدهم داده ساختارها و الگوریتم

۲۵ آبان ۱۴۰۰

فهرست مطالب

۱ روش آدرس دهی باز (Open Addressing) برای مقابله با برخورد

ایده حل مشکل این است که به جای استفاده از زنجیر در خانه های جدول، کل اشیا را بدون استفاده از زنجیر در خود جدول ذخیره کرد و در صورت برخورد دو شی، تلاشی مجدد برای درج شی میکنیم. در این شیوه اندازه جدول (m) را همواره طوری نگه میداریم که: $m \geq 2n$

۱.۱ دنباله وارسی

دنباله وارسی شی x (Probing Sequence) به صورت زیر تعریف میشود:
$h(x,0),\ h(x,1),\ h(x,2),\ \ldots\ ,h(x,m-1)$
در این دنباله، $h(x,i)$ نشانگر تلاش $i+1$ام برای درج عنصر x در جدول تحت تابع درهم سازی h میباشد.
در یک دنباله وارسی، مطلوب این است که طول دنباله تا حد امکان کاهش یابد و به مقادیری مانند n یا $m/2$ نرسد.
حال عملیات سه گانه جدول درهم سازی را با این روش بررسی میکنیم:

۱.۱.۱ insert

برای درج کافی است از ابتدای دنباله شروع به بررسی خانه های جدول بکنیم و هرگاه به اولین خانه خالی رسیدیم، عنصر را درج کنیم.

۲.۱.۱ search

برای جست و جو نیز خانه های جدول را به ترتیب دنباله وارسی بررسی میکنیم و این کار را تا رسیدن به شی مدنظر یا یک خانه ادامه میدهیم. این بدین معنی است که هر شی x تا یک مرحله خاص از دنباله جلو رفته است و اگر به یک $h(x,i)$ خالی برسیم یعنی x در جدول درج نشده است.

۳.۱.۱ delete

برای حذف عنصر x نیاز به دقت بیشتری داریم؛ اگر پس از حذف یک کلید خانه مربوط به آنرا خالی کنیم، جست و جو های بعدی ممکن است درست کار نکند. به همین جهت پس از حذف شئ، در خانه مربوط به آن مینویسم "حذف شده". به این تکنیک، پرچم (flag) یا سنگ قبر (tomb stone) میگویند.

هنگام جست و جو، این خانه ها به مانند یک خانه پر در نظر گرفته میشوند و در هنگام درج نیز مشابه خانه های خالی هستند. در هنگام درج نیز برای جلوگیری از مشکلات احتمالی، با جست و جو در جدول، تکراری بودن کلید را بررسی میکنیم؛ اگر کلید تکراری نبود به صورت عادی درج انجام میگیرد و درغیر اینصورت در خانه اول جست و جو شده درج انجام میگیرد.
حذف زیاد باعث افزایش پرچم ها در جدول و افزایش زمان اجرای الگوریتم میشود. برای حل این مشکل، هنگامی که تعداد خانه های با پرچم به ۱۰ درصد کل جدول رسید، جدولی جدید با تایع درهم سازی و اندازه قبلی میسازیم. اثبات میشود که هزینه سرشکن ایجاد جدول جدید برابر $O(1)$ میباشد.

۲.۱ روش های وارسی

۱.۲.۱ وارسی خطی یا **Linear Probing**

$h(x,i) = (h'(x) + i)\ mod\ m$
به مانند پارک کردن ماشین در یک قسمت از پارکینگ عمل میکنیم؛ یعنی از یه جای خالی شروع کرده و به جلو میرویم تا به اولین جای خالی برسیم.
تابع $h'(x)$ نیز یک تابع درهم سازی از توابع معرفی شده در جلسه قبل است.
از معایب این روش میتوان به پر شدن یک قسمت از جدول (اینجاد خوشه) اشاره کرد. هرگاه که $h'(x)$ در یک قسمت پر جدول بیوفتد، زمان درج و جست و جو نیز افزایش می یابد. هرچه خوشه ها بزرگتر باشند، احتمال رشد آنها نیز بیشتر میشود و احتمال وجود خوشه به اندازه $\Theta(logn)$ نیز افزایش می یابد که زمان های درج و جست و جو در این حالت به $O(logn)$ میرسد.

۲.۲.۱ وارسی درجه دو

$h(x,i) = ((h'(x) + ax^2 + bx)\ mod\ m$
تابع $h'(x)$ نیز مانند حالت قبل یه تابع معمول درهم سازی است. همچنین ضرایب a و b باید به خوبی انتخاب شوند و لزومی به انتخاب تصادفی نیست.
در روش قبلی، اگر یک خانه پر بود به سراغ خانه بعدی اش میرفتیم که این کار از معایب وارسی خطی بود؛ اما در این روش خانه بعدی که میخواهیم آنرا بررسی کنیم در یک ناحیه دیگر در جدول قرار دارد و به اصطلاح پخش میشویم و مشکل خوشه تا حدودی حل میشود.
ثابت شده وارسی درجه ۵ (با انتخاب مناسب ضرایب) خوب کار میکند و وارسی های درجه ۴ نیز بر خلاف درجه ۵، عملکرد بدی دارند.

۳.۲.۱ درهم سازی دوگانه

$h(x,i) = (h_1(x) + ih_2(x))\ mod\ m$
توابع $h_1(x)$ و $h_2(x)$ جز توابع معمول درهم سازی هستند که باید $h_2(x)$ و m نسبت به هم اول باشند. بدین منظور:
۱. m اول باشد و $h_2(x)$ بین ۰ تا m-1 باشد.
۲. m توانی از ۲ و $h_2(x)$ فرد باشد.
راه حل دوم برای درهم سازی دوگانه بهتر است.

۳.۱ فرض درهم سازی یکنواخت یا Uniform Hashing Assumption

این فرض بیان میکند که دنباله وارسی هر کلید، یکی از $m!$ جایگشت ممکن را به صورت تصادفی و مستقل تولید میکند.
هیچ کدام از روش های مطرح شده بصورت کامل این فرض را پیاده سازی نمیکنند اما درهم سازی دوگانه و درجه ۵ تا حد خوبی به این فرض نزدیک هستند.
حال با فرض درستی این فرض، زمان اجرای عملیات سه گانه را در روش آدرس دهی باز بررسی میکنیم:

۱.۳.۱ insert

در مرحله اول n خانه پر و $m-n$ خانه خالی هستند. پس احتمال موفقیت برابر $\frac{m-n}{m}$ میباشد.
$P = \frac{m-n}{m}$
در مرحله دوم یک خانه حذف می شود و احتمال به $\frac{m-n}{m-1}$ میرسد.
.
.
.
در مرحله iام احتمال موفقیت برابر با $\frac{m-n}{m-(i-1)}$ میباشد.
از مرحله دوم، احتمال موفقیت بزرگتر از P میباشد یا به دیگر بیان، حداقل برابر با P میباشد. پس میانگین تعداد مراحل مورد نیاز برای درج برابر میشود با:
$E[\#trials] \leq \frac{1}{P} = \frac{1}{1-n/m} = \frac{1}{1-\alpha}\ ,\ \alpha = n/m$
در نتیجه، زمان درج برابر با $O(\frac{1}{1-\alpha})$ میباشد.

۲.۳.۱ delete و search

به طریق مشابه و مانند درج، برای حذف و جست و جو نیز ثابت میشود که زمان مورد نیاز برابر $O(\frac{1}{1-\alpha})$ می باشد.

۲ فیلتر بلوم یا Bloom Filter

ابتدا با مفاهیم مربوط آشنا میشویم.
سرور های CDN سرورهایی هستند که برای ذخیره سازی داده های ثابت به کار میروند (مانند تصاویر و متن های ثابت یک سایت یا برنامه). از اولین شرکت هایی که چنین سرور هایی در اختیار شرکت هایی مانند یاهو قرار داد، آکامای نام داشت.
این شرکت دید که یک سوم حجم سرور هایش پر از فایل های one-hit-wonders (فایل هایی که یک بار و توسط یک فرد استفاده شده است) میباشد. راه حل این مشکل فیاتر بلوم بود.
اگر یک فایل توسط حداقل دو نفر استفاده میشد، روی سرور ها باقی میماند و در غیر اینصورت حذف میشد.
فیلتر بلوم یک روش بسیار پرکاربرد برای جست و جو (وجود یا عدم وجود) احتمالی یک عضو در مجموعه میباشد؛ بدین صورت که اگر این الگوریتم پاسخ یک کوئری جست و جو را مثبت بدهد (عنصر وجود دارد)، به احتمال $1-P$ این عنصر وجود دارد و اگر پاسخ منفی باشد، قطعا عنصر مدنظر وجود ندارد.
در فیلتر بلوم عمل درج داریم ولی حذف نداریم!
فیلتر بلوم در مرورگر کروم برای بررسی مخرب بودن یا نبودن یک سایت، در بیت کوین برای همگام سازی کیف پول، در اتریوم و در پایگاه های داده نیز استفاده میشود.

۱.۲ پیاده سازی فیلتر بلوم

در این فیلتر، یک جدول صفر و یک و به تعداد k تابع درهم سازی داریم. خروجی توابع درهم سازی، مکان یک ها را در جدول نشان میدهد و با یک کردن آن خانه ها درج صورت میگیرد.
برای جست و جو نیز k خانه که خروجی توابع درهم سازی هستند را بررسی میکنیم؛ اگر حداقل یکی صفر بود با قطعیت میگوییم عنصر در جدول وجود ندارد و در غیر اینصورت با یک احتمال خطا میگوییم عنصر در جدول وجود دارد. احتمال خطا نیز به صورت زیر تعریف میشود:

$$P = (1 - e^{-kn/m})^k$$

که با فیکس کردن احتمال خطا روی یک مقدار خاص، سایز جدول به دست میاید:
$m = \frac{-nlog_e p}{(ln2)^2}$, $k = (\frac{m}{n})ln2$

جزوه جلسه هفدهم داده ساختارها و الگوریتم

۲ آذر ۱۴۰۰

فهرست مطالب

۱ درهم سازی کامل یا Perfect Hashing

این متد درهم سازی اولین بار در سال ۱۹۸۴ در ژورنال ACM معرفی شد.
در این روش درهم سازی، یک جدول $\mathrm{m = n}$ برای ذخیره سازی اشیا نیاز است. همچنین در نسخه اصلی این روش، همه اشیا از ابتدا در دسترس هستند و فقط میتوانیم اشیا را جست و جو کنیم (insert، delete و update نداریم) و این دیگر عملیات در نسخه های دیگر وجود دارند. حال به بررسی عملیات مورد انتظار در این روش می پردازیم:

۱.۱ delete و search

در روش آدرس دهی باز، در هر مرحله، یک بار تابع Hash محاسبه می شد؛ اما در این روش برای هر جست و جو (و در ادامه آن حذف) نیازمند ۲ بار محاسبه Hash می باشد.

۲.۱ insert

در مرحله اول با تابع درهم سازی اولیه h، کلید را به یک خانه از جدول می نگاریم (Mapping). در حالت نامطلوب، از قبل تعدادی شئ دیگر نیز در این خانه وجود دارد. فرض کنیم تعداد کلیدهای موجود در یک خانه برابر با k باشد. حال یک جدول دیکر به اندازه k^2 در نظر میگیریم و با تابع درهم سازی ثانویه h'، k شی مدنطر را در این جدول قرار می دهیم.
به احتمال حداقل $1/2$ برخورد صورت نمی گیرد. اگر در این مرحله نیز برخورد رخ داد، تابع ثانویه یا h' را عوض می کنیم. همچنین در نظر داریم که h' برای خانه های مختلف جدول متفاوت است.
ثابت می شود حافظه مورد نیاز، به صورت میانگین برابر $O(n)$ می باشد.(اثبات با در نظر گرفتن یک بودن ضریب بارگذاری (α) و انجام محاسبات روی توان دوم آن انجام می گیرد.)

۱.۲.۱ پارادوکس روز تولد

برای اثبات احتمال $1/2$ ذکر شده، به ذکر پارادوکس روز تولد می پردازیم.
پارادوکس روز تولد می گوید احتمال اینکه روز تولد دو نفر در یک جمع ۲۰ نفره یکسان باشد، حدودا $1/2$ است. ($20 \approx \sqrt{365}$)
به همین دلیل می توان ادعا کرد احتمال برخورد حداکثر برابر $1/2$ می باشد.

$$E[\#conflicts] = \binom{k}{2} \times \frac{1}{k^2} = \frac{k^2-k}{2k^2} = \approx 1/2 \ (< 1/2)$$

در عبارت بالا، $\binom{k}{2}$ برابر تعداد حالات انتخاب ۲ کلید و $\frac{1}{k^2}$ احتمال برخورد ۲ عضو انتخاب شده می باشد.
حال می توان نتیجه گرفت که برای میانگین تعداد برخورد ها در حالت های مختلف کمتر از $1/2$ در فضای صفر و یکی (برخورد میکند یا نمیکند) داریم:

$$Pr[conflicts] < 1/2$$

پس احتماب برخوردها نیز کمتر از $1/2$ است و در نتیجه برای نصف خروجی های h' برخورد نداریم.

۲ درهم سازی کوکو یا Cuckoo Hashing

روشی دیگر برای رفع مشکل برخورد، استفاده از درهم سازی کوکو است. ایده حل مشکل، استفاده از ۲ جدول (T_1 و T_2) و ۲ تابع مجزا و مستقل از هم h_1 و h_2 برای هر کدام از جدول هاست. اندازه جدول نیز در این حالت برابر $m = 2n$ می باشد و هر شی در یکی از جدول ها درج میشود.

۱.۲ search و delete

برای جست و جوی یک کلید، یک یا دو بار نیاز به فراخوانی تابع درهم سازی داریم؛ بدین صورت که ابتدا $h_1(key)$ را محاسبه میکنیم؛ اگر $T_1[h_1(key)]$ حاوی شی مدنظر بود آنرا برمی گردانیم. در غیر این صورت به سراغ $T_2[h_2(key)]$ می رویم. اگر شی مورد نظر پیدا شد آن را برمی گردایم و در غیر این صورت ادعا میکنیم شی مدنظر در جدول وجود ندارد.
برای حذف نیز مشابه مراحل بالا، اگر شی در هرکدام از مراحل پیدا شد آن را حذف میکنیم.

۲.۲ insert

برای درج هر شئ، ۲ کاندید داریم؛ $T_1[h_1(key)]$ و $T_2[h_2(key)]$. اگر خانه مربوط به T_1 خالی بود، درج در آن خانه انجام میگیرد و در غیر اینصورت به سراغ T_2 می رویم. اگر این خانه خالی بود درج صورت می گیرد و در صورتی که هردو خانه پر باشند، عنصر موجود در خانه $T_2[h_2(key)]$ را بیرون انداخته و شئ را در انجا درج می کنیم. حال عنصری را که بیرون انداخته بودیم مطابق الگوریتم ذکر شده در یکی از دو جدول درج می کنیم.
حالتی نامطلوب است که این روند بیرون انداختن عناصر و درج مجدد طولانی شود. اگر تعداد مراحل از $2logn$ بیشتر شد، جدول ها را از ابتدا با ۲ تابع درهم سازی جدید

میسازیم.
به دلیل اینکه رخ دادن چنین حالتی بسیار کم است، پس میتوان گفت هزینه درج به صورت سرشکن برابر $O(1)$ می باشد.

۳ توابع درهم سازی در رمزنگاری

توابع درهم سازی که در رمز نگاری استفاده می شوند پیچیده تر از توابع درهم سازی که بررسی کردیم (یا حداقل حاصل چندین مرتبه فراخوانی توابع معمول درهم سازی) هستند. همچنین خروجی ها نیز به دلیل افزایش امنیت دارای تعداد بیت زیادی هستند. (حداقل ۱۶۰ بیت)
حال به بررسی ویژگی های توابع مناسب رمزنگاری می پردازیم.

۱.۳ یک طرفه بودن (OW)

این ویژگی به زبان ساده یعنی برای هر y نتوان کلیدی مانند x پیدا کرد که $\mathrm{h(x)=y}$ طولانی بودن اندازه خروجی توابع درهم سازی به تقویت این ویژگی کمک می کنند؛ بدین صورت که با افزایش طول خروجی تولید حالت های مختلف برای هر بررسی $\mathrm{h(x)=y}$ وقت گیرتر می شود.
در سیستم عامل لینوکس، hash رمز تمامی اکانت کاربران در $\mathrm{/etc/shadow}$ نگه داری می شود. اگر تابع درهم سازی مورد استفاده این ویژگی را دارا نباشد، میتوان یک ورودی را به عنوان رمز یک اکانت وارد کرد که hash آن با hash رمز اصلی یکسان باشد و وارد اکانت شد. در این حالت لزومی بر برابر بودن رمز اصلی و رمز وارد شده وجود ندارد.
اگر کاربر نتواند وارد حساب خود شود، یعنی رمز وارد شده آن به صورت قطعی نادرست است ولی درصورت وارد شدن به حساب، نمیتوان ادعا کرد رمز وارد شده همان رمز اصلی کاربر است.

۲.۳ مقاوم در برابر برخورد (CR)

در این ویژگی انتظار داریم برای تابع درهم سازی نتوان x_1 و x_2 متفاوتی پیدا کرد که خروجی hash آنها یکسان باشد ($h(x_1) = h(x_2)$)
سناریویی را در نظر بگیرید که یک متن را امضا میکنیم. امضا کردن متن زمان بر است به همین دلیل Hash آنرا امضا میکنیم. (منظور از امضا کردن یک متد خاص برای رمز کردن یک فایل است که فرد امضا کننده فقط میتواند فایل را رمز نگاری کند و همه می توانند آن را رمزگشایی کنند). حال اگر این ویژگی برقرار نباشد، بعد از امضا کردن متن x_1 می توان ادعا کرد متن x_2 امضا شده است؛ زیرا که خروجی تابع Hash آن ها یکسان است.

۳.۳ مقاومت در برابر برخورد هدف دار (TCR)

این ویژگی نیز به معنی این است که برای کلید x_1 داده شده، نتوان x_2 پیدا کرد که $h(x_1) = h(x_2)$ و خروجی Hash آنها یکسان باشد ($x_1 \neq h_2$)
سناریوی امضای متن را مجددا در نظر بگیرید. در این حالت برای متنی که امضا کردیم نباید متن دیگری تولید کرد که Hash آنها یکسان باشد. (توجه شود در قسمت قبل هر دو متن x_1 و x_2 از ابتدا در دسترس بودند.)
برای مثال دیگر، سرویس DropBox را در نظر بگیرید. به دلیل حجیم بودن فایل ها به نسب Hash آن ها، برای تضخیص تغییرات فایل ها و سینک کردن آن، از Hash فایل در سرور استفاده می شود. اگر این ویژگی برقرار نباشد، می توان فایل را تغییر داد به گونه ای که این تغییر مخفی بماند.
بدین منظور این سرویس به هنگام دریافت یک فایل، Hash آن را نیز در اختیار کاربر قرار میدهد تا از تغییرات فایل در مسیر مطلع شود. احتمال برابر بودن Hashها و یکسان نبودن فایل ها در صورت تغییر، بسیار کم است.

۴.۳ امنیت توابع درهم سازی

توابه بسیاری برای رمزنگاری پیشنهاد شده اند. اما هنگامی که مشخص شود تابع معرفی شده حداقل یکی از ویژگی های فوق را ندارد ناامن طلقی میشود. برای مثال تابع MD-5 که نسل پنجم توابع MD می باشد و برای درهم سازی رمز های کاربران در لینوکس نیز استفاده میشد ثابت شده که ویژگی CR را ندارد.
تابع SHA-1 نیز بطور مشابه به دلیل نقض ویژگی CR کنار گذاشته شد.
تابع SHA-2 که در سال ۲۰۱۵ معرفی شده، جدیدترین تایع پیشنهاد شده و مورد استفاده در این زمان است.

جزوه جلسه هجدهم داده ساختارها و الگوریتم

۷ آذر ۱۴۰۰

فهرست مطالب

۱ گراف

هر گراف مجموعه ای از رئوس و یال ها می باشد و به جز این دو مورد، بقیه اطلاعات دور ریخته می شود. $G = (V, E)$ نشان دهنده گراف G با مجموعه رئوس V و مجموعه یال های E است.
اگر گراف جهت دار باشد، هر یال یک زوج مرتب از ۲ راس $((u, v))$ می باشد که به معنی یک یال از راس u به v است. در گراف بدون جهت نیز هر یال، یک مجموعه دوتایی از راس هاست $(\{u, v\})$ که به معنی وجود اتصال بین ۲ راس u و v است.
در این درس، مباحث مرتبط با پیمایش گراف و بعد از آن، پیدا کردن کوتاه ترین مسیر بررسی می شود.

۱.۱ مثال های گراف

کاربرد های پیمایشی:

- پیدا کردن مسیری از یک راس به عنوان مبدا به راس مقصد
- بازدید از همه راس ها یا یال های گراف که از یک راس مبدا به نام S قابل دسترس است.

کاربرد های غیر پیمایشی:

- رنگ آمیزی گراف به نحوی که دو سر هر یال دارای به یک رنگ رنگ آمیزی نشده باشد.
- Matching
- درخت پوشای کمینه
- گراف مسطح

۲.۱ کاربردهای گراف

- خزش وب یا Web Crawling: الگوریتم گوگل برای شناسایی صفحات جدید وب اینگونه کار میکند که تمام لینک های موجود در صفحات شناخته شده را پیمایش می کند و در صورت پیدا کردن صفحه جدید که ناشناخته است، آنرا برای موتور جست و جو شناسایی می کند.
- شبکه های اجتماعی: پیمایش شبکه دوستی و پیشنهاد دادن دوستان جدید

- مسیریابی یا Navigation
- پیدا کردن مسیرهای هوایی مستقیم
- زباله روبی در زبان های برنامه نویسی (Garbage Collection): آبجکت هایی که به همدیگر رفرنس دارند، در صورتی که یک رفرنس حذف شود، زباله روب اشیا بعدی که به رفرنس حذف شده، خود رفرنس داشتند را زباله در نظر می گیرد.
- چک کردن مدل یا Model Checking: در علوم مختلف و در صنایع حساس (از آسانسور گرفته تا صنایع نظامی و هسته ای!) مدل های خاصی را طراحی می کنند تا عملکرد پروژه در حالت های مختلف بررسی شود. حالت های مختلف در این حالت، پیمایش و بررسی می شوند.
- حل برخی پازل ها مانند روبیک $2 \times 2 \times 2$: تمام حالت ها با Model Checking بررسی می شوند.
- شواهد دیجیتال: برای مثال حرکت های کارآگاهی! و دنبال کردن پیام ها و تماس ها

۳.۱ داده ساختار های مناسب برای ذخیره سازی گراف

فرض می کنیم رئوس گراف با اعداد ۰ تا n-1 نامگذاری شده اند.

۱.۳.۱ ماتریس مجاورت

یک ماتریس به ابعاد $n \times n$ به نام A که $A[i][j] = 1$ اگر و تنها اگر $(i,j) \in E$ یا $\{i,j\} \in E$
از مزایای این روش می توان به انجام اعمال جبرخطی، بررسی وجود یک یال و حذف یا اضافه کردن یک یال در $O(1)$ اشاره کرد.
در طرف دیگر زمانبر بودن استخراج لیست همسایه ها ($\Theta(n)$) و حافظه مورد نیاز به اندازه $\Theta(n^2)$ برای ذخیره سازی اطلاعات از معایب این روش هستند.

۲.۳.۱ لیست مجاورت

برای هر راس مانند u، لیست همسایه هایش یا $Adj(u)$ را نگه داری می کنیم.
در این شیوه، مجموع اندازه لیست ها برای گراف جهت دار از مرتبه e و برای گراف بدون جهت از مرتبه $2e$ است. حافظه مصرفی در این روش نیز برابر $\Theta(n+e)$ می باشد. (e تعداد یال ها و n تعداد رئوس است)
در این روش با توجه به توضیحات فوق، مشکل حافظه و دسترسی به همسایه ها حل

شد. اضافه کردن یک یال نیز در $O(1)$ انجام می گیرد.
اما چک کردن و حذف و اضافه کردن یک یال دیگر در $O(1)$ قابل انجام نیست. (می توان به جای لیست پیوندی از جدول درهم سازی برای ذخیره سازی همسایه ها استفاده کرد که عملیات بررسی و حذف و اضافه در $O(1)$ انجام گیرد اما پیاده سازی با لیست پیوندی اصطلاحا good enough است.)
همسایه های u را میتوان به صورت یک آرایه از همسایه ها ($Adj[u]$) یا یک property در کلاس مربوط به راس ($u.neighbour$) در نظر گرفت.
همچنین نگه داری همسایه های هر راس میتواند صریح (از ابتدا همه لیست ها موجود باشند) یا ضمنی (لیست ها به هنگام نیاز ساخته شوند) باشد.
در ۹۰ درصد مواقع، گراف با این متد پیاده سازی می شود.

۴.۱ الگوریتم پیمایش جست و حوی سطح-اول گراف یا Breadth First Search

از یک راس در سطح صفر شروع به پیمایش می کنیم و همسایه های هر سطح، سطح بعدی را می سازند که هر سطح یک لایه (Layer) نام دارد.
در هر مرحله، یک مجموعه مرزی به نام frontier داریم که آخرین لایه پیمابش شده است و از روی لایه مرزی و همسایه هایی که تا کنون بازدید نشده اند، لایه بعدی یا next ساخته می شود ک مجموعه مرزی در مرحله بعدی است.

```
1.  def BFS(V, Adj, s)
2.      level = {s: 0}
3.      parent = {s: None}
4.      i = 1
5.      frontier = [s]
6.      while frontier:
7.          next = []
8.          for u in frontier:
9.              for v in Adj[u]:
10.                 if v not in level:
11.                     level[v] = i
12.                     parent[v] = u
13.                     next.append(v)
14.         frontier = next
15.         i += 1
```

۱.۴.۱ تحلیل زمانی الگوریتم **BFS**

هر راس حداکثر یک بار در مجموعه frontier قرار می گیرد. پس حلقه for خط هشتم گویا روی کل رئوس گراف است. برای هر راس نیز روی تمام همسایه هایش لوپ میزنیم که برابر با مجموع طول لیست های مجاورت است. پس شرط خط دهم نیز به اندازه تعداد یالها (m یا $2\mathrm{m}$) بررسی می شود و بلوک آن نیز برای هر راس یک بار انجام میشود.
با تفاسیر فوق این الگوریتم از مرتبه $\Theta(n+e)$ می باشد.
در نظر داریم که این الگوریتم برای مولفه های همبند، گراف را پیمایش میکند.

جزوه جلسه نوزده داده ساختارها و الگوریتم

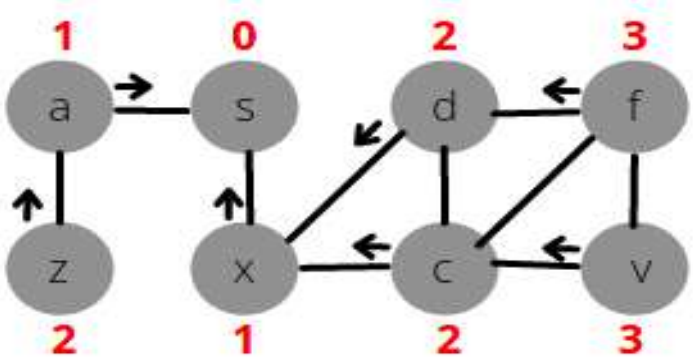
۹ آذر ۱۴۰۰

فهرست مطالب

۱ ادامه مباحث پیمایش BFS

۱.۱ مثال از پیمایش BFS یک گراف ۸ راسی

در گراف زیر، مبدا راس s در نظر گرفته شده. لایه ای که هر راس در آن قرار دارد با عدد قرمز روی راس و Parent هر راس نیز با فلش مشخص شده است.



شکل ۱: پیمایش گراف هشت راسی با الگوریتم BFS

در هر مرحله از الگوریتم نیز، frontier و next به شکل زیر مشخص می شوند:

- frontier $= \{s\} \implies$ next $= \{a, x\}$
- frontier $= \{a, x\} \implies$ next $= \{z, d, c\}$
- frontier $= \{z, d, c\} \implies$ next $= \{f, v\}$
- frontier $= \{f, v\} \implies$ next $= \{\}$ (end of algorithm)

در این پیمایش، اگر راسی در هیچ یک از levelها قرار نگرفت، آن راس را در لایه ∞ قرار می دهیم. همچنین منظور از لایه iام یعنی با دقیقا i حرکت از راس به مبدا می رسیم.

۲.۱ درخت BFS یا درخت کوتاه ترین مسیر

با توجه به مسیر هایی که از هر راس به والدش در شکل بالا داریم (فلش ها) گراف جهت دار متناظر با گراف، به نام درخت BST به وجود می آید. در این درخت، یال های بدون جهت حذف می شوند و از هر راس کوتاه ترین مسیر به مبدا مشخص می شود.

اگر میخواستیم تمام مسیر های منتهی به مبدا را ذخیره کنیم حافظه ای به اندازه $\Theta(n^2)$ نیاز بود. اما در این گراف برای ذخیره سازی نیاز به حافظه ای از مرتبه n $(\Theta(n))$ نیاز داریم.
یال های بدون جهت که در درخت وجود ندارند یا بین رئوس یک لایه (مانند یال fv) یا بین لایه های متوالی (مانند یال fc) قرار دارند.

۲ الگوریتم جست و جوی عمق اول یا Depth First Search (DFS)

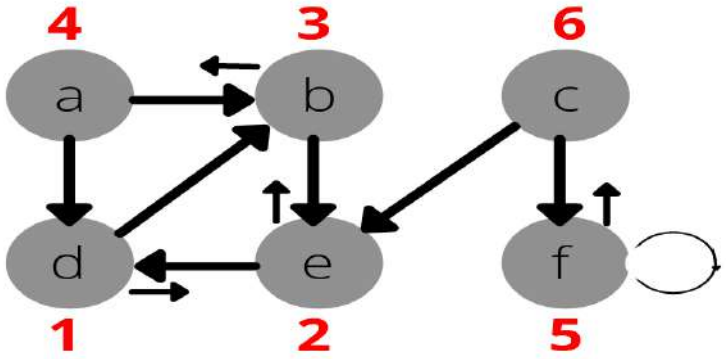
برای خروج از یک Maze رویکرد ما می تواند در پیش گرفتن یک مسیر با قرار دادن یک دست ثابت روی دیوار باشد. در این رویکرد بن بست ها را به صورت خودکار بر میگردیم و مسیر های جدید را بازدید میکنیم. همچنین درصورتی که از ورودی هزارتو وارد شده باشیم در loop نمی افتیم.

```
1.  def DFS (V, Adj):
2.      parent = []
3.      def visit (s)
4.          for v in Adj[s]:
5.              if v not in parent:
6.                  parent[v] = s
7.                  visit(v)
8.      for s in V:
9.          if s not in parent:
10.             parent[s] = None
11.             visit(s)
```

در این الگوریتم بر خلاف BFS راس مبدا نداریم.
حلقه for خط ۸ نیز باعث میشود مطمئن شویم همه رئوس بازدید شده اند.

۱.۲ پیمایش یک گراف ۶ راسی با الگوریتم DFS

پیمایش از راس a شروع شده است. parentها نیز با فلش مشخص شده اند و ترتیب اتمام الگوریتم برای رئوس با عدد قرمز رنگ مشخص شده است.

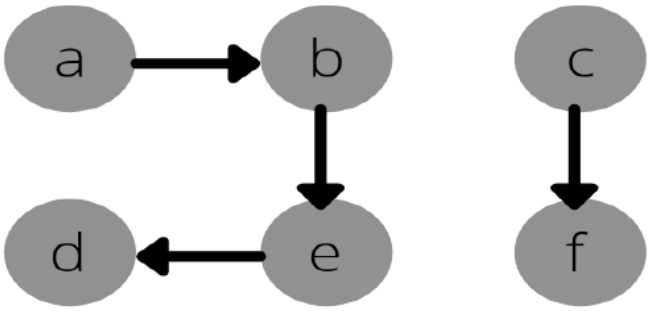


شکل ۲: پیمایش یک گراف ۶ راسی با الگوریتم DFS

در استک حافظه نیز به ترتیب از سر استک در مرحله اول visit(d)، visit(e)، visit(b) و visit(a) قرار دارند. سپس با فرض اینکه تابع visit برای راس c صدا زده شود، در مرحله دوم ترتیب به صورت visit(f) و visit(c) می باشد.

۲.۲ درخت **DFS**

مانند درخت BFS، مسیرهایی که در پیمایش DFS طی می کنیم را درخت DFS می نامیم. ممکن است این درخت همبند نباشد و چند مولفه همبند داشته باشد (جنگل باشد).

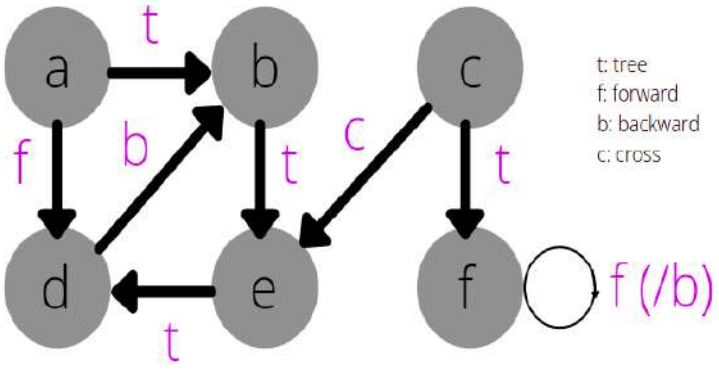


شکل ۳: درخت DFS گراف مثال بالا

۱.۲.۲ دسته بندی یالهای یک گراف جهت دار

- یالهای درخت DFS
- یالهای رو به جلو یا forward از ریشه به برگ ها و گره ها
- یالهای بازگشتی یا backward از گره ها و برگ ها به ریشه
- یالهای متقاطع یا cross یین دو شاخه از درخت

شکل صفحه بعد یالهای گراف ۶ راسی ذکر شده را نشان میدهد.



شکل ۴: یالهای گراف جهت دار ۶ راسی

۲.۲.۲ دسته بندی یالهای یک گراف بدون جهت

در گراف بدون جهت یالهای backward و forward یکی هستند و به آنها یال backward گفته می شود.
همچنین یال متقاطع نیز نداریم. زیرا اگر فرض کنیم که چنین یالی وجود دارد، امکان ندارد یک شاخه را تا ته پیمایش کنیم و یال متصل به شاخه دیگر یا همان یال کراس را پیمایش نکنیم. پس هر دو راس متصل به هم در یک شاخه قرار دارند.

۳.۲ زمان اجرای الگوریتم DFS

هر راس در این الگوریتم تنها یک بار بررسی می شود و تابع visit آن تنها یکبار فراخوانی می شود. هر تابع visit نیز برای یک راس به تعداد همسایه های هر راس زمان می برد تا اجرایش به اتمام برسد. پس در نتیجه زمان اجرای الگوریتم برابر $\Theta(n+e)$ میباشد.
(تابع visit خط سه از اردر e و حلقه خط هشت از اردر n)

۳ مقایسه DFS و BFS

BFS مانند یک صف به صورت FiFo رئوس را پیمایش میکند ولی DFS مانند استک و به صورت LiFo پیمایش را انجام میدهد.

جزوه جلسه بیستم داده ساختارها و الگوریتم

۹ آذر ۱۴۰۰

فهرست مطالب

۱ محدودیت های DFS

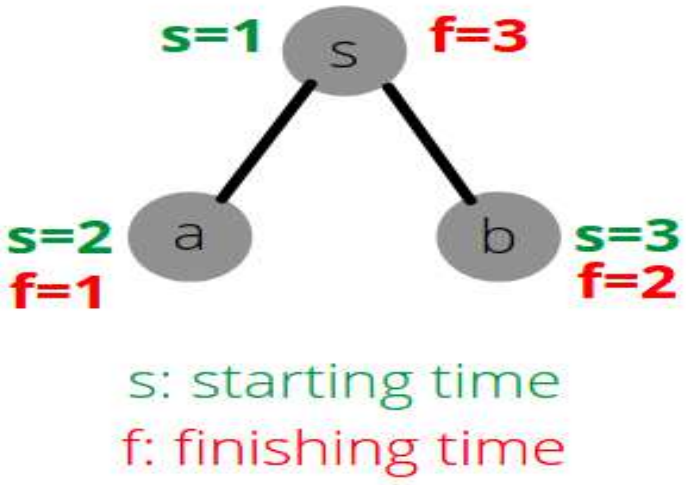
این الگوریتم برای گراف هایی با تعداد راس و یال فراوان می تواند منجر به بروز خطا بشود. حافظه ای که کامپیوتر از سیستم عامل می گیرد، بخش کمی از آن را به stack اختصاص می دهد و به همین دلیل،پیمایش یک مسیر طولانی می تواند منجر به بروز خطای پر شدن حافظه استک بشود.

۲ کاربردهای الگوریتم DFS

پیش از شروع، دو مفهوم در مورد الگوریتم DFS:

- **starting time**: زمان اجرای تابع visit را برای هر راس نشان می دهد. (بعد از شروع اجرای تابع و خط def visit(s)
- **finishing time**: زمانی را نشان میدهد که اجرای تابع visit برای راس تمام شده است. (در انتهای تابع و خارج از حلقه for)

توجه شود که زمان های فوق منطقی هستند و صرفا توالی زمانی را نشان میدهند.
مثال یک گراف سه راسی در شکل زیر آمده است.



شکل ۱: زمان های شروع و پایان برای یک گراف سه راسی

حال به ذکر کاربردها می پردازیم:

۱. پیدا کردن دور در گراف جهت دار و بدون جهت

۲. بررسی دو بخشی بودن گراف برای رنگ آمیزی (رنگ آمیزی گراف به نحوی است که رنگ دو سر هر یال متفاوت باشد)

۳. مرتب سازی توپولوژیک گراف جهت دار بدون دور (DAG)

۴. مولفه های همبندی گراف (با اجرای هر حلقه for در گراف، رئوسی که دیده شده باشند در یک مولفه قرار می گیرند.)

۵. پیدا کردن مولفه های قویا همبند در گراف جهت دار (بین هر دو راس در یک مولفه قویا همبند مسیر رفت و برگشت وجود دارد)

۶. پیدا کردن مولفه های دوهمبند راسی و یالی در گراف بدون جهت (در گراف دوهمبند راسی بین هر دو راس، ۲ مسیر وجود دارد که مسیرها راس مشترک ندارند و در گراف دوهمبند راسی بین هر دو مسیر بین ۲ راس، راس مشترک می تواند وجود داشته باشد ولی مسیرها یال مشترک ندارند)

۷. پیدا کردن راس یا یال برشی (با حذف راس یا یال برشی، مولفه های همبندی گراف افزایش می یابد)

با اضافه کردن starting time و finishing time، عملیات بالا در زمان $\Theta(n+e)$ انجام می گیرد.
حال به بررسی مورد اول، سوم و پنجم می پردازیم

۱.۲ پیدا کردن دور در گراف

ادعا میکنیم در هر گراف جهت دار یا بدون جهت دور وجود دارد اگر و تنها اگر یال بازگشتی (backward) داشته باشیم.
برای گراف جهت دار و بدون جهت اگر یال بازگشتی داشته باشیم، طبق تعریف یال بازگشتی دور نیز تشکیل می شود. حال اگر فرض کنیم دور داشته باشیم برای اثبات وجود یال بازگشتی مسیر $<v_1, v_2, ..., v_k>$ را در نظر می گیریم.

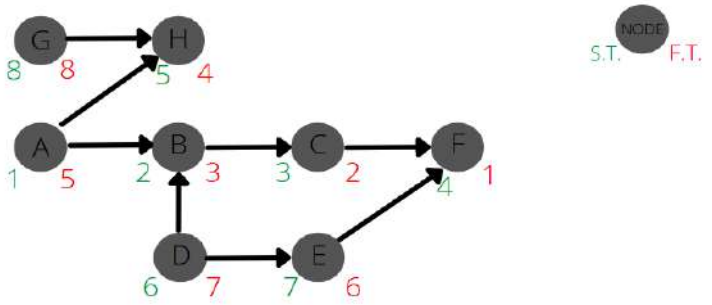
- لم: اگر یال (v_i, v_{i+1}) در گراف وحود داشته باشد، قبل از به پایان رسیدن $\text{visit}(v_i)$ تابع $\text{visit}(v_{i+1})$ با پایان می رسد.

اگر $\text{visit}(v_i)$ بعد از انمام $\text{visit}(v_{i+1})$ شروع شده باشد که لم اثبات می شود.
اگر $\text{visit}(v_{i+1})$ بعد از $\text{visit}(v_i)$ شروع شود، یا بعد مستقیما بعد از ویزیت v_i شروع

شده یا غیرمستقیم و بعد از فراخوانی visit برای چند راس دیگر. در هر دو حالت نیز مسئله اثبات شده است.
حال فرض کنید v_0 اولین راسی باشد که آنرا ویزیت می کنیم. به صورت استقرایی می توان ثابت کرد قبل از اتمام $\text{visit}(v_i)$، تابع $\text{visit}(v_k)$ برای رئوسی که $k > i$ است شروع شده وقبل از به پایان رسیدن $\text{visit}(v_i)$ به پایان رسیده است. پس پیش از اتمام $\text{visit}(v_0)$، تابع ویزیت برای همه v_iها شروع شده است. پس یال (v_k, v_0) یک یال بازگشتی است.

۲.۲ مرتب سازی توپولوژیک یک گراف جهت دار بدون دور

فرض کنید تعدادی کار داریم که بعضی از آنها پیش نیاز بعضی دیگر هستند. با مرتب سازی توپولوژیک میتوان ترتیبی از کارها ارائه داد که در آن پیش نیازی ها رعایت شده باشد.
برای این کار می توان رئوس را به ترتیب عکس finishing timeها مرتب کرد و خروجی داد. شکل زیر یک مثال از گراف ۸ راسی است.



طبق شکل بالا، یک ترتیب برای انجام کارها
G, D, E, A, H, B, C, F
می باشد.
توجه داشته باشید این مسئله با BFS قابل حل نیست. زیرا ممکن است بعضی رئوس پیمایش نشوند.
برای اثبات درستی این الگوریتم نیز از لم قبلی میتوان اثبات کرد اگر یال (u,v) در گراف وجود داشته باشد، قبل از اینکه $\text{visit}(u)$ به پایان برسد $\text{visit}(v)$ شروع می شود.
همچنین به دلیل عدم وجود دور اثبات فوق انجام می شود.

۳.۲ پیدا کردن مولفه های قویا همبند

با یک بار اجرای الگوریتم DFS نیز می توان مولفه های قویا همبند را پیدا کردن اما الگوریتم ذکر شده نیازمند دو بار اجرای الگوریتمDFS است.

۱. یکبار DFS را روی گراف اجرا می کنیم و finishing time رئوس را به دست می آوریم.

۲. جهت یال ها را عوض کرده و مجددا با اجرای DFS، زمان اتمام رئوس را محاسبه میکنیم.

۳. finishing timeها را به صورت نزولی مرتب می کنیم و راس ها را در هر درخت حاصل از جست و جوی عمق اول به عنوان یک مولفه قویا همبند خروجی می دهیم.

برای اطلاعات بیشتر به صفحه ویکی پدیا الگوریتم مراجعه کنید.
https://en.wikipedia.org/wiki/Kosaraju's_algorithm

جزوه جلسه بیست‌ویکم داده ساختارها و الگوریتم

۱۴ آذر ۱۴۰۰

فهرست مطالب

۱ مقدمات کوتاه ترین مسیر در گراف

الگوریتم BFS که در جلسات قبل بررسی کردیم، تا حدودی کوتاه ترین مسیر در گراف بدون وزن (گرافی که تفاوتی بین رئوس وجود ندارد) را در لایه های مختلف نشان می داد. حال در ادامه به بررسی الگوریتم های دیگر پیدا کردن کوتاه ترین مسیر در گراف می پردازیم.

۱.۱ کاربردهای کوتاه ترین مسیر

- مسیریابی جاده‌ای در برنامه هایی مانند waze یا google map
- مسیریابی شبکه در routerها برای رسیدن یک پکت از مبدا به مقصد

۲.۱ تعریف دقیق مسئله و برخی تعاریف مهم

- گراف وزن دار: در این گراف، به هر یال یک عدد حقیقی تحت عنوان وزن آن نسبت داده می شود. پس در نمایش گراف، تابع وزن یا w را نیز نشان می دهیم.
$G(V, E, W)$, $W: E \to R$

- مسیر: مسیر دنباله‌ای از رئوس است که هر یال بین ۲ راس متوالی، عضو E باشد. مسیر P در شکل زیر نمایش داده شده است.
$P=<v_0, v_1, ..., v_k>$, $(v_i, v_{i+1}) \in E$
طبق تعریف بالا مشکلی با تکراری بودن یال یا راس نداریم.

- مسیر ساده: مسیری که در آن راس تکراری نداریم.
در مسئله کوتاه ترین مسیر کار کردن با این تعریف سخت است، پس تعریف قبلی را در نظر می گیریم.

- وزن مسیر: وزن یک مسیر (w) برابر است با مجموع وزن یال های مسیر.
$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$

- وزن کوتاه ترین مسیر: بین دو راس u و v وزن کوتاه ترین مسیر را $\delta(u, v)$ می نامیم که برابر است با مینیمم وزن مسیرهایی که از u به v وجود دارد. همچنین دو راس که مسیری به هم ندارند وزن مینیمم آنها را ∞ در نظر می گیریم.
حالتی را در نظر بگیرید که دور منفی در مسیر داشته باشیم (دور منفی دوری است که جمع وزن یالهای آن منفی است). پس میتوان با هر با دور زدن طول مسیر را حداقل یک واحد کاهش داد و در نهایت می توان به $-\infty$ رسید. برای

رفع این مشکل به جای در نظر گرفتن مینیمم طول مسیرها، اینفیمم آن ها را در نظر می گیریم.

یکی از کاربرد های دور منفی در تبدیل ارز یا رمزارز می باشد. به این صورت که هر یال (u,v) با وزن w برابر است با نرخ تبدیل یک u به v می باشد. حال اگر از یک راس شروع به دور زدن کنیم و وزن یال ها را در هم ضرب کنیم در صورتی که مقدار نهایی برابر یک نشود، نشانگر وجود یک مشکل در وضعیت ثبات قیمت رمزارز هاست.
برای شهود بهتر نسبت به رمزارز میتوان وزن یالها را برابر لگاریتم نرخ تبدیل قرار داد و در صورتی جمع وزن ها صفر نشود، وجود یک مشکل گزارش می شود.

$$ln(w_1) + ln(w_2) + ... + ln(w_k) \neq 0$$

۳.۱ دسته بندی مسائل کوتاه ترین مسیر

۱. کوتاه ترین مسیر بین یک جفت راس

۲. کوتاه ترین مسیر از یک مبدا به بقیه راس‌ها (Single Source Shortest Path) یا درخت کوتاه ترین مسیر

۳. کوتاه ترین مسیر بین هر جفت راس (All Pair Shortest Path)

در این درس الگوریتم های دوم یا SSSP رابررسی می کنیم. همچنین الگوریتم هایی که می شناسیم (مانند برنامه مسیریابی waze) در بدترین حالت مسئله SSSP را حل میکنند.

۱.۳.۱ الگوریتم های معروف برای حل مسئله **SSSP**

- BFS: برای گراف هایی با وزن مثبت و برابر کاربرد دارد و در زمان $O(n+e)$ انجام می شود.
- کوتاه ترین مسیر در گراف جهت دار بدون دور: این الگوریتم نیز در زمان $O(n+e)$ انجام می شود.
- الگوریتم دکسترا (Dijkstra): این الگوریتم برای گراف هایی با وزن نامنفی در بهترین زمان $O(nlogn + e)$ کوتاه ترین مسیر را پیدا می کند.
- الگوریتم بلمن فورد: این الگوریتم در زمان $O(ne)$ کوتاه ترین مسیر را در هر گرافی پیدا می کند. به دلیل زیاد بودن زمان اجرا، استفاده از این الگوریتم زمانی توصیه می شود که نتوان از سه الگوریتم بالا استفاده کرد.

در مسائل SSSP مشابه آنچه در BFS دیدیم، میتوان درخت کوتاه ترین مسیر از یک راس را تشکیل داد.

۲.۳.۱ اسکلت بندی الگوریتم های کوتاه ترین مسیر (بدون دور منفی)

- مرحله Initialize: در این مرحله، $d[v]$ را برای هر راس برابر طول کوتاه ترین مسیر از مبدا s به v تعریف می کنیم.
- مرحله Main: در طول اجرای الگوریتم اجازه میدهیم $d[v]$ بیشتر مساوی مقدار واقعی خود باشد و در انتها با آن برابر شود.
  به این کار آسان کردن شرایط، یا relax کردن میگویند. البته در اصل عمل متضاد relax انجام می شود و شرایط را سخت تر می کنیم!
  relax کردن یال (u, v) با وزن $w(u, v)$ به شکل زیر می باشد:
  اگر $d[v] > d[u] + w(u, v)$ باشد، آنگاه می توان مسیر به v را بهبود بخشید و قرار داد: $d[v] \leftarrow d[u] + w(u, v)$

ترتیب اجرای عملیات relax روی رئوس مهم است و در صورت عدم رعایت، زمان اجرای الگوریتم می تواند چند جمله ای نباشد.

Initialize:
$\forall v \in V$: $d[v] \leftarrow -\infty$, $parent[v] \leftarrow None$ (for root: $d[s] \leftarrow 0$)

Main:
repeat:
somehow select (u, v)
if d[v] > d[u] + w(u, v):
d[v] = d[u] + w(u, v)
parent[v] = u

تفاوت الگوریتم های ذکر شده تنها در خط somehow select (u, v) می باشد.

جزوه جلسه بیست و دوم داده ساختارها و الگوریتم

۱۶ آذر ۱۴۰۰

فهرست مطالب

۱ کوتاه ترین مسیر در گراف جهت دار بدون دور

این الگوریتم، کوتاه ترین مسیر را در زمان $\Theta(n+e)$ پیدا می کند.
مراحل الگوریتم به شکل زیر است:

- راس ها را به ترتیب توپولوژیک می چینیم (ترتیب نزولی درجه رئوس خروجی)
- $\mathrm{d[v]}$ برای همه رئوس مشخص شده است. به ترتیب روی رئوس حرکت می کنیم و یالهای خروجی راسی که روی آن قرار داریم را relax می کنیم. الگوریتم را میتوان به ظور مشابه برای یال های ورودی نیز relax کرد.

۱.۱ اثبات درستی الگوریتم

برای اثبات درستی الگوریتم از استقرا استفاده می کنیم. درستی الگوریتم برای حالت تک راسی واضح است.
برای ادامه اثبات از استقرای قوی استفاده می کنیم. کوتاه ترین مسیر از مبدا s به راس y را در نظر می گیریم و آخرین یال ورودی به y را $(\mathrm{x,\ y})$ می نامیم. طبق فرض استقرا $\mathrm{d[x]}$ مقدار واقعی خود را دارد. حال یال $(\mathrm{x,\ y})$ را ریلکس می کنیم و $\mathrm{d[y]}$ نیز محاسبه می شود.
طبق این اثبات، $\mathrm{d[y]}$ کوچکتر مساوی مقدار واقعی خود است.
اردر اجرای الگوریتم نیز برابر $\Theta(n+e)$ می باشد (حلقه روی همه رئوس و پیمایش یال های متصل به آنها)

۲ الگوریتم بلمن-فورد

در این الگوریتم برای محاسبه کوتاه ترین مسیر وجود یال منفی بدون دور منفی مشکلی ندارد. اما در صورت وجود دور منفی چه کار میتوان کرد؟

۱. می توان ادعا کرد وقتی دور منفی وجود نداشته باشد خروجی الگوریتم صحیح است اما در صورت وجود دور منفی، تضمینی بر صحیح بودن خروجی نیست.
در قسمت های بعدی به دنبال بهبود این حالت هستیم:

۲. فقط وجود دور منفی را گزارش دهیم

۳. رئوسی که به دور منفی ربطی ندارند را خروجی صحیح و بقیه رئوس را $-\infty$ در نظر بگیریم.

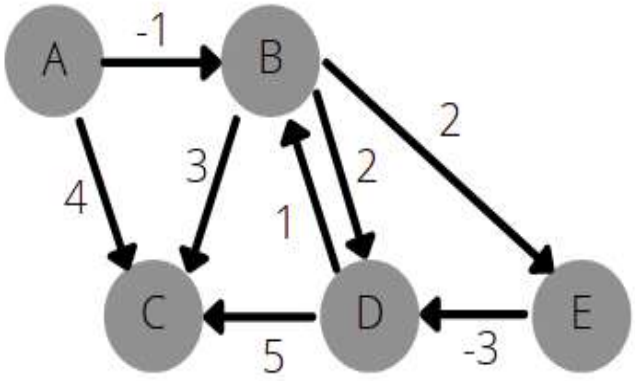
۴. خود دور منفی را گزارش دهیم.

کد مربوط به این الگوریتم نیز قابل مشاهده است.

```
1.  Bellman-Ford:
2.      initialize parents and d arrays
3.      for round in range(n-1):
4.          for edge (u,v) in G:
5.              relax(u, v)
6.      somehow handle negative weight cycles
7.      return parent, d
```

۱.۲ حالت پایه الگوریتم بلمن فورد

در هر یک round حلقه اصلی، هر یال یکبار relax می شود. همچنین حالت پایه خط ششم کد بالا را ندارد.
ترتیب ریلکس کردن یالها نیز مهم است به نحوی که روی اردر زمانی کد تاثیر دارد.



شکل ۱: ترتیب پیمایش یالهای گراف

اگر رئوس به ترتیب AB, AC, BC, BD, BE, DB, DC, ED ریلکس شوند در اولین حلقه و اگر با ترتیب برعکس ریلکس شود در حلقه سوم کوتاه ترین مسیرها پیدا می شوند.
برای بهبود عملکرد الگوریتم تیز می توان بررسی کرد اگر بعد از یک round مسیر ها تغییری نکردند الگوریتم به پایان برسد. برای اثبات درستی این ادعا نیز با استقرای قوی

بعد از راند iام، d[v] کمتر مساوی طول کوتاه ترین مسیر از s به v با حداکثر i یال است.
پس مسیرهایی با حداکثر i یال از مبدا به v پیدا شده اند.
همچنین بعد از راند n-1ام مسیر هایی به طول n-1 نیز بررسی شده اند. پس تمام مسیرها بررسی شده اند و الگوریتم به درستی تمام مسیرها را بررسی کرده است.

۲.۲ روش های مقابله با دور منفی

۱.۲.۲ ادامه الگوریتم برای یک مرحله دیگر

کاری که برای n-1 بار انجام دادیم را برای یک بار دیگر انجام دهیم. اگر حداقل یک یال ریلکس شد و طول مسیز یک واحد کاهش یافت نتیجه می گیریم دور منفی وجود دارد.
همچنین برای پیدا کردن دور منفی باید d[v] همه راس ها را صفر در نظر گرفت.

```
1.  for edge (u,v)vin G:
2.    if d[v] > d[u] + w(u, v):
3.      raise ValueError("There is a negative cycle reachable from s!")
```

۲.۲.۲ گزارش راس هایی که کوتاه ترین مسیر از s یه آنها $-\infty$ است

در این روش، n راند دیگر الگوریتم را ادامه میدهیم و رئوسی که d[v] انها در هر مرحله آپدیت می شود را به عنوان راس $-\infty$ گزارش دهیم.
راه دیگر DFS زدن روی رئوسی است که بعد از یک مرحله اضافه d[v] آنها تغییر کرده است.

۳.۲.۲ پیدا کردن یک دور منفی

در راند nام که یال (u,v) را آپدیت می شود، پس در نتیجه یک دور منفی در مسیر s به v وجود دارد. با استفاده از parent رئوس و پیمایش گراف، به محض دیدن یک راس تکراری دور منفی را گزارش می دهیم. همچنین خود راس v نیر در دور منفی وجود ندارد

۳.۲ کاربردها

- در شبکه و نقشه مسیریابی و همچنین تبدیل رمز و رمز ارز دور منفی نداریم و باید شناسایی شوند.

- یک دستگاه معادلات داریم که شامل m نامساوی به صورت $x_k - x_i \leq b_k$ می باشند. میتوان جواب های x_i را از الگوریتم بلمن فورد استخراج کرد.
هر $x - i$ را یک راس در نظر می گیریم و برای $x_j - x_i$ یک یال جهت دار با وزن b_k در نظر می گیریم. با اجرای الگوریتم باید در نهایت داشته باشیم: $x_j \leq x_i + b_k$.
در ابتدا x_iها صفر و b_kها منفی هستند. در صورتی که دور منفی داشته باشیم، مسئله قابل حل نیست.

جزوه جلسه بیست و سوم داده ساختارها و الگوریتم

۲۱ آذر ۱۴۰۰

فهرست مطالب

۱ الگوریتم دکسترا برای حل مسئله SSSP

در این الگوریتم در گراف یال منفی نداریم. به همین دلیل زمان $\theta(ne)$ الگوریتم بلمن-فورد به $\Theta(e + nlogn)$ (در بهترین حالت پیاده سازی) می رسد.
ایده این الگوریتم در نظر گرفتن یک مجموعه مرزی که شامل رئوسی است که فاصله آنها از مبدا s حداکثر x می باشد و افزایش گام به گام x می باشد. رئوسی که درون مجموعه قرار دارند را relax شده در نظر می گیریم و با افزایش x، رئوس خارج از مجموعه مرزی را نیز relax می کنیم و با ورود همه رئوس به مجموعه مرزی، الگوریتم به پایان می رسد.
حالت پایه این الگوریتم زمانی است که $x = 0$ باشد و تنها مبدا در مجموعه مرزی قرار بگیرد. الگوریتم باید برای این حالت نیز درست کار کند. (فرض می کنیم که یال با وزن صفر نیز نداریم)
اگر وزن همه رئوس یک بود، دقیقا مسئله BFS را داشتیم؛ اما در این مسئله وزن یالها لزوما صحیح نیستند و عمل افزایش x باید با دقت و احتیاط انجام گیرد. به این صورت که افزایش زیاد x باعث ignore شدن بعضی رئوس و افزایش کم آن باعث عدم پایان یافتن الگوریتم در زمان معمول می شود.
بهترین گام افزایش x، فاصله رئوس بیرون مجموعه مرزی از مبدا منهای x است. در این حالت تنها event های مهم (رئوس جدید) پیمایش می شوند. به این منظور نیز در هر مرحله، فاصله مجموعه مرزی تا رئوس باقی مانده را به روز نگه می داریم و آن ها را در یک صف اولویت قرار می دهیم.
شبه کد این الگوریتم در تکه کد زیر آمده است: در گام اول مجموعه مرزی خالی است و

```
def dijkstra(Adj, w, s):
  parent = [None] * len(Adj)  # Same
  parent[s] = s               # init
  d = [math.inf] * len(Adj)   # as
  d[s] = 0                    # before.

  Q = PriorityQueue.build(Item(id=u, key=d[u]) for u in Adj)

  while len(Q) > 0:
    u = Q.delete_min().id     # Delete and process u
    for v in Adj[u]:              # Same
      if d[v] > d[u] + w(u,v):    # relax
        d[v] = d[u] + w(u,v)      # as
        parent[v] = u             # before.
        Q.decrease_key(id=v, new_key=d[v]) # NEW!

  return d, parent
```

شکل ۱: شبه کد الگوریتم دکسترا

با شروع از مبدا s، راس از صف حذف می شود و وارد مجموعه مرزی می شود. همچنین در این الگوریتم هر یال حداکثر ۲ بار و در اردر $\Theta(e)$ یال ها پیمایش می شوند.

۱.۱ پیاده سازی های مختلف الگوریتم دکسترا

۱.۱.۱ پیاده سازی صف اولویت یا **binary heap**

اگر صف اولویت با binary heap پیاده سازی شود، در اینصورت زمان عملیات delete_min و decrease_key هردو در زمان $O(logn)$ انجام میشود و در نهایت به دلیل انجام n عمل delete_min و e عمل decrease_key، این الگوریتم در زمان $O((n+e)logn)$ انجام می شود.
در صورت وجود یال منفی، در هنگام relax کردن با پیمایش مکرر یال منفی، طول مسیر را کوتاه کرد. به همین دلیل وجود یال منفی در این الگوریتم منع شده است.
همچنین اگر گراف جهت دار باشد نیز در صورت وجود یال منفی، تضمینی بر درست بودن الگوریتم وجود ندارد.

۲.۱.۱ پیاده سازی الگوریتم در زمان $O(n^2)$

اگر تعداد یال ها زیاد باشد می توان بدون استفاده از صف اولویت نیز الگوریتم را پیاده سازی کرد. در این حالت با for زدن روی رئوس بیرون مجموعه و لیست همسایه ها می توان در زمان $On^2)$ الگوریتم را پیاده سازی کرد. در این حالت زمان delete_min برابر $O(n)$ و زمان decrease_min نیز برابر $O(1)$ است. پس در نهایت اردر زمانی الگوریتم برابر می شود با $O(n^2)$

۳.۱.۱ پیاده سازی با هرم فیبوناچی

هرم فیبوتاچی نیز یک داده ساختار ارائه شده برای interface هرم است. در این داده ساختار عملیات delete_min در زمان $O(logn)$ (به طور کلی به دلیل کران پایین مرتبط سازی، این عملیات در هر هرمی حداقل از اردر $logn$ می باشد) و decrease_key در زمان سرشکن $O(1)$ انجام می گیرد. پس زمان اجرای الگوریتم برابر می شود با:
$O(n*O(logn)+e*O(1))=O(nlogn+e)$
توجه شود که این مدل پیاده سازی برای n های کوچک به دلیل ضریب بالای عملیات سرشکن مناسب نیست و استفاده از هرم دودویی توصیه می شود. ولی برای n های بزرگ استفاده از هرم فیبوتاچی بهتر است.

۲.۱ اثبات درستی الگوریتم دکسترا

با relax کردن یال ها، d راس ها همواره بیشتر یا مساوی فاصله واقعی آن ها از مبدا است. حال ادعا می کنیم وقتی یک راس به مجموعه مرزی اضافه می شود، d آن نهایی شده است. با اثبات این ادعا، درستی الگوریتم دکسترا نیز ثابت می شود.
با فرض خلف، فرض می کنیم راس w اولین راسی است که با اضافه شدن به مجموعه مرزی، فاصله آن از مبدا نهایی نشده است و صحیح نیست. کوتاه ترین مسیر از s به w را در گراف داده شده در نظر می گیریم. با حرکت از s به w، اولین یالی را در نظر بگیرید که از داخل مجموعه به خارج آن میرویم و اسم آن را (v, u) در نظر می گیریم. چون یال منفی نداریم پس فاصله واقعی u از مبدا کمتر از فاصله واقعی w از مبدا است. d[v] به دلیل قرار گرفتن درون مجموعه مرزی درست محاسبه شده است. همچنین به دلیل relax کردن یال (v, u)، d[u] نیز درست محاسبه شده است؛ پس $d[w] \geq d[u]$
همچنین اگر یال صفر نداشتیم، باید راس u به مجموعه مرزی اضافه می شد. همچنین در صورت وجود یال صفر نیز مشکلی برای الگوریتم به وجود نمی آید؛ زیرا در این صورت فرقی بین u و w به منطور اضافه کردن به مجموعه مرزی وجود ندارد.
$d[u] \geq d[w] \geq dis(s, w) \geq dis(s, u) = d[u]$
به دلیل برابر شدن ابتدا و انتهای نامساوی، پس حالت تساوی همه نامساوی ها رخ داده و در نتیجه: $d[w] = dis(s, w)$ پس فرض خلف نادرست است و اثبات به پایان می رسد.

۳.۱ اجرای هوشمندانه تر الگوریتم دکسترا!

در عمل برای رسیدن به راس مشخص t از مبدا s، شروع کردن از مبدا و بزرگ کردن مجموعه مرزی تا رسیدن به t از لحاظ زمانی مقرون به صرفه نیست. البته در بدترین حالت، برای مسیریابی بین دو نقطه کار بهتری نیز نمیتوان انجام داد. به جای این کار از مبدا در جهت یال ها و از مقصد (t) در خلاف جهت یال ها الگوریتم دکسترا را اجرا می کنیم تا به اولین راس مشترک در مجموعه های مرزی برسیم. کوتاه ترین مسیر لزوما از راس مشترک نمی گذرد و به همین دلیل باید تمام یال های موجود بین ۲ مجموعه را بررسی کنیم تا کوتاه ترین مسیر پیدا شود

جزوه جلسه بیست و چهارم داده ساختارها و الگوریتم

۲۳ آذر ۱۴۰۰

فهرست مطالب

۱ مجموعه های مجزا

مجموعه های مجزا یک interface با عملیات تعریف شده زیر می باشد:

- ایجاد یک مجموعه جدید
- ادغام دو مجموعه
- پیدا کردن مجموعه مربوط به عنصر داده شده

برای مثال در یک گراف، هر راس را در ابتدا می توان درون یک مجموعه مجزا قرار داد که هیچ یالی به هیچ راسی نیز ندارد. در ادامه با اضافه شدن یالها، می توان مولفه های همبندی (ادغام دو مجموعه) ایجاد کرد. به طور کلی یالها یا بین مولفه های همبندی هستند یا درون رئوس یک مولفه همبند. همچنین می توان پرسید هر راس در کدام مجموعه همبندی است (پیدا کردن مجموعه مربوط به عنصر داده شده)
همچنین فرض های ساده سازی زیر را نیز انجام می دهیم:

- به جای نام مجموعه، برای هر مجموعه یکی از عناصرش را به عنوان نماینده در نظر می گیریم.
- عناصر را اعداد 1 تا n فرض می کنیم. (به طبع، ایجاد مجموعه جدید نداریم و همه اعضا را از قبل داریم) پس هر کدام از اعداد 1 تا n مجموعه اختصاصی خود را دارند.

۱.۱ اردر زمانی عملیات تعریف شده با پیاده سازی های مختلف

۱.۱.۱ ذخیره مجموعه مربوط به عناصر در یک لیست

در یک آرایه به طول n (A) ذخیره کنیم که عنصر iام در کدام مجموعه قرار دارد. پس:

- قرار دادن عناصر در مجموعه های اختصاصی خود در $\Theta(n)$ انجام می شود.
- پیدا کردن مجموعه مربوط به یک عنصر در $\Theta(1)$ انجام می شود.
- ادغام دو مجموعه نیز با تغییر A[i] عناصر موجود در مجموعه مبدا در $\Theta(n)$ صورت می گیرد.

۲.۱.۱ نگه داری عناصر هر مجموعه در کنار آرایه قسمت قبل

در این بخش، در کنار آرایه A قسمت قبل، برای هر مجموعه لیست عناصر موجود در آن را ذخیره می کنیم. در این حالت نیز:

- ساختن لیست های اولیه در $\Theta(n)$ انجام می گیرد.
- پیدا کردن مجموعه مربوط به یک عنصر در زمان ثابت $\Theta(n)$ صورت می گیرد.
- برای ادغام ۲ مجموعه α و β، با فرض کمتر بودن تعداد اعضای مجموعه β، به جای for زدن روی کل آرایه A، $\mathrm{A[k]}$ را برای عناصر موجود در β تغییر می دهیم. پس این عملیات در $\|\beta\|$ انجام می شود. (در حالت کلی برای ادغام دو مجموعه α و β زمان $min(\|\alpha\|, \|\beta\|)$ انجام می شود). البته واضح است برای مجموعه های بزرگ، به دلیل allocate کردن حافظه جدید، این عملیات در $O(n)$ انجام می شود.

حال اگر بخواهیم مرتبه زمانی ادغام را دقیق تر تحلیل کنیم، نیاز است مجموع همه ادغام ها را محاسبه کنیم. ادعا می کنیم همه ادغام ها در زمان $O(nlogn)$ انجام می گیرد. زیرا در ابتدا همه عناصر در مجموعه های تک عضوی بودند. در مرحله بعد مجموعه هایی به طول حداقل ۲، بعد از آن مجموعه هایی به حداقل طول ۴ و ... داریم. پس هر عنصر به اندازه $logn$ بار مجموعه مربوط به خودش عوض می شود و برای همه عناصر، زمان کل ادغام ها برابر با $O(nlogn)$ می شود.
پس به صورت سرشکن زمان ادغام ۲ مجموعه برابر با $O(logn)$ می شود.

۳.۱.۱ پیاده سازی با اشاره غیر مستقیم هر عنصر به نماینده مجموعه

برای کاهش زمان ادغام، مجبوریم زمان پیدا کردن مجموعه یک عنصر خاص را افزایش دهیم.
در حالت قبلی دسترسی به مجموعه یک عنصر در $\Theta(1)$ انجام می شد. زیرا هر عنصر مستقیما به نماینده خود اشاره می کرد. اما در این حالت، هر عنصر لزوما به نماینده اشاره نمی کند و به یک عنصر دیگر در مجموعه اشاره می کند. همچنین نماینده نیز به خودش اشاره می کند.
با این فرض، برای ادغام دو مجموعه α و β، کافی است پوینتر α را به β تغییر دهیم که در زمان $\Theta(1)$ انجام می شود.
همچنین با این رویکرد، پیدا کردن مجموعه یک عنصر در بدترین حالت در (n) انجام می شود. برای کوتاه کردن این مسیر می توان از ۲ ایده زیر بهره برد:

۱. فشرده سازی مسیر: هنگامی که از یک مسیر به نماینده رسیدیم، همه عناصر مسیر را به صورت مستقیم به نماینده متصل کنیم.

۲. وصل کردن درخت با مرتبه کمتر به درخت با مرتبه بیشتر: منظور از مرتبه، ارتفاع درخت قبل فشرده سازی است. ریشه درخت نماینده مجموعه و رئوس آن بقیه عناصر مجموعه هستند.

شبه کد مجموعه های مجزا با فشرده سازی مسیر و در نظر گرفتن مرتبه در نکه کد زیر آمده است:

```
1.  def find_set(parent, x):
2.      if x != parent[x]:
3.          parent[x] = find_set(parent, parent[x])
4.      return parent[x]

5.  def union(parent, rank, x, y):
6.      x, y = find_set(parent, x), find_set(parent, y)
7.      if rank[x] < rank[y]:
8.          parent[x] = y
9.      else:
10.         parent[y] = x
11.         if rank[x] == rank[y]:
12.             rank[x] += 1
```

۴ خط اول عملیات فشرده سازی و برگرداندن نماینده و ۸ خط بعد نیز عملیات ادغام را نشان می دهد.
در کد بالا، منظور از parent[x]، عنصری است که x به آن اشاره می کند. همچنین در ابتدا و برای initialize، برای همه عناصر داریم: parent[x] = x , rank[x] = 0
الگوریتم فوق در 1964 معرفی شد و در سال 1973 اثبات شد که زمان اجرای الگوریتم برابر با $O(log^*(n))$ می باشد. برای درک تفاوت log^* و log به مثال زیر توجه کنید:

$$D = 2^{2^{2^{2}}} \implies log(D) = 2^{2^{2}} = 64536 \ , \ log^*(D) = 5$$

در سال 1975 ثابت شد که زمان اجرا از log^* نیز کمتر است و برابر است با $O(m\alpha(n))$ می باشد که تابع α معکوس تابع Ackermann است. برای اطلاعات بیشتر به لینک زیر مراجعه کنید.

https://en.wikipedia.org/wiki/Ackermann_function

در نهایت در سال 1989 اثبات شد که زمان اجرا برابر است با $\Omega(\alpha(n))$

۲ درخت عبارت

عبارات ریاضی به صورت یکتا نمایش داده نمی شوند و نیاز به پرانتز گذاری دارند.
به دلیل خوش ترتیب نبودن عبارت هایی مانند $2 + 3 * 4$، اولویت عملگرها تعریف شد

و تا حدودی مشکل را حل کرد.
برای نمایش یکتای یک عبارت ریاضی، هم میتوان آنرا پرانتر گذاری کرد و هم می توان آنرا در یک درخت نشان داد. راس های درخت می تواند اعداد، متغییرها و عملگرهایی مانند جمع، ضرب، تقسیم، تفریق (برای تفریق دو عدد) و منفی (برای قرینه کردن یک عدد) باشد.
نمایش in-order این درخت ها بدون پرانترگذاری یکسان است و باعث ایجاد ابهام می شود. اما نمایش pre/post-order آنها متفاوت است.

۱.۲ قضیه برای درخت مرتب

درخت مرتب درختی است که فرزندان هر راس ترتیب خاصی دارند که ترتیب ذکر شده نیز مهم است.
طبق این قضیه، اگر نمایش pre/post-order یک درخت را به همراه تعداد فرزندان هر راس داشته باشیم، درخت مدنظر به صورت یکتا مشخص می شود.
همانگونه که ذکر شد هر راس یا یک عملگر است و یا یک متغیر یا عدد. با توجه به اینکه تعداد فرزندان بر اساس نوع راس مشخص می شود، نمایش pre/post-order درخت عبارت ریاضی را بدون نیاز به پرانتربندی مشخص می شود.

۲.۲ نکات تکمیلی درخت عبارت

با استفاده از یک stack machine می توان از روی نمایش pre-order درخت، عبارت را ساخت.
همچنین با استفاده از این درخت می توان عملیات پیچیده مانند مشتق گیری از رابطه مشخص شده راانجام داد.

جزوه جلسه بیست و پنجم داده ساختارها و الگوریتم

۲۸ آذر ۱۴۰۰

فهرست مطالب

۱ داده ساختارهای افزوده یا Augmented Data Structures

در این داده ساختارها، به منظور نگه داری اطلاعات اضافی و همچنین انجام عملیات جدید، تغییراتی در داده ساختار انجام می شود. منظور از نگه داری اطلاعات، ذخیره سازی $f(v)$ از زیردرخت v در راس v می باشد که تابع f نیز یک تابع ساده است.
تابع f برای راس v را اگر بتوان از روی f فرزندان راس v و مقدار خود v محاسبه کرد، عملیات تعریف شده برای درخت نیز در همان زمان قبلی خود انجام می گیرند (معادلا یعنی محاسبه $f(v)$ در $O(1)$ انجام می شود)
توابعی مانند جمع، ضرب، XOR، Min و Max این ویژگی را دارند اما تابعی مانند میانه، چنین ویژگی را ندارد.
اگر ویژگی ذکر شده برای f برقرار باشد، درج، حذف و یا تغییر کلید یک عنصر، تنها روی f راس های مسیر آن راس تا ریشه تاثیر میگذارد. پس به زمان اجرای قبلی عملیات داده ساختار، $\Theta(h)$ نیز اضافه می شود.
یک رویکرد دیگر برای اعمال تغییرات، انجام تغییرات در برگ ها و آپدیت کردن f برای بقیه رئوس است. درخت های AVL و R-B به دلیل کم بودن ارتفاعشان، برای اینکار مناسب هستند.

۱.۱ مسئله جمع زیربازه ها

آرایه A را داریم که در هر Query، جمع عناصر اندیس i تا j را خروجی میدهیم.
ساده ترین راه حل، استفاده از آرایه است. با انجام یک پیش پردازش $O(n)$، آرایه B را می سازیم به نحوی که:

$B[i] = \sum_{j=1}^{i} A[j]$

حال در هر کوئری با دریافت i و j برای پاسخ دادن به مسئله در $O(1)$ داریم:

$B[j] - B[i-1] = A[i] + ... + A[j]$

اگر در حین اجرای الگوریتم، مقادیر آرایه تغییر کند، اجرای الگوریتم در زمان گفته شده انجام نمی شود و کمی پیچیده می شود.
برای رفع این مشکل میتوان آرایه A را در یک درخت ذخیره کرد و $f(v)$ را مجموع کلیدهای زیر درخت v در نظر گرفت. در این پیاده سازی باید قسمت های جدید به کد مربوط به درخت AVL یا R-B اضافه کرد که در عمل چنین چیزی اتفاق نمی افتد. راه حل این مسئله با این رویکرد در جلسه بعد ذکر می شود.

۲ درخت مرتبه آماری یا Order Statistic Tree

در واسط مربوط به این درخت، عملیات زیر قابل تعریف است:

- درج یک کلید
- حذف یک کلید
- پیدا کردن عنصر بعدی: با گرفتن یک کلید، عنصر بعد کلید را خروجی میدهد.
- مرتبه عنصر داده شده: یک کلید ورودی می گیرد و مرتبه آن (چندمین عنصر در آرایه مرتب شده کلیدها) کلید را بر میگرداند.
- انتخاب عنصر با مرتبه خاص: با ورودی گرفتن یک عدد، عنصر با مرتبه ورودی را بر میگرداند.

در این داده ساختار نیاز داریم تا مرتبه عناصر را نیز ذخیره کنیم. بدین منظور تابع f را برابر با تعداد عناصر زیر درخت هر راس تعریف می کنیم.
تنها چالش ما در این داده ساختار، به روز نگه داشتن f بعد از هر درج و حذف است و بقیه عملیات به سادگی انجام می شوند.

۱.۲ پیدا کردن مرتبه عنصر داده شده

از ریشه شروع به حرکت می کنیم و یک متغیر را به صورت $order = 0$ تعریف میکنیم که در نهایت خروجی ماست. با حرکت به سمت راست $order$ را با f زیر درخت چپ بعلاوه یک جمع می کنیم و با حرکت به سمت چپ هیچ کاری نمیکنیم. در نهایت هنگام رسیدن به خود عنصر $order$ را با یک جمع می کنیم و $order$ را خروجی می دهیم.

۲.۲ انتخاب عنصر با مرتبه خاص

به f زیر درخت چپ نگاه می کنیم؛ اگر بیشتر از مرتبه ورودی بود در زیر درخت چپ به دنبال کلید می کردیم و در غیر اینصورت به مقدار f زیر درخت چپ بعلاوه یک را از مرتبه ورودی کم میکنیم و در زیر درخت راست به دنبال کلید می گردیم.

۳ درخت پاره خطی یا Segment Tree

فرض می کنیم که کلیدها اعداد صحیح یک تا m هستند. (لزومی بر برابر m و تعداد عناصر (n) نیست)
برای ایجاد درخت پاره خطی نیز یک درخت دودویی تقریبا کامل ایجاد میکنیم و f را مانند درخت مرتبه آماری برای هر راس نگه داری می کنیم.
حافظه کل مورد نیاز برابر با $4m$ می باشد.

۱.۳ گام های تشکیل درخت و تحلیل آن

برای تشکیل دادن درخت نیز m را تا اولین توان ۲ ادامه میدهیم و آنها را برگ های درخت قرار می دهیم. در گام بعدی با شروع از برگ ها، جمع هر دو راس را به عنوان parent ۲ راس ذکر شده قرار می دهیم.
مقدار f نیز برای هر راس برابر با جمع رئوس متصل به آن می شود.

- نکته قوت: ساختار درخت ثابت است و نیازی به دروان ندارد
- نکته ضعف: زمان هر عملیات زیربازه ای برابر است با $\Theta(logm)$ که برای m های بزرگ بهینه نیست و مطلوب ما $\Theta(logn)$ می باشد. (اگر$m = \Theta(n)$ الگوریتم بهینه است و در غیر این صورت خیر)

۲.۳ پیدا کردن جمع زیربازه

با داشتن a و b به عنوان ابتدا و انتهای بازه، از ریشه شروع به حرکت می کنیم. در حرکت به سمت راست، تمام عناصر سمت چپ عنصر فعلی و در حرکت به چپ تمام عناصر سمت راست راس فعلی در جمع نهایی ظاهر می شوند.
پس تعداد اعدادی که جمع می کنیم برابر است با $2log(m)$ (یک بار حرکت برای پیدا کردن a در $log(m)$ و تکرار عملیات برای b)

۳.۳ آپدیت کردن یک عنصر

برای آپدیت یک عنصر، ۲ رویکرد داریم:

- f همه رئوس از برگ آپدیت شده تا ریشه را تغییر دهیم
- رگ یا راس مدنظر را تغییر دهیم و f ها را مجدد از برگ ها تا ریشه محاسبه کنیم.

برای آپدیت بازه ای نیز برای کاهش مرتبه زمانی از تکنیک Lazy Propagation استفاده می کنیم؛ بدین صورت که هنگام رسیدن به یک راس که تمام فرزاندش تغییر کرده اند، با یک flag این تغییر را نشان می دهیم و به هنگام پایین رفتن از آن راس، در صورت لزوم شروع به تغییر فرزندان می کنیم. در این رویکرد تا حد امکان از تغییر بی مورد f رئوس دوری می کنیم.
با این تکنیک عملیات تعریف شده برای Segment Tree در $\Theta(logn)$ انجام می شود.

۴ درخت فنویک

این درخت برای f های جمع، ضرب و XOR کار میکند و مثلا برای Min و Max کار نمیکند.
برای بازه های ۱ تا یک جای خاص را می توان از این درخت استخراج کرد. به همین دلیل جمع، ضرب و XOR پشتیبانی می شود.
کد پایتون این درخت با فرض اندیس گذاری از صفر در تکه کد زیر آمده است:

```
1.  def sum(index):
2.      result = 0
3.      whild index != 0:
4.          result += array[index]
5.          index -= index & -index
6.      return result

7.  def update(index, add):
8.      while index < len(self.array):
9.          array[index] =+ add
10.         index += index & -index
```

حافظه مصرفی در این حالت نیز از مرتبه n است که به نسبت درخت پاره خطی مقدار کمتری است.

جزوه جلسه بیست و شش داده ساختارها و الگوریتم

۳۰ آذر ۱۴۰۰

فهرست مطالب

۱ مسئله پرسش کمینه یک محدوده

یک لیست به طول n داریم. هر بار عضو کمینه یک بازه از لیست مورد پرسش قرار می گیرد. راه حل های مختلف پوشش یافته در این بخش عبارتند از:

۱. راه حل ساده!

۲. ایده جذر

۳. ایده توان های ۲

۴. درخت پاره ای

۵. راه حل بهینه

۱.۱ راه حل ساده

در راه حل ساده، در هر Query، با یک حلقه از ابتدا تا انتهای بازه و بدون انجام هیچ پیش پردازشی، مقدار مینیمم محاسبه می شود.
پس زمان های زیر را داریم:

- پیش پردازش: $\Theta(1)$
- حافظه اضافی: $\Theta(1)$
- پرسش: $\Theta(n)$
- تغییر عنصر: $\Theta(1)$

۲.۱ ایده جذر

در این راه حل، آرایه را به قسمت هایی به طول $\sqrt{n}$ تقسیم می کنیم و مقدار $\mathrm{minimum}$ را برای هر کدام ذخیره می کنیم. برای پاسخ به هر پرسش نیز با فرض بازه [i, j]، بازه هایی که خود i و j در آن حضور دارند و تمام بازه هایی بین آن ها را بررسی می کنیم و مقدار مینیمم را از بین مینیمم های کاندید انتخاب می کنیم. برای تغییر عنصر نیز، جایگاه آن را در بازه مربوط به خود پیدا میکنیم و عنصر را تغییر می دهیم و در صورت کوچک بودن مقدار جدید از مقدار مینمم، مینیمم را نیز به مقدار جدید تغییر می دهیم.
توجه شود که بازه به طول $\sqrt{n}$ بهینه ترین حالت ممکن برای طول های مختلف است و زمانی بهتر از $\Theta(\sqrt{n})$ نمی توان متصور شد. همچنین ایده استفاده شده در این مسئله بسیار کاربردی است برای زمان های زیر نیز داریم:

- پیش پردازش: $\Theta(n)$
- حافظه اضافی: $\Theta(\sqrt{n})$
- پرسش: $\Theta(\sqrt{n})$
- تغییر عنصر: $\Theta(1)$

۳.۱ ایده توان های ۲

برای هر j در بازه [1, n]، بازه هایی که از j شروع می شوند و طول 2^i دارند ($0 \leq i \leq log_2 n$) را در نظر می گیریم و کمینه آن را در پیش پردازش به شکل زیر محاسبه می کنیم:
$min(A[j, j + 2^i - 1] = min(min(A[j, j + 2^{i-1} - 1]), min(A[j + 2^i, j + 2^i - 1 + 2^{i-1}]))$
با انجام پیش پردازش بالا، پاسخ به هر پرسش در زمان (1) به شکل زیر داده می شود:
$min(A[i, k] = min(min(A[j, j + 2^i - 1]), min(A[k - 2^i + 1, k]))$
توجه داشته باشید که برای رسیدن به پاسخ یک پرسش، می توان بازه را به چند زیر بازه تقسیم کرد و مینیمم آن ها را پیدا کرد در حالی که بازه ها باهم اشتراک داشته باشند. برای مثال اگر طول بازه مورد پرسش برابر با ۱۵ باشد، در نظر گرفتن دو بازه به طول هشت ما را به جواب می رساند. اما باید پیدا کردن طول بازه مد نظر برای هر طول بازه ای در $\Theta(1)$ انجام بگیرد. بدین منظور با انجام یک پیش پردازش دیگر در $\Theta(n)$، برای هر عدد بزرگترین توان ۲ کوچکتر مساوی آن را ذخیره می کنیم. رابطه بالا برای رسیدن به پاسخ هر Query نیز از این پیش پردازش استفاده می کند.
در نهایت زمان عملیات برای این راه حل برابر است با:

- پیش پردازش: $\Theta(nlogn)$
- حافظه اضافی: $\Theta(nlogn)$
- پرسش: $\Theta(1)$
- تغییر عنصر: در این راه حل، تغییر عنصر نداریم.

۴.۱ درخت پاره ای

برگ های درخت را برابر با اعداد آرایه در نظر می گیریم و هر راس درونی برابر است با مینیمم فرزندانش. حافظه اضافی در این راه حل از مرتبه $\theta(n)$ می باشد. پیش پردازش نیز از برگ ها شروع و به ریشه ختم می شود و زمان کل آن نیز برابر است با $\Theta(n)$. پرسش ها نیز در زمان $\Theta(logn)$ (به طور دقیق در $2logn$) انجام می شود.

- پیش پردازش: $\Theta(n)$
- حافظه اضافی: $\Theta(n)$
- پرسش: $\Theta(logn)$
- تغییر عنصر: $\Theta(logn)$

۱.۴.۱ درخت دکارتی

داده ساختار درختی دیگر که می توان از آن استفاده کرد، درخت دکارتی است. مینیمم آرایه کل در ریشه، عناصر سمت چپ آن در زیر درخت چپ و عناصر سمت راس در زیر درخت چپ به صورت بازگشتی درخت را تشکیل می دهند.
با داشتن یک استک می توان درخت را ساخت. آرایه را از چپ می خوانیم و در استک می ریزیم. در هر لحظه اگر عنصر جدید کوچکتر از سر استک بود، عناصر موجود در استک را تا جایی که سر استک کوچک تر از عضو جدید باشد پاک می کنیم و به سمت چپ عنصر جدید اضافه می کنیم و عضو جدید را در استک اضافه می کنیم. اما اگر عضو جدید بزرگتر از سر استک بود، به استک اضافه شده و در سمت راست سر استک قرار می گیرد.
به دلیل نزدیکی این درخت و مسئله Range Minimum Query (RMQ) با داشتن RMQ نیز می توان درخت را ساخت.
همچنین می توان درخت را تشکیل داد و در $\Theta(1)$ به مسئله RMQ پاسخ داد. بدین منظور برای بازه [i, j] ، پایین ترین جد مشترک Lowest Common Ancestor (LCA) دو راس A[i] و A[j] را پیدا می کنیم و به عنوان پاسخ مسئله خروجی می دهیم. تنها چالش باقی مانده پیدا کردن پایین ترین جد مشترک در $\Theta(1)$ است که با انجام پیش پردازش هایی این امر نیز ممکن می شود.

جزوه جلسه بیست و هفتم داده ساختارها و الگوریتم

۵ دی ۱۴۰۰

فهرست مطالب

۱ داده ساختارهای مرتبط با رشته (String)

۱.۱ مسئله تطابق رشته ها

یک رشته بزرگ به نام t و یک رشته کوچک s داریم. میخواهیم s را به عنوان زیررشته ای از t پیدا کنیم.
مسئله فوق یک مسئله کلاسیک در الگوریتم است.
الگوریتم های معروف حل این مسئله عبارتند از:

- الگوریتم KMP در زمان $\Theta(|t| + |s|)$
- الگوریتم بویر-مور که از روی رشته s یک اتوماتا درست می کند و با پیمایش t در آن اتوماتا، وجود s بررسی می شود. زمان اجرای آن نیز $\Theta(|t| + P(|s|))$ می باشد که P یک چندجمله ای می باشد.
- الگوریتم رابین-کارپ با استفاده از درهم سازی، برای هر زیر رشته به طول $|s|$ در t، هش آن را محاسبه می کنیم و در زمان $O(1)$ نیز در رشته t جلو میرویم و هش را مجددا حساب می کنیم. زمان اجرا نیز برابر با $\Theta(|s| + |t|)$ می باشد.

نسخه دیگر از مسئله بالا، به این صورت است که در هر Query، یک رشته s ورودی داده میشود تا در الگوریتم بررسی شود (مشابه کاری که IDEها هنگام indexing پروژه انجام میدهند.)

۲.۱ مسئله دیکشنری

رشته های $T_1, T_2, ..., T_k$ را داریم. پس از انجام پیش پردازش، در هر Query یک رشته s داده می شود و باید جایگاه s خروجی داده می شود. اگر فرض کنیم T_iها به صورت لغت نامه ای (Lexicographical) مرتب شده باشند، رشته قبل s مورد پرسش است: زیرا لزوما s در T_iها نیست.
یک ایده ساده برای حل این مسئله، جست و جوی دودویی ساده است. در این صورت هر پرسش در $O(|s|.logk)$ پاسخ داده می شود. با انجام درهم سازی روی پیشوند کلمات، این زمان به $O(logk.log|s| + |s|)$ نیز کاهش می یابد.
هدف ما رسیدن به زمان $O(log|\Sigma| + |s|)$ می باشد که Σ تعداد کل حروف الفبا می باشد.

۳.۱ ترای و ترای فشرده (Trie)

Trie یک داده ساختار درختی برای حل مسائل ذکر شده می باشد. اسم درخت نیز از کلمه re**Trie**val به معنی بازیابی می آید.

در این درخت، فرزندان هر راس با حروف القبا (Σ) در ارتباط هستند و کلیدها به جای رئوس روی مسیر از ریشه به برگ ها ذخیره می شوند. در انتهای هر رشته نیز کاراکتر $ به منظور نمایش انتهای رشته می آید.

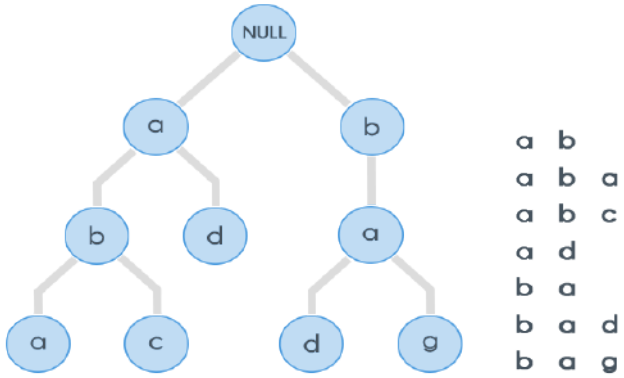


Fig. 1

شکل ۱: مثال یک Trie

درج هر رشته به طول k، در $O(k)$ انجام می شود.
هر جست و جو نیز در $O(n)$ انجام می شود که n طول بزرگترین کلمه در Trie است.
ایراد این داده ساختار حافظه مصرفی زیاد آن در پیاده سازی معمولی است که بیشتر حافظه نیز بلا استفاده می ماند.

۱.۳.۱ فشرده سازی **Trie**

فشرده سازی یکی از راه حل های کاهش حافظه مصرفی می باشد.
در ساده ترین نوع فشرده سازی، فشرده سازی مسیر هایی است که شاخه ندارند. با حفظ ساختار قبلی درخت، فشرده سازی انجام می شود و تعداد فرزندان هر راس تغییری نمی کند و تنها محتویات یال ها تغییر می کند. در صورت نیاز و به هنگام اضافه کردن رشته جدید، این فشرده سازی مجدد به حالت قبلی برمی گردد.
در حالت غیر فشرده مجموع کل یال ها به تعداد حروف رشته ها وابسته است؛ در صورتی که در شکل فشرده، به تعداد رشته ها وابسته است و برای k رشته، تعداد کل یال ها حداکثر $2k-1$ می باشد. تعداد کل راس ها نیز حداکثر $2k$ تا می باشد.
برای حل مسئله دیکشنری با این داده ساختار، با داشتن s، تا جای ممکن در درخت مسیر را از ریشه طی می کنیم. هنگامی که دیگر امکان حرکت نبود، اولین مسیر منتهی

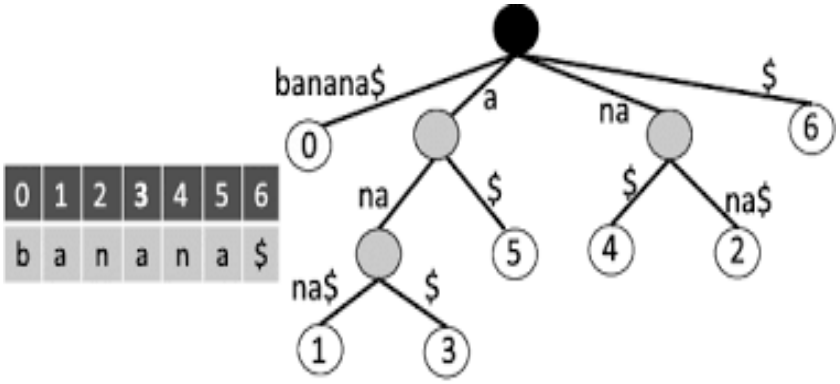
به برگ ($) را خروجی می دهیم. (منظور از اولین راس یعنی در هر راس به اولین راس مراجعه کنیم تا به انتهای رشته برسیم). بدین ترتیب هر Query در زمان $\Theta(|s| + |\Sigma|)$ پاسخ داده می شود.

۲.۳.۱ داده ساختارهای هر راس **Trie**

- آرایه به طول Σ: زمان هر کوئری برابر با $O(|s|+|\Sigma|)$ می باشد اما حافظه مصرفی آن زیاد است.
- لیست مرتبط یا درخت دودویی جست و جو: حافظه این روش کمتر از روش اول است و زمان هر کوئری نیز به $O(|s|.log(|\Sigma|))$ می رسد.
- جدول درهم سازی: حافطه این روش نیز از روش اول بهتر است و زمان هر کوئری نیز مانند روش اول، $O(|s| + |\Sigma|)$ می باشد.
- درخت ون امدبواس: حافظه این روش نیز بهینه است و زمان پاسه به هر کوئری نیز به $O(|s|.loglog|\Sigma|)$ می رسد.
- درخت دودویی جست و جوی متوازن وزن دار: زمان هر پاسخ به هر پرسش برابر با $O(|s|+logk)$ می باشد که این زمان را تا $O(|s|+log|\Sigma|)$ نیز کاهش می یابد.

۴.۱ درخت پسوندی یا **Suffix Tree**

درخت پسوندی، یک ترای فشرده است که شامل همه پسوندهای یک رشته می باشد.



شکل ۲: مثال یک Suffix Tree

این داده ساختار قابلیت حل هردو مسئله ذکر شده در ابتدا را دارد. برای پیدا کردن زیر

رشته s در t، هم زمان با پیمایش درخت و خواندن s، آنرا پیدا می کند.
درخت پسوندی در زمان $\Theta(|t|)$ ساخته می شود!

کاربردهای درخت پسوندی:

- تطابق رشته ها
- در مسئله تطابق رشته ها، به جای پیدا کردن یک مورد خاص، تعداد تکرار و حتی محل آن را گزارش می دهیم.
- طولانی ترین زیر رشته تکراری در یک رشته
- طولانی ترین زیر رشته مشترک kتا زیر رشته (این کار در زمان خطی $O(\Sigma(k_i))$ (جمع طول زیر رشته ها) انجام می شود.) بدین منظور از k رشته داده شده یک رشته به صورت $T_1\$_1T_2\$_2...T_k\$_k$ می سازیم و درخت پسوندی آنرا می سازیم. برای هر راس می فهمیم آیا در صورت ادامه دادن از آن به تمام $\$_i$ها میرسیم. اگر راسی با این شرایط وجود داشته باشد، از ریشه تا آن راس در تمام رشته های T_i وجود دارد. برای کل درخت این عملیات را انجام میدهیم و رشته با طول ماکسیمم را خروجی می دهیم.

۱.۴.۱ آرایه پسوندی یا **Suffix Array**

آرایه پسوندی معادل فشرده و ساده تر درخت پسوندی است. درخت پسوندی را از روی ظاهرش می توان فهمید اما آرایه پسوندی اینگونه نیست.
برای ساختن آرایه پسوندی، ابتدا به هر پیشوند یک رقم از صفر تا $|t|$ میدهیم و سپس پیشوندها را مرتب می کنیم و ایندکس پیشوند های مرتب شده را در آرایه پیشوندی قرار می دهیم.
مثال برای Banana:

$0.Banana\$$
$1.anana\$$
$2.nana\$$
$3.ana\$$
$4.na\$$
$5.a\$$
$6.\$$
$\implies SuffixArray = [6, 5, 3, 1, 0, 4, 2]$

از روی درخت پیشوندی و آرایه پیشوندی، هرکدام را می توان سریع ساخت.
آرایه کمکی دیگری به نام LCP وجود دارد که نشان می دهد در حالت مرتب شده پسوندها که از روی آنها آرایع پسوندی را ساختیم، در چند حرف باهم مشترک هستند.

پیشوندها به صورت متوالی بررسی می شوند.
برای مثال بالا داریم:
$LCP = [0, 1, 3, 0, 0, 2]$
داشتن آرایه LCP باعث می شود حل مسئله تطابق رشته ها از $O(|s|.log|t|)$ به $O(|s| + log|t|)$ برسد.
خود آرایه پسوندی را می توان مستقیما از روی رشته t در زمان $O(|t|.log^2|t|)$ ساخت که این زمان به $O(|t|.log|t|)$ نیز کاهش می یابد.

مسئله تعداد زیر رشته های متفاوت یک رشته:
به کمک آرایه LCP می توان این مسئله را حل کرد. بدین منظور از تعداد کل زیر رشته ها باید تعداد زیر رشته های تکراری را کم کرد. تعداد زیررشته های تکراری نیز برابر با جمع عناصر آرایه LCP می باشد.
تعداد کل زیر رشته های متفاوت یک رشته = $\binom{n}{2}$ - $\Sigma LCP[i]$

۲.۴.۱ تبدیل باروز-ویلر

برای کاهش حافظه مصرفی، با دریافت ورودی، همه rotation های آن را تولید می کنیم و به ترتیب Lexicographical مرتب می کنیم. سپس آخرین حرف هر rotation را ذخیره می کنیم. در خروجی نیز تعداد تکرار حروف ورودی مشخص است. همچنین به جای ذخیره سازی کل حروف، تعداد تکرار را برای هر کدام ذخیره می کنیم.
برای مثال اگر ورودی $Banana\$$ باشد، خروجی به شکل $Bnn\hat{}aa\$a$ می شود.

عکس تبدیل باروز-ویلر:
با داشتن ستون آخر و مرتب سازی آنها، ستون اول به دست می آید. حال با یک شیفت، ۲ ستون اول به صورت نامرتب ساخته می شوند. حال می توان با مرتب کردن آن ها و ادامه این فرایند ورودی اولیه را ساخت.

این الگوریتم به جز رشته ها و متن ها، برای فشرده سازی فایل ها نیز استفاده می شود.

جزوه جلسه بیست و هشتم داده ساختارها و الگوریتم

۷ دی ۱۴۰۰

فهرست مطالب

۱ تحلیل سرشکن

در طول ترم، در مباحثی مانند:

۱. آرایه پویا

۲. جدول درهم سازی پویا

۳. هرم فیبوناچی

۴. مجموعه های مجزا

با تحلیل سرشکن سروکار داشتیم.
تحلیل سرشکن یعنی یک زمان فرضی برای عملیات در نظر می گیریم که برابر با زمانهای واقعی نیستند اما مجموع این زمانها از اول تا آخر به هم ربط دارد.
در اصل، در تحلیل سرشکن، هزینه زیاد یک عملیات را بین عملیات دیگر پخش یا سرشکن می کنیم.
در ادامه، روش های تحلیل سرشکن را بررسی می کنیم.

۱.۱ روش تجمیعی یا انبوهه

هزینه سرشکن هر عملیات، برابر است با مجموع هزینه ها تقسیم بر تعداد عملیات.
اگر بخواهیم کمی پیچیده تر به این حالت نگاه کنیم، می توانیم یک درخت AVL را در نظر بگیریم که هزینه سرشکن درج در آن $O(logn)$ و هزیته سرشکن حذف برابر با 0 است!
می دانیم هزینه واقعی هر دو عملیات برابر با $clogn$ می باشد. درج را در $2clogn$ و حذف را در 0 می توان طبق تعریف بالا انجام داد. به بیان دیگر با فرض خالی بودن درخت در ابتدا، به هنگام درج هر عنصر در درخت، هزینه حذف آن را نیز می پردازیم.
با توجه به مطالب ذکر شده، استفاده از این متد زمانی پیشنهاد می شود که تعداد عملیات تعریف شده برای داده ساختار فقط یکی باشد و با زیاد شدن عملیات، باگ های این متد پیدا می شوند.
به منظور پوشش ایرادات روش قبلی، روش حسابداری معرفی شده است.

۲.۱ روش حسابداری

برای هر عمل، علاوه بر هزینه اصلی خود، یا هزینه بیشتری برای پس انداز در نظر می گیریم و یا هزینه کمتری در نظر می گیریم و از پس انداز استفاده می کنیم. توجه شود که پس انداز نباید منفی شود.

برای مثال درخت AVL ذکر شده در بخش قبل، اگر برای هر درج علاوه بر زمان $clogn$ خودش، زمان $clogn$ را نیز پس انداز کنیم، کافی است اثبات کنیم هزینه حذف را می توان کاملا از پس انداز ها خرج کرد. این ادعا نیز اثبات می شود؛ زیرا هر درج، یک حذف معادل دارد.
همچنین در آرایه پویا می توان هزینه هر درج معمولی بدون نیاز به افزایش اندازه آرایه را چند $O(1)$ در نظر گرفت و هزینه کپی و بزرگ کردن آرایه را از پس اندازها خرج کرد. همچنین باید توجه کرد جمع پس اندازها هزینه کپی و بزرگ کردن آرایه را بدهد. بدین منظور زمان هر درج معمولی را ۳ یا ۵ برابر زمان معمول ($O(1)$) در نظر می گیریم.

۳.۱ روش شارژ کردن

در این روش مقداری از هزینه هر عمل را روی اعمال قبلی شارژ می کنیم. در این حالت، هزینه سرشکن هر عمل برابر است با هزینه واقعی بعلاوه هزینه ای که در آینده رو آن شارژ خواهد شد منهای هزینه ای که روی بقیه شارژ می کند (هزینه را دیگران بپردازند).
برای مثال در آرایه پویا:

- هنگام دو برابر کردن آرایه هزینه واقعی خودش زیاد است و هزینه ای که روی بقیه شارژ می کند نیز زیاد است به همین دلیل هزینه سرشکن آن کم می شود.
- در بقیه موارد نیز هزینه واقعی و هزینه ای که در آینده روی آن شارژ خواهد شد هر دو کم هستند و هزینه سرشکن نیز کم است.

حال وجود چندین عملیات (برای مثال درج و حذف در آرایه پویا) را می توان با این روش تحلیل کرد.

۱.۳.۱ شمارشگر بیتی یا **Binary Counter**

در یک شمارشگر n-بیتی، فرض کنید در state زیر هستیم:
$XX..XX0111...111$
در کلاک بعدی باید در state زیر باشیم:
$XX..XX1000...000$
اگر تعداد یک های حالت اول و صفر های حالت دوم در سمت راست اعداد را m تا فرض کنیم، می توانیم هزینه بیت $m+1$ام را روی بقیه شارژ کنیم تا هزینه شمارش در هر کلاک برابر با $O(1)$ باشد. بدین منظور هزینه هر شارژ را 1 در نظر می گیریم (هزینه اضافی برای هر عملیات یک واحد است). پس برای حالت بالا هزینه شمارش برابر است با:
$2 + m + 1 - (m - 1) = 2$

برای بقیه حالت ها نیز داریم:

$1 + 1 + 0 = 2$

۴.۱ تابع پتاسنیل (قوی ترین روش تحلیل سرشکن)

از تایع پتانسیل برای نامنفی کردن یالهای منفی در الگوریتم دکسترا و همچنین در مجموعه های مجزا استفاده کرده ایم.
تابع پتانسیل Φ تابعی است از وضعیت داده ساختار به اعداد نامنفی.
در این روش هزینه سرشکن برابر است با هزینه واقعی بعلاوه تغییرات تابع پتانسیل ($\Phi_1 - \Phi_0 = \Delta\Phi$ که Φ_1 برابر با مقدار جدید تایع پتانسیل و Φ_0 برابر با مقدار قدیم آن است).
تعریف تایع پتاسنیل نیز ساده نیست.
همچنین برای هر i، مقدار Φ_i باید نامنفی باشد.
اگر مجموع هزینه های واقعی را با $\sum_{i=1}^{n} c_i$ نشان دهیم و مجموع هزینه های سرشکن را با $\sum_{i=1}^{n} c_i + (\Phi_i - \Phi_{i-1})$ نشان دهیم با فرض بزرگتر یا مساوی بودن مجموع هزینه های سرشکن از مجموع هزینه های واقعی، داریم:
$\sum_{i=1}^{n} c_i + (\Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0 \geq \sum_{i=1}^{n} c_i \implies \Phi_n - \Phi_0 \geq 0 \implies \Phi_n \geq \Phi_0$
که با فرض نامنفی بودن Φ_0، برای هر n، Φ_n نامنفی است.
با بررسی توابع پتانسیل پیشنهادی زیر برای شمارشگر بیتی، بهترین تابع پتانسیل برابر است با تعداد کل یک ها در هر لحظه.

- تعداد صفر های ابتدا
- تعداد صفر های کل
- تعداد یک های ابتدا
- تعداد یک های کل

۱.۴.۱ مثال هایی از توابع پتانسیل سرشکن

- شمارشگر بیتی: تعداد کل یک های شمارشگر
- درج عنصر iام در آرایه پویا: $2i - 2^{\lceil log(i) \rceil}$
- مجموعه های مجزا: $\sum \Phi(x)$ for all x
  تابع Φ نیز برای x هایی که ریشه نباشند و یا رنک آنها بزرگتر از صفر باشد به

شکل:

$$(\alpha(x) - level(x)).rank(x) - iter(x)$$

و برای بقیه x ها به شکل:

$$\alpha(x).rank(x)$$

تعریف می شود.