دانشکده مهندسی کامپیوتر

جزوه درس

ساختمان‌های داده

استاد درس: سید صالح اعتمادی [1]

پاییز ۱۳۹۸

[1] تشکر ویژه از آقای میلاد اسفندیاری، آقای سپهر باباپور و خانم زهرا حسینی که پس از جمع‌آوری جزوه‌ها برای تکمیل، یکسان‌سازی و بهبود شکل‌ها، الگوریتم‌ها، قطعه‌های کد، اشکال‌های منطقی، نوشتاری و نگارشی تلاش زیادی کردند. استاد درس مطالب این جزوه را از نظر درستی نکرده کرده است.

فهرست مطالب

۱ آشنایی با الگوریتم ۵
میلاد اسفندیاری فر - ۱۳۹۸/۷/۱

۱.۱ الگوریتم و کاربرد آن چیست؟ ۵

۳ مقایسه الگوریتم ها ۹
محمد امین قسوری جهرمی - ۱۳۹۸/۷/۰۶

۱.۳ مقدمه ۹
۲.۳ بزرگترین مضرب مشترک دو عدد[۱] ۹
۳.۳ نوشتن الگوریتم بدیهی برای GCD ۱۰
۴.۳ الگوریتم اقلیدسی GCD ۱۰
۵.۳ مقدمه ای بر مقایسه الگوریتم ها ۱۱
۶.۳ پیچیدگی زمانی ۱۱
۷.۳ Ω and Big-O ۱۲
۸.۳ سرعت رشد توابع ۱۳

۴ حریصانه ۱۵
زهرا حسینی - ۱۳۹۸/۷/۸

۱.۴ روش های طراحی الگوریتم ها ۱۵
۲.۴ مثال های الگوریتم حریصانه ۱۶

۵ تقسیم و حل ۲۱
فاطمه احمدی - ۱۳۹۸/۷/۱۳

۶ قضیه اصلی و مرتب سازی ۲۴
ملیکا نوبختیان - ۱۳۹۸/۷/۱۸

۱.۶ معیار ارزیابی جزوه ۲۴
۲.۶ Master Theorem ۲۴
۳.۶ شکل کلی قضیه اصلی ۲۵
۴.۶ اثبات قضیه اصلی ۲۵
۵.۶ مرتب سازی ۲۶

[۱] GCD

۶.۶ Selection Sort ۲۶
۷.۶ Merge Sort ۲۸

۷ Divide and Conquer ۳۱
پریسا علائی - ۱۳۹۸/۷/۲۰

۱.۷ مرتب سازی ۳۱
۲.۷ نکات مثال ۳۲
۳.۷ اثبات اینکه n log n بهترین حالت است ۳۲
۴.۷ Example ۳۲
۵.۷ Sorting Small Integers ۳۴
۶.۷ Stable Sort ۳۶
۷.۷ Quick Sort ۳۷
۸.۷ چند نکته : ۳۸
۹.۷ Random Pivot ۳۸
۱۰.۷ Equal Elements ۳۹
۱۱.۷ Sort Visualization ۴۰

۸ Tail Recursion، برنامه نویسی پویا ۴۱
سهراب نمازی نیا - ۱۳۹۸/۷/۲۲

۱.۸ Tail Recursion ۴۱
۲.۸ Intro Sort ۴۲
۳.۸ برنامه نویسی پویا ۴۲
۴.۸ خلاصه ۴۶

۱۱ برنامه ریزی پویا ۴۸
سهند نظرزاده - ۱۳۹۸/۸/۵

۱.۱۱ مقدمه ۴۸
۲.۱۱ مسائل بررسی شده ۴۸

۱۲ برنامه نویسی پویا-ادامه ۶۲
یاسین عسکریان - ۱۳۹۸/۸/۶

۱.۱۲ کوله پشتی ۶۲
۲.۱۲ کوله پشتی گسسته ۶۳
۳.۱۲ قرار دادن پرانتز ۶۷

۱۳ برنامه نویسی پویا — ساختار داده ای آرایه ۷۲
غزل زمانی نژاد - ۱۳۹۸/۸/۱۱

۱.۱۳ مسئله ی پیدا کردن بیشترین مقدار یک عبارت ریاضی با پرانتزگذاری: ۷۲
۲.۱۳ ساختار داده: آرایه ۷۵
۳.۱۳ عملیات اضافه یا حذف کردن یک عنصر در آرایه: ۷۷

۱۴ لیست پیوندی ۷۸

محمد مصطفی رستم خانی - ۱۳۹۸/۸/۱۳

۱.۱۴ لیست پیوندی ۷۸
۲.۱۴ لیست پیوندی یک طرفه: ۷۹
۳.۱۴ درج کردن: ۷۹
۴.۱۴ حذف کردن ۸۰
۵.۱۴ لیست پیوندی دو طرفه ۸۲

۱۵ صف و پشته ۸۵
محمد صدرا خاموشی فر - ۱۳۹۸/۷/۱۸

۱۶ آرایه های پویا و اصل سرشکن کردن ۹۱
هادی شیخی - ۱۳۹۸/۸/۲۰

۱.۱۶ آرایه های پویا ۹۱
۲.۱۶ اصل سرشکن ۹۲

۱۷ صف اولویت دار ۹۴
حسن صبور - ۱۳۹۸/۸/۲۵

۱.۱۷ تعریف ۹۴
۲.۱۷ پیاده‌سازی با استفاده از آرایه‌ی نامرتب ۹۵
۳.۱۷ پیاده‌سازی با استفاده از آرایه‌ی مرتب ۹۵
۴.۱۷ پیاده‌سازی با استفاده از آرایه‌ی نیمه‌مرتب ۹۶
۵.۱۷ heap ۹۷

۱۸ Disjoint Sets ۱۱۲
امید میرزاجانی - ۱۳۹۸/۸/۲۰

۱.۱۸ عملیات ساپورت شده Disjoint Sets ۱۱۲
۲.۱۸ یک مثال از Disjoint Sets ۱۱۳
۳.۱۸ پیاده سازی ۱۱۳

۱۹ ادامه ی Hash Table + Disjoint sets ۱۱۵
آرمین غلام پور - ۱۳۹۸/۹/۲

۱.۱۹ Disjoint sets ۱۱۵
۲.۱۹ Hash Table ۱۱۷

۲۰ جدول هش / hash table ۱۲۰
نگار زین العابدین - ۱۳۹۸/۹/۴

۱.۲۰ نکته هایی از مطالب قبل ۱۲۰
۲.۲۰ function hash ۱۲۰
۳.۲۰ Addressing Direct ۱۲۲
۴.۲۰ collision ۱۲۲
۵.۲۰ Map ۱۲۳
۶.۲۰ Has Key ۱۲۵
۷.۲۰ منابع بیش تر ۱۲۶

۲۲ جدول هش ۱۲۷

محتبی نافذ - ۱۳۹۸/۱۱/۹

۱.۲۲ مقدمه ای بر تابع هش ۱۲۷
۲.۲۲ ویژگی های تابع هش خوب ۱۲۸
۳.۲۲ denied of service attack (DOS) ۱۲۹
۴.۲۲ Universal Family ۱۲۹
۵.۲۲ Load Factor ۱۳۰
۶.۲۲ یک تابع هش universal family برای اعداد ۱۳۰
۷.۲۲ یک تابع هش universal family برای رشته ها ۱۳۱
۸.۲۲ یافتن یک زیر رشته در یک رشته ۱۳۱
۹.۲۲ الگوریتم RabinKarp ۱۳۲

۲۳ Binary Search Trees ۱۳۵
فاطمه امیدی - ۱۳۹۸/۹/۱۶

۱.۲۳ what is Binary Search Tree ۱۳۵
۲.۲۳ Operations ۱۳۵
۳.۲۳ Order ۱۳۷
۴.۲۳ AVL Tree ۱۳۸
۵.۲۳ Splay Tree ۱۳۸

۲۴ درخت AVL ۱۳۹
هژار آزیز - ۱۳۹۸/۹/۱۸

۱.۲۴ مروری بر مباحث جلسه گذشته ۱۳۹
۲.۲۴ مقدمه‌ای بر درخت AVL ۱۳۹
۳.۲۴ درخت AVL و پیاده‌سازی آن ۱۳۹

۲۷ پیمایش در گراف ۱۴۲
ملیکا احمدی رنجبر و رضا علیدوست - ۱۳۹۸/۹/۳۰

۱.۲۷ نمایش گراف و انواع گراف ۱۴۲
۲.۲۷ پیمایش گراف ۱۴۲
۳.۲۷ مولفه های همبند ۱۴۵
۴.۲۷ گراف جهتدار ۱۴۵

۲۹ Topological Sort - SCCs ۱۴۶
سید مصطفی مسعودی - یاسین عسکریان - ۱۳۹۸/۱۰/۰۷

۱.۲۹ مرتب سازی موضعی (Topological Sort) ۱۴۶
۲.۲۹ گراف قویا همبند ۱۴۸

۳۰ درخت ۲-۳ و درخت قرمز-سیاه ۱۵۳
زهرا حسینی - ۱۳۹۸/۱۰/۹

۱.۳۰ درخت ۲-۳ ۱۵۳
۲.۳۰ Red-Black Tree ۱۵۴
۳.۳۰ منابع بیش تر ۱۵۶

۳۱ فهرست دانشجویان ۱۵۸

جلسه ۱

آشنایی با الگوریتم

میلاد اسفندیاری فر – ۱۳۹۸/۷/۱

جزوه جلسه ۱ام مورخ ۱۳۹۸/۷/۱درس ساختمان‌های داده تهیه شده توسط میلاد اسفندیاری فر. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

۱.۱ الگوریتم و کاربرد آن چیست؟

هر برنامه ای تشکیل شده از یک سری کارهای متوالی و پشت سرهم تا نهایتا به هدف برنامه منتهی شود. گاهی این روند ساده است مانند چاپ عدد ۱ تا ۱۰۰ اما گاهی پیچیدگی آن زیاد میشود مانند پیدا کردن یک کلمه در درون یک متن که بسته به طول کلمه و طول متن پیچیدگی آن بیشتر میشود و در این شرایط اهمیت الگوریتم و بهینه بودن آن مشخص میشود مانند سرچ گوگل که در کسری از ثانیه نتایج را پیدا میکند در صورتی که اگر یک فردی که دانش الگوریتم ندارد بخواهد این قابلیت را پیاده سازی کند، زمان بسیار بیشتری از گوگل طول بکشد. در درس ساختمان و طراحی الگوریتم نحوه کار به این شکل است که ورودی و خروجی کاملا مشخص است و صرفا هدف ما سریع انجام دادن کار ها تا رسیدن به خروجی مد نظر است. حال برای شهود بیشتر به الگوریتم و اهمیت آن به یک مثال میپردازیم.

۱.۱.۱ مسئله محاسبه دنباله اعداد فیبوناچی.

همه ما با دنباله زیر اشنایی داریم که به آن دنباله اعداد فیبوناچی میگوییم:

$$1,1,2,3,5,8,13,21,...$$

که فرمول کلی آن به صورت زیر است :

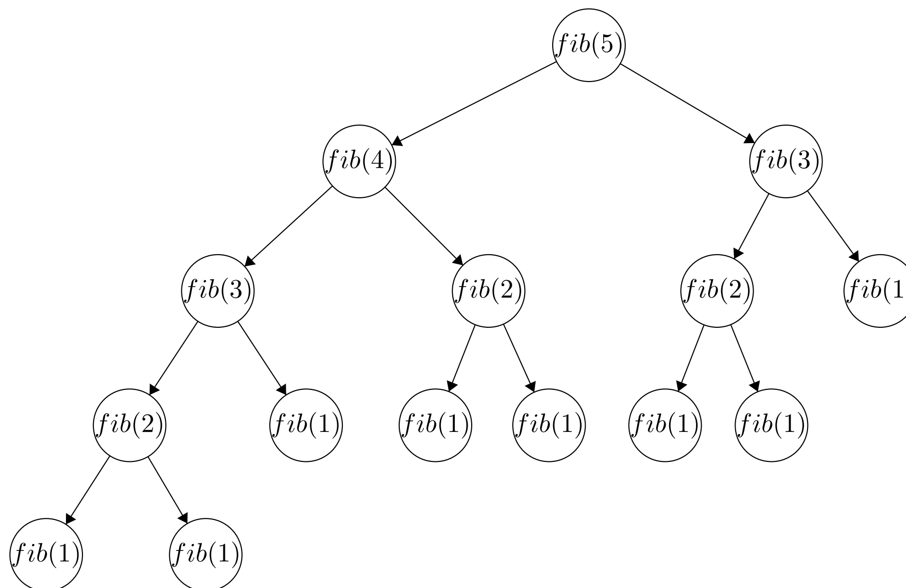
$$f_n = f_{n-1} + f_{n-2} \quad (f_0 = 1, f_1 = 1)$$

برای حل این مسئله ابتدا اولین راه حلی که به ذهنمان بعد از دیدن فرمول محاسبه دنباله فیبوناچی میرسد یعنی راه حل بازگشتی حل میکنیم سپس الگوریتم آن را تحلیل میکنیم و در آخر سعی میکنیم الگوریتم بهینه تری را جایگزین آن کنیم.

آ) **راه حل بازگشتی محاسبه جمله n ام دنباله فیبوناچی** برای حل کردن مسئله فیبوناچی به صورت بازگشتی کافیست فقط حواسمان به پایه تابع بازگشتی باشد که در حلقه بینهایت گیر نکنیم. و مقدار بازگشتی کلی تابع دقیقا مانند خود فرمول آن است. قطعه کد زیر نمونه حل آن در زبان برنامه نویسی سی شارپ است.

```
۱    private static long Fib(long n)
۲    {
۳        if(n == 0 || n == 1)
۴            return 1;
۵        else
۶            return Fib(n - 1) + Fib(n - 2);
۷    }
```

ب) **:بررسی الگوریتم بازگشتی محاسبه جمله n ام دنباله فیبوناچی** بعد از پیاده سازی کد بالا متوجه میشویم که الگوریتم برای n های بزرگ تر بسیار کند میشود تا حدی که از جمله ۵۰ ام در نظر میگیریم محاسبه نمیشود. حال علت آن را از روی درخت بازگشتی روش بالا نشان میدهیم.



شکل ۱.۱: درخت محاسبه جمله پنجم فیبوناچی

شکل ۱.۱ به طور مثال درخت واره روش بازگشتی برای محاسبه جمله ۵ام دنباله فیبوناچی می باشد. باتوجه به شکل میتوان مشاهده کرد که جملات به دفعات متعدد تکراری محاسبه می شوند . به طور مثال جمله ۲ام ۳ بار حساب شده است و علت زمان بالا این روش همین محاسبات تکراری است. برای حل مشکل کافی است مسئله فیبوناچی را به روش بهینه حل کنیم.

ج) **:روش بهینه و سریع تر مسئله فیبوناچی**

کافی است دنباله را یک آرایه n تایی در نظر بگیریم که در هر خانه مقدار n ام دنباله فیبوناچی قرار دارد. و برای مقدار دهی هر خانه کافی است در هر خانه جمع دو خانه قبلی را بریزیم فقط باید توجه داشت که دو خانه اول به مقدار ۱ مقداردهی شده باشند.

قطعه کد زیر راه حل بهینه شده برای مسئله فیبوناچی در زبان برنامه نویسی سی شارپ می باشد.

```
۱    private static long Fib(long n)
۲    {
۳        long[] fib = new long[n+1];
۴        fib[0] = 1;
۵        fib[1] = 2;
۶
۷        for(int i = 2; i <= n; i++)
۸            fib[i] = fib[i-1] + fib[i-2];
۹        return fib[n];
۱۰   }
```

با مقایسه سرعت روش بهینه با روش بازگشتی متوجه اهمیت فوق العاده الگوریتم میشویم

جلسه ۳

مقایسه الگوریتم ها

محمد امین قسوری جهرمی - ۱۳۹۸/۷/۰۶

جزوه جلسه ۳ام مورخ ۱۳۹۸/۷/۰۶ درس ساختمان‌های داده تهیه شده توسط محمد امین قسوری جهرمی.

۱.۳ مقدمه

جلسه قبل در مورد این که الگوریتم چیست و چرا مهم است توضیح دادیم. (مثال فیبوناچی) اگر یادتان باشد دیدیم که که مثال به دست آوردن n امین عدد این دنباله را با دو الگوریتم بدست آوردیم و دیدیم که با الگوریتم بازگشتی از جایی به بعد محاسبات بسیار زیاد میشود و بسیار طول میکشد تا به جواب برسیم. در حالی که با استفاده از الگوریتم دوم قادر بودیم خیلی سریع تر به جواب برسیم.

در این جلسه به بیان دو مطلب می پردازیم. اولین مطلب بیان یک مصداق کمی پیچیده تر است برای اهمیت الگوریتم و وجود الگوریتم های مختلف برای حل یک مسئله و مطلب بعدی هم در مورد روش مقایسه ی دو الگوریتم و بیان معیارمان برای آن ها است که با notation ای به نام big O notation آشنا خواهید شد.

۲.۳ بزرگترین مضرب مشترک دو عدد[۱]

مصداق و مثال دوم پیدا کردن GCD یا ب.م.م دو عدد است. GCD یا ب.م.م دو عدد با بزرگترین مقسوم علیه مشترک دو عدد برابر است. برای مثال GCD دو عدد ۴۵ و ۱۵ برابر ۱۵ است چون ۱۵ بزرگترین عددی است که هم بر ۱۵ هم بر ۴۵ بخش پذیر است. برای بدست آوردن GCD دو عدد مثل ۲۴ و ۱۶ یک راه حل ریاضی بدین شکل دارد: ابتدا اعداد را تجزیه میکنیم : ۲۴ = ۲×۲×۲×۳ و ۱۶ = ۲×۲×۲×۲ و سپس از هر کدام از اجزای مشترک آن ها کمترین توان را بر میداریم. در این مثال جز مشترک عدد ۲ است که در یکی توان ۳ و در دیگری توان ۴ دارد . پس توان ۳ انتخاب می شود یعنی ۸ بزرگترین عدد بخش پذیر بر ۲۴ و ۱۶ است و برابر GCD آن دو عدد است.

[۱] GCD

۳.۳ نوشتن الگوریتم بدیهی برای GCD

خب الان با مفهوم ب.م.م آشنا شدیم ولی برای این که برنامه ای بنویسیم که بتواند ب.م.م حساب کند ما نیاز به الگوریتم داریم نه صرفا تعریف ریاضی. یک الگوریتم بدیهی به شکل زیر است که بگوییم: از عدد ۱ تا عدد کوچکتر بین دو عدد a و b جلو می رویم و برای همه ی آن اعداد محاسبه میکنیم که ایا باقیمانده دو عدد a و b بر آن صفر میشود یا خیر. اگر صفر نشد که ادامه میدهیم و گرنه آن را به عنوان بهترین جواب در آن مرحله در جایی ذخیره میکنیم. بدین ترتیب در آخر عددی که در متغییر می ماند بزرگترین عدد بخش پذیر بر a و b است که همان تعریف GCD است.

```
Data: long a and long b
Result: GCD(a,b)
initialization;
i = 1;
GCD = 1;
while  i < Min(a,b) do
    if Max(a,b) % i equals 0 then
        GCD = i;
    else
        Continue;
    end
end
return GCD;
End;
```

Algorithm 1: Simple Solution For GCD

ولی همیشه بعد از نوشتن الگوریتم از خودمان می پرسیم آیا الگوریتمی بهینه تر وجود دارد یا خیر؟ آیا این الگوریتمی که نوشته‌ام اندازه کافی سرعت بالایی دارد یا خیر؟ یکی از اهداف ما در این درس مقایسه الگوریتم ها و پیدا کردن بهترین است.

۴.۳ الگوریتم اقلیدسی GCD

الگوریتمی بهتر برای حل این مسئله وجود دارد. الگوریتم اقلیدس، روشی موسوم به روش نردبانی یا تقسیمات متوالی برای یافتن بزرگترین مقسوم علیه مشترک دو عدد است که در ادامه، با مثالی آن را شرح می‌دهیم. مثال: برای محاسبه ،GCD(۸۴۶ ۲۰۴) عدد بزرگتر یعنی ۸۴۶ را بر ۲۰۴ تقسیم می‌کنیم و سپس ۲۰۴ را بر باقی ماندهٔ تقسیم قبل تقسیم می‌کنیم و این عمل را تا جایی که باقی مانده صفر شود ادامه می‌دهیم، آخرین باقی‌مانده غیرصفر، بزرگترین مقسوم علیه مشترک دو عدد مزبور است بنابراین ،GCD(۸۴۶ .۲۰۴) = ۶ نکته: ب.م.م هر عددی با ۰ برابر خود آن عدد میشود.

```
A = KB + X
GCD(A,B)= GCD(A,X) = GCD(B,X)
```

در نتیجه الگوریتم ما به شکل زیر می شود در ابتدا عدد بزرگتر را a و عدد کوچکتر را b می نامیم اگر عدد کوچکتر برابر صفر بود عدد بزرگتر ما برابر است با GCD(a،b) وگرنه مراحل زیر را برای GCD(b، a

برای مثال اگر ب.م.م دو عدد ۱۵ و ۶ را بخواهیم حساب کنیم میگوییم جواب مسئله برابر است با ب.م.م عدد ۶ و باقی مانده ۱۵ بر ۶ که میشود ۳ و در ادامه میگوییم جواب مسئله برابر است با ب.م.م عدد ۳ و باقی مانده صفر که میشود ۳.

```
Data: long a and long b
Result: GCD(a,b)
if a < b then
  | Swap(a,b);
end
if b == 0 then
  | return a;
else
  | return GCD(b,a%b);
end
```

Algorithm 2: Fast Algorithm For GCD

این الگوریتم از الگوریتم قبلی ما سریع تر است چون اعداد به سرعت ریز می شوند و محاسبه راحت تر می گردد پس انتخاب الگوریتم بهینه بسیار حائز اهمیت است.

۵.۳ مقدمه ای بر مقایسه الگوریتم ها

دومین مطلب مربوط به مقایسه الگوریتم ها است. سرعت اجرای برنامه ها به چه چیزی وابسته هست. عوامل موثر در سرعت اجرای برنامه قدرت پردازنده یا سرعت دسترسی به حافظه یا ... است. پس زمان اجرای برنامه در کامپیوتر های مختلف فرق دارد پس نمی‌توانیم بر اساس زمان اجرای الگوریتم بر روی کامپیوترها آن ها را مقایسه کنیم و نیاز به معیار بهتری دارد. شاید بر روی کامپیوتری که قدرت بیشتری دارد سریعتر اجرا شود در حالیکه در کامپیوتری با پردازنده ضعیف تر زمان اجرای برنامه بیشتر است الگوریتم‌های مختلفی برای حل یک مسئله ممکن است طراحی شده باشند. برای انتخاب بهترین الگوریتم باید معیاری جهت مقایسه کارایی الگوریتم‌ها داشته باشیم. آنالیز کارایی یک تخمین اولیه‌است با دو معیار سنجیده می شود :

- پیچیدگی حافظه [۲]
- پیچیدگی زمانی [۳]

۶.۳ پیچیدگی زمانی

زمان اجرای یک برنامه به موارد زیر بستگی دارد:

- سخت‌افزار
- سیستم‌عامل
- کامپایلر

---

[۲] complexity space
[۳] complexity time

- نوع الگوریتم
- آرایش داده‌های ورودی

زمان اجرای برنامه‌ها به صورت رابطه بین بزرگی سایز ورودی و زمان مورد نیاز برای پردازش ورودی است. زمان اجرا یکی از ملاک‌های مقایسه چند الگوریتم برای حل یک مسئله می‌باشد. منظور از واحد زمانی، واحدهای زمانی واقعی مانند میکرو یا نانو ثانیه نمی‌باشد بلکه منظور واحدهای منطقی است که رابطه بین بزرگی (n) یک فایل یا یک آرایه و زمان مورد نیاز برای پردازش داده‌ها را شرح می‌دهد. (توجه کنید که هر دستور یک واحد زمانی اشغال می‌کند)

مثلاً دستورهای ؛a=b c/d=e هر کدام یک واحد زمانی را دربردارند. بنابراین تعداد مراحل برای هر عبارت یک برنامه بستگی به؛ نوع عبارت دارد، بطوریکه در عبارات توضیحی برابر صفر و در دستور انتسابی بدون فراخوانی برابر یک می‌باشد؛ و در دستورهای غیربازگشتی حلقه ،for ،while repeat until به تعداد تکرار حلقه در نظر گرفته می‌شود. هدف از محاسبه پیچیدگی زمانی یک الگوریتم این است که بفهمیم نیازمندی یک الگوریتم به زمان با چه تابعی رشد می‌کند و هدف اصلی بدست آوردن این تابع رشد است. برای مثال هرچه زبان برنامه‌نویسی به زبان ماشین نزدیک تر باشد، برنامه با سرعت بیشتری به جواب خواهد رسید زمان اجرا مقدار زمانی از کامپیوتر است که برنامه برای اجرای کامل مصرف می‌کند. برای محاسبه پیچیدگی زمان الگوریتم ابتدا تعداد قدم‌های الگوریتم به صورت تابعی از اندازه مسئله مشخص می‌شود، برای انجام این کار تعداد تکرار عملیات اصلی الگوریتم محاسبه می‌شود و به صورت تابع f بیان می‌شود. سپس تابع ،g که مرتبه بزرگی تابع f را وقتی اندازه ورودی به اندازه کافی بزرگ است نشان می‌دهد، بدست می‌آید. در نهایت پیچیدگی الگوریتم برای نشان دادن رفتار الگوریتم با ورودی‌های مختلف با استفاده از نمادها ،O ⍰ وΩ که در بخش بعدی با آن‌ها آشنا می‌شویم، بیان می‌شود.

برای مثال:

```
int func(int n)
int i;
int sum=0;
for (i=1;i<=n;i++) sum=sum+i;
return sum;
```

عبارت مساوی T(N) = ۲n + ۳ می‌شود. همان‌طور که مشاهده می‌کنید زمان اجرای هر عبارت جایگزینی یا محاسباتی را مساوی ۱ واحد زمانی فرض می‌کنیم. هم چنین دستور داخل حلقه n بار انجام می‌شود ولی آزمایش کردن شرط حلقه در خط for به تعداد n+۱ بار صورت می‌گیرد. دستور Return نیز مساوی یک واحد زمانی است.

۷.۳ Ω and Big-O

برای نمایش پیچیدگی الگوریتم‌ها از تعاریف زیر استفاده می‌شود:

Big-O (حدبالا) تابع f(n) را برای n≥۰ در نظر بگیرید. می‌گوئیم(((O(g(n) = f(n است اگر ثابت مثبت و حقیقی c و عدد صحیح و غیر منفی N وجود داشته باشند به طوریکه به ازای تمام مقادیر :n≥N f(n)≤cg(n) برقرار باشد. این نماد حد بالائی برای تابع f(n) می‌دهد و وقتی بکار می‌رود که رفتار الگوریتم بدترین حالت و بیشترین زمان اجرا را برای مقادیر معین ورودی دارد. برای مثال داریم: O(n۳). = n۲ = F(n) به ازای N=۱ همیشه رابطه n۲ < cn۳ برقرار است یا O(n۲). = ۴ + ۳n + n۲ = F(n) در نتیجه داریم> ۸n۲ n۲ + ۳n۲ + ۴n۲ > n۲ + ۳n + ۴ و به ازای N > ۱ و C = ۸ در نتیجه از O(n۲) است. در نتیجه

برای پیدا کردن اوردر یک عبارت بزرگترین جمله را از لحاظ رشد در نظر میگیریم. در مثال ۳-۱ که یک کد ساده را بررسی کردیم عبارت مساوی T(N) = ۲n + ۳ شد و میتوانیم بگوییم که از O(n) است. توجه داشته باشید که مثلا n۲ هم O(n۳) هم O(nn) یا O(n۲) ... است.

امگا/Ω (حدپائین) بر عکس big O notation تابع f(n) را برای n≥۰ در نظر بگیرید. می‌گوئیم f(n) = Ω(g(n)) اگر ثابت مثبت و حقیقی c و عدد صحیح و غیر منفی N وجود داشته باشند به طوریکه به ازای تمام مقادیر n≥N: f(n) ≥ c(g(n)) برقرار باشد. این نماد حد پائینی برای تابع f(n) می‌دهد و وقتی بکار می‌رود که رفتار الگوریتم بهترین حالت و کمترین زمان اجرا را برای مقادیر معین ورودی دارد.

برای مثال داریم: F(n) = n۲ = Ω(n). به ازای N=۱ همیشه رابطه cn < n۲ برقرار است. توجه داشته باشید که مثلا n۲ هم Ω(۱) یا Ω(n) ... است.

۸.۳ سرعت رشد توابع

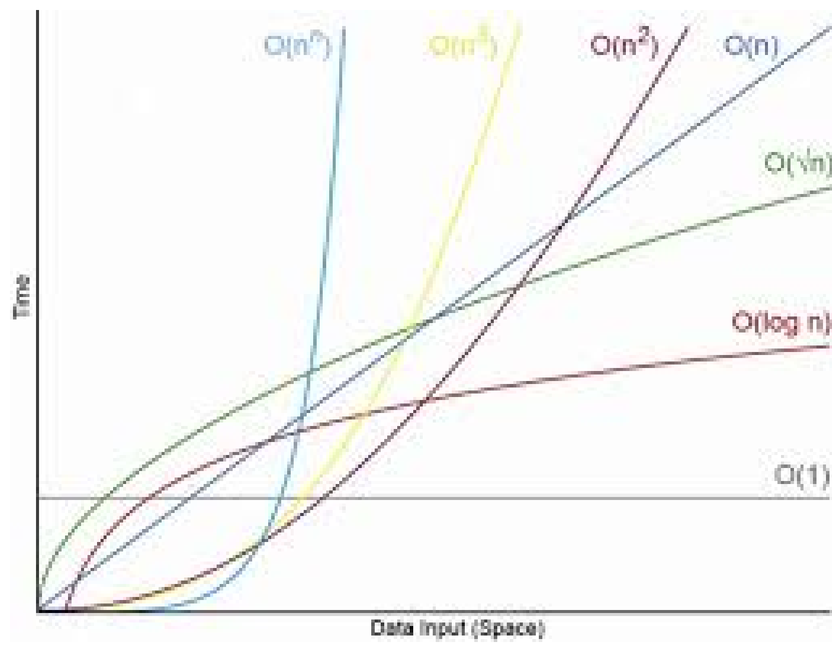
همانطور که گفتیم برای بدست آوردن اوردر یک عبارت باید بزرگترین عبارت را از لحاظ رشد پیدا کنیم یعنی زمانی که ورودی ما بزرگ می شود کدام توابع سریعتر و کدام کند تر پیش میروند: رشد توابع بصورت زیر است:

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84 x 10^{19}

شکل ۱.۳: جدول رشد توابع

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n$$

شکل ۲.۳: سرعت رشد توابع



شکل ۳.۳: نمودار سرعت رشد توابع

جلسه ۴

حریصانه

زهرا حسینی – ۱۳۹۸/۷/۸

جزوه جلسه ۴ام مورخ ۱۳۹۸/۷/۸ درس ساختمان‌های داده تهیه شده توسط زهرا حسینی. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

۱.۴ روش های طراحی الگوریتم ها

در ادامه ی بخش اول،آشنایی کلی با الگوریتم،هدف این است که مسئله داده شده را با بهترین راه ،حل کنیم. سه روش کلی که به ترتیب زیر به بررسی هریک میپردازیم:

- الگوریتم حریصانه
- الگوریتم تقسیم وحل
- الگوریتم برنامه نویسی پویا

۱.۱.۴ حریصانه

در این روش مسئله قدم به قدم کوچک میشود.روش کوچک شدن با یک انتخاب صورت میگیرد،باید توجه کرد که جواب بهینه حذف نشود.

۲.۱.۴ تقسیم و حل

مسئله را به دو یا چند قسمت تقسیم میکنیم،هر زیر مسئله را مستقل از دیگری حل میکنیم در نهایت با توجه به صورت سوال جواب های به دست آمده را باهم مقایسه ، جمع و ... میکنیم.باید دقت کرد که زیرمسئله ها

هم پوشانی نداشته باشند.این روش معمولا به صورت بازگشتی انجام میشود .

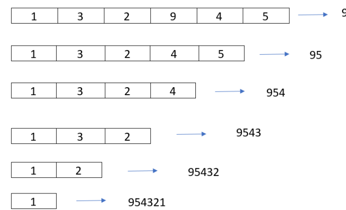
۳.۱.۴ برنامه نویسی پویا

روش همچون روش تقسیم و حل بر پایه‌ی تقسیم مسئله بر زیرمسئله‌ها کار می‌کند. داده‌های زیرمسئله وابسته به هم بوده و یا با هم اشتراک دارند یا به عبارتی هم‌پوشانی دارند. در این روش هر زیر مسئله یکبار حل میشود یعنی حافظه ایی داریم که پاسخ های به دست آمده را ذخیره میکند و دیگر زیرمسئله ایی دوباره حل نمی شود.

۲.۴ مثال های الگوریتم حریصانه

۱.۲.۴ حداکثر حقوق

ورودی آرایه ایی از اعداد است ۱،۳،۲،۹،۴،۵ با این اعداد بیشترین حقوق پیشنهادی خود را تولید کنید.
به روش حریصانه بزرگترین عدد موجود را پیدا میکنیم و آن را درچپ ترین جایگاه قرار میدهیم ،بزرگترین عدد ۹ است آن را انتخاب میکنیم حال یک انتخاب صورت گرفته و مسئله کوچک میشود در واقع عدد انتخاب شده را در هر مرحله از آرایه ورودی حذف میکنیم.



شکل ۱.۴: حداکثر حقوق

اثبات اینکه جواب بدست آمده بهترین جواب است از برهان خلف استفاده میکنیم به این صورت که عدد ۹ اگردر جایگاهی به غیر از چپ ترین جایگاه قرار بگیرد با جا به جا کردن عدد ۹ مقدار بیشتری به دست می آِید پس فرض خلف باطل است.

```
Input: Digits
Output: answer
Function MaxSalary(Digits):
    answer ← emptystring
    while Digits is not empty do
        maxDigit ← −∞
        foreach digit ∈ Digits do
            if digit ≥ maxDigit then
                maxDigit ← digit
        end
        maxDigit append to answer
        remove maxDigit from Digits
    end
    return answer
End
```

Algorithm 3: Maximizing Your Salary

۲.۲.۴ حداقل انتظار بیماران

افرادی در مطب پزشک منتظر هستند،هر کدام از این افراد مدت زمان متفاوتی با پزشک ملاقات میکند،چگونه میزان منتظر بودن هریک از افراد کمترین مقدار ممکن میشود؟ نفر اول انتظاری نمیکشد،نفر دوم به اندازه مدت زمانی که نفر اول نزد پزشک است منتظر میماند ،نفر سوم به اندازه مجموع این زمان برای نفر اول و دوم و به همین ترتیب نفر آخر به اندازه مجموع زمان های افراد به جز زمان خودش منتظر خواهد ماند.

$$t_1 * (n-1) + t_2 * (n-2) + ... + t_{n-1} * (1)$$

همانطور که مشخص است ضریب نفر اول(تعدادی که زمان نفر اول محاسبه میشود) ،بزرگترین ضریب است پس باید نفر اول با کمترین زمان انتخاب شود و انتخاب ها به صورت صعودی باشند یعنی نفر آخر بیشترین زمان را داشته باشد.

برای اثبات درستی راه حلمان از برهان خلف استفاده میکنیم در این حالت

$$t_i$$

ایی را فرض میکنیم و آن را با

$$t_{i+1}$$

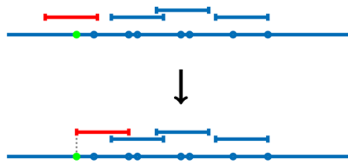
جا به جا میکنیم، اگر فرض ابتدایی ما مبنی بر بهینه نبودن جوابمان درست باشد باید

$$t_{i+1} - t_i$$

منفی شود ولی میدانیم که زمان ها را به ترتیب صعودی مرتب کرده ایم پس این اختلاف مثبت بوده و فرض خلف باطل است.

۳.۲.۴ جشن تولد

افراد حاضر در یک جشن تولد را میخواهیم به گونه ایی دسته بندی کنیم که اختلاف سنی هر یک از افراد گروه با سایر اعضا حداکثر دو سال باشد.همچنین قصد داریم برای هر گروه یک مربی استخدام کنیم، هدف در این مسئله پیدا کردن کمترین تعداد مربی ممکن است. ابتدا افراد را از کوچکترین سن به بزرگترین سن مرتب میکنیم،سپس اولین نفر را انتخاب میکنیم و تا جایی که اختلاف سنی اولین نفر و آخرین نفر حداکثر دو سال باشد ، انتخاب کردن را ادامه میدهیم.حال نفر بعدی که انتخاب میشود اولین نفر گروه بعد و مانند مرحله قبل ادامه میدهیم.



شکل ۲.۴: جشن تولد

فرض کنید

$$x_1 \leq x_2 \leq ... \leq x_n$$

Input: points=

$$x_1, ..x_n$$

Output: *Segments*
Function *PointsCoverdSorted(points)*:
  $Segments \leftarrow emptylist$
  $left \leftarrow 1$
  while $left \leq n$ **do**
    $(l, r) \leftarrow (xleft, xleft + 2)$
    $(l, r) appendtoSegments$
    $left \leftarrow left + 1$
  end
  return *Segments*
End

Algorithm 4: Birthday Party

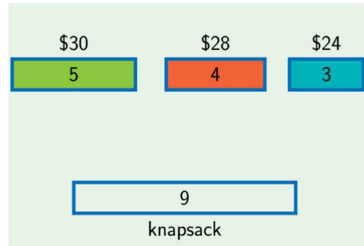
انتخاب بهینه انتخابی است که چپ ترین نقطه را پوشش دهد و بازه ایی باشد که با آن نقطه آغاز شود.

۴.۲.۴ کوله پشتی

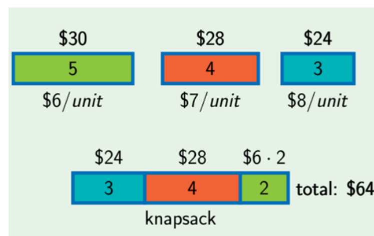
فرض کنید میخواهیم از یک فروشگاه دزدی کنیم(!)،کوله پشتی ایی با ظرفیت مشخص داریم،انتخاب های ما به چه صورتی باشد که بیشترین درآمد حاصل شود؟ با توجه به اینکه اشیا دارای حجم متفاوتی هستند،باید اشیایی را انتخاب کنیم که نسبت به حجمشان ارزش بیشتری داشته باشند.برای اشیا خاصیت چگالی را به این صورت که میزان ارزش تقسیم بر حجم شی تعریف میکنیم حال اشیا را بر اساس چگالی اشان به صورت نزولی مرتب

میکنیم و انتخاب را تا جایی که ظرفیت کوله پشتی اجازه میدهد ادامه میدهیم. فرض کنید طبق شکل زیر کوله پشتی ایی با ظرفیت ۹ و اشیایی با اطلاعات زیر در اختیار داریم

حال برای هریک از اشیا چگالی تعریف میکنیم،ابتدا شی که بیشترین چگالی را دارد اگر حجمش از ظرفیت داده شده تجاوز نکند به طور کامل انتخاب میشود و بقیه انتخاب ها هم مانند مرحله ی قبل برای حجم باقی مانده صورت میگیرد.باید به این نکته توجه داشت که میتوان برای تکمیل کردن ظرفیت کوله پشتی بخشی از شی را انتخاب کرد.



شکل ۳.۴: کوله پشتی۱



شکل ۴.۴: کوله پشتی۲

```
Input: W,weights=[w_1, ..w_n], values = [v_1, ..v_n]
Output: Segments
Function KnapsackFast(W, weights, values):
    amounts ← [0, 0, .., 0]
    totalValue ← 0
    for i ← 1ton do
        if W is 0 then
            return (totalValue, amounts)
        end
        a ← min(w_i, W)
        totalValue ← totalValue + a ∗ (v_i ÷ w_i)
        w_i ← (w_a)
        amounts_i ← (amounts_i + a)
        W ← W − a
    end
    return (totalValue, amounts)
End
```

Algorithm 5: Knapsack

جلسه ۵

تقسیم و حل

فاطمه احمدی - ۱۳۹۸/۷/۱۳

یکی از مسائلی که در آن از الگوریتم تقسیم و حل استفاده می شود مسئله پیدا کردن مینیمم است که به صورت زیر انجام می شود:

```
Data: Ages of Students
Result: how to write algorithm to find youngest student
initialization;
MinMin(nums,low,high) if low = high then
 |  return nums(high);
end
mid <- low + (high - low)/2;
return Math.Min(Min(nums,low,mid),Min(nums,mid,high));
```

Algorithm 6: How to find Min with divide and conquer

پیچیدگی زمانی الگوریتم فوق از رابطه زیر پیروی می کند:

```
T(n) = 2T(n/2) + c
```

توجه کنید که الگوریتم فوق الگوریتم از نوع تقسیم و حل است و مسئله پیدا کردن مینیمم دارای جوابی با الگوریتم حریصانه نیز هست.

یکی دیگر از مسائلی که با الگوریتم تقسیم و حل قابل حل است مسئله تعیین بودن یک آیتم در یک گروه از آیتم ها است که در زیر بررسی شده است:

```
Data: Array of Numbers
Result: v is in Array or not ?
initialization;
FindFind(v,nums,low,high)
if high < low then
 |  return false;
end
mid <- low + (high - low)/2;
if nums(mid) = v then
 |  return true;
end
return (Find(v,nums,low,mid - 1) or Find(v,nums,mid + 1,high))
```

Algorithm 7: Checking v is in the Array or not

پیچیدگی زمانی الگوریتم فوق نیز از رابطه زیر پیروی می کند:

```
T(n) = 2T(n/2) + c
```

الگوریتم تقسیم و حل بیشترین کاربرد را در آرایه های مرتب شده دارد در این حالت الگوریتم در پیچیدگی زمانی log(n) انجام می شود.

در شبه کد زیر مسئِله قبل را با آرایه مرتب شده بررسی میکنیم:

```
Data: SortedArray of Numbers
Result: v is in Array or not ?
initialization;
FindFindSortedArray(v,nums,low,high)
if high < low then
 |  return false;
end
mid <- low + (high - low)/2;
if nums(mid) = v then
 |  return true;
end
if v < nums(mid) then
 |  return FindSortedArray(v,nums,low,mid);
else
 |  return FindSortedArray(v,nums,mid + 1,high);
end
```

Algorithm 8: Checking v is in the SortedArray or not

پیچیدگی زمانی الگوریتم فوق نیز از رابطه زیر پیروی می کند:

```
T(n) = T(n/2) + c
```

مسئله ضرب دو چند جمله ای نیز با الگوریتم تقسیم و حل قابل بررسی است واین روش نیز خود به دو

حالت ساده و سریع انجام می شود که با استفاده از روش سریع پیچیدگی محاسباتی کم می شود به طوری که به جای ۴ عملیات ضرب با ۳ عملیات ضرب به جواب می رسیم و رابطه اول زیر به رابطه دوم تبدیل می شود این در حالی است که پیچیدگی در حالت عادی در زمان n^2 انجام می شود:

```
T(n) = 4T(n/2) + kn
T(n) = 3T(n/2) + kn
```

روش اول ضرب را به صورت زیر انجام می دهد:

```
A(x) = a1x + a0
B(x) = b1x + b0
C(x) = a1b1x^2 + (a1b0 + a0b1)x + a0b0
```

روش سریع نیز به صورت زیر انجام می شود:

```
C(x) = a1b1x^2 + ((a1 + a0)(b1 + b0) - a1b1 - a0b0)x + a0b0
```

جلسه ۶

قضیه اصلی و مرتب سازی

ملیکا نوبختیان - ۱۳۹۸/۷/۱۸

جزوه جلسه ۶ام مورخ ۱۳۹۸/۷/۱۸ درس ساختمان‌های داده تهیه شده توسط ملیکا نوبختیان. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

۱.۶ معیار ارزیابی جزوه

معیارهای مورد استفاده برای ارزشیابی کیفیت جزوه به شرح زیر است:

- پوشش مطالب
- رعایت قواعد نگارشی دستور زبان فارسی
- استفاده از اشکال مناسب
- اشاره به منابع کمک آموزشی

۲.۶ Master Theorem

در تحلیل و واکاوی الگوریتم ها،قضیه اصلی یک راه حل سر راست برای بسیاری از الگوریتم های تقسیم و حل که بازگشتی هستند ارائه می کند.الگوریتم هایی را که می توان به شکل :

$$T(n) = aT([\frac{n}{b}]) + O(n^d)$$

نوشت،با بسط دادن می توان کار کل انجام شده آن را بدست آورد اما قضیه اصلی به ما اجازه می دهد به آسانی پیچیدگی زمانی این الگوریتم های بازگشتی را بدون بسط دادن حساب کنیم.

۳.۶ شکل کلی قضیه اصلی

اگر الگوریتمی را بتوان به شکل زیر نوشت،پیچیدگی زمانی آن به راحتی حساب می شود:

$$T(n) = aT([\frac{n}{b}]) + O(n^d)$$

$$a > 0, b > 1, d >= 0$$

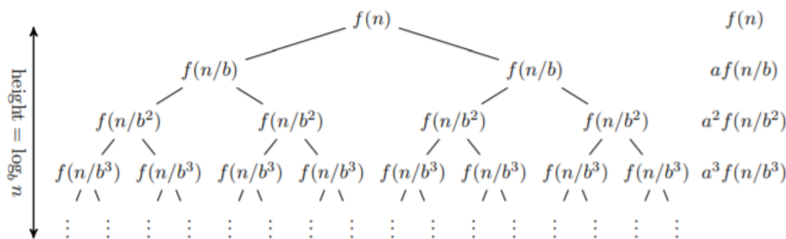
در اینجا aتعداد زیر مسئله ها، n/bاندازه هر زیر مسئله و n^dمقدار کاری است که باید انجام دهیم تا این زیر مسئله ها را با هم جمع کنیم. شکل کلی:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

شکل ۱.۶: Master Theorem

۴.۶ اثبات قضیه اصلی

ابتدا درخت الگوریتم مورد نظرمان را که به دلیل رابطه بازگشتی ساخته می شود را می کشیم.درخت ما عمق$log_b n$ را دارد. a تعداد فرزندان هرگره درخت را نشان می دهد و در عمق i از درخت a^i گره داریم. پیچیدگی زمانی هر گره n/b^i است. پیچیدگی زمانی کلی الگوریتم برابرجمع پیچیدگی های زمانی همه ی گره ها است. بنابراین در این درخت$n^{\log_b a}$ برگ داریم پس پیچیدگی زمانی کلی برگ ها $O(n.\log_b a)$ می شود.حالا می توانیم با توجه به پیچیدگی زمانی $O(n^d)$ پیچیدگی زمانی کلی الگوریتم را حساب کنیم



شکل ۲.۶: اثبات قضیه اصلی

$$
\begin{aligned}
O(n^d) + a.O(n/b)^d + ... + a^i.O(n/b^i)^d + ... + a^{\log_b n} &= \\
O(n^d) + O(n^d)(a/b^d) + ... + O(n^d)(a/b^d)^i + ... + O(n^{\log_b a}) &= \\
\sum_{i=0}^{\log_b n} O(n^d)(a/b^d)^i
\end{aligned} \tag{۱.۶}
$$

۵.۶ مرتب سازی

چرا مرتب سازی مهم است؟

- مرتب سازی اطلاعات یک قدم مهم در بسیاری از الگوریتم های کارامد است.
- اطلاعات مرتب شده اجازه جست و جوهای کارآمدتر را به ما می دهد.

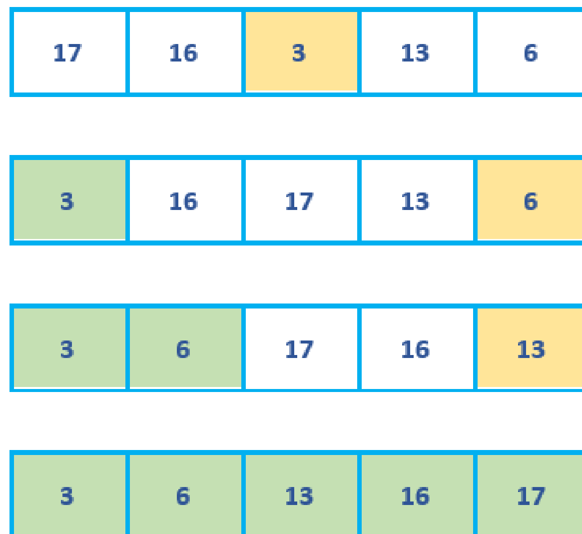
۶.۶ Selection Sort

مرتب سازی انتخابی یکی از ساده ترین الگوریتم های مرتب سازی است.پیچیدگی زمانی آن $O(n^2)$است.
مرتب سازی انتخابی شامل سه مرحله است:

- پیدا کردن کوچک ترین عدد با بررسی آرایه
- جا به جا کردن آن عدد با عنصر اول آرایه
- تکرار کردن همین روش با قسمت باقیمانده آرایه

```
Data: A[1....n]
Result: Sorted Array of A
initialization;
for i from 1 to n do
    minIndex ← i;
    for j from i+1 to n do
        if A[j]<A[minIndex] then
            minIndex ← j;
        end
        Swap(A[i],A[minIndex]);
    end
end
```

Algorithm 9: Selection Sort

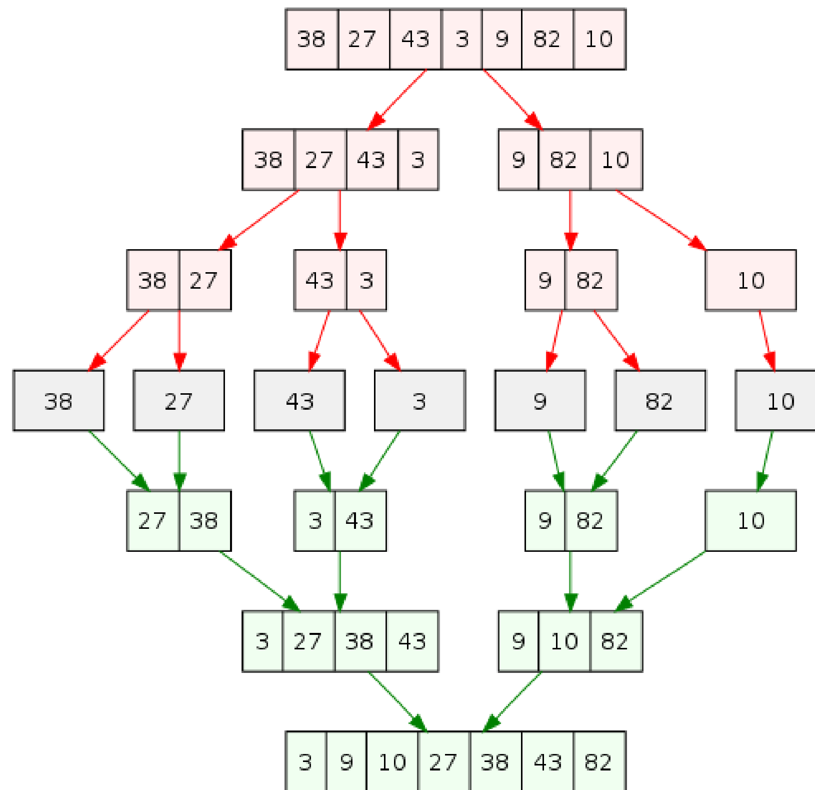


شکل ۳.۶: selection sort

۷.۶ Merge Sort

مرتب سازی ادغامی یکی از روش های مرتب سازی و نمونه ای از الگوریتم های تقسیم و حل است.پیچیدگی زمانی این الگوریتم$O(nlogn)$است. مرتب سازی ادغامی شامل مراحل زیر است:

- تقسیم کردن آرایه به دو نیمه
- مرتب کردن نیمه ها به طور بازگشتی
- ادغام کردن نیمه های مرتب شده در یک آرایه



شکل ۴.۶: merge sort

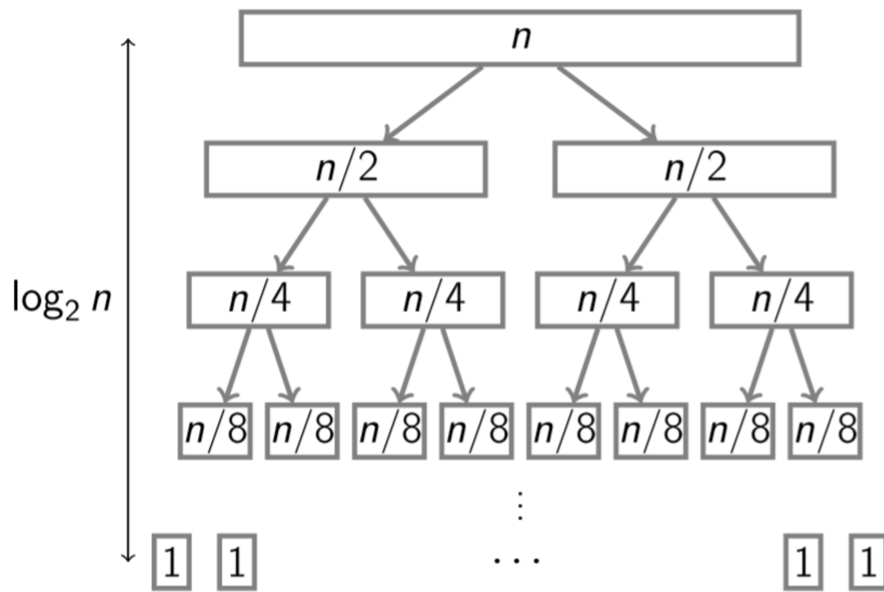
پیچیدگی زمانی مرحله ادغام کردن $O(n)$است.پس پیچیدگی زمانی کلی الگوریتم از رابطه بازگشتی زیر به دست می آید:

$$T(n) = 2T([\frac{n}{2}]) + O(n)$$

چون پیچیدگی زمانی الگوریتم ما از رابطه بازگشتی به دست می آید پس می توانیم با استفاده از قضیه اصلی پیچیدگی زمانی آن را بدست آوریم. شبه کدهای دو مرحله ادغام کردن و مرتب سازی یه صورت زیر است:

```
private static long[] MergeSort(long[] A, long n)
{
    if(n = 1)
        return A;
    var m = (long)(n/2);
    var b = MergeSort(A[1...m], m);
    var c = MergeSort(A[m+1...n], n-m);
    var a = Merge(B,c);
    return a;
}
```

```
private static long[] Merge(long[] a, long[]b)
{
    q = B.Length p; = A.Length Sorted; are b & a //
    long[] d = new long[p+q];
    int count = 0;
    while(a.Length !=0 && b.Length !=0)
    {
        var firstA = a[0];
        var firstB = b[0];
        if(firstA <= firstB)
            d[i] = firstA;
        else
            d[i] = firstB;
    }
    for(int i=count;i<d.Length;i++)
    {
        d[i] = a[i]; b[i] = d[i] or //
    }
    return d;
}
```



شکل ۵.۶: merge sort tree

جلسه ۷

Divide and Conquer

پریسا علائی - ۱۳۹۸/۷/۲۰

جزوه جلسه ۷ام مورخ ۱۳۹۸/۷/۲۰درس ساختمان‌های داده تهیه شده توسط پریسا علائی. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

۱.۷ مرتب سازی

برای مرتب سازی مقایسه ای ، حداقل زمان لازم n log n است. در واقع برای این نوع مرتب سازی ، یک درخت تصمیم گیری داریم .

`(Example` اگر

$$a_1$$

،

$$a_2$$

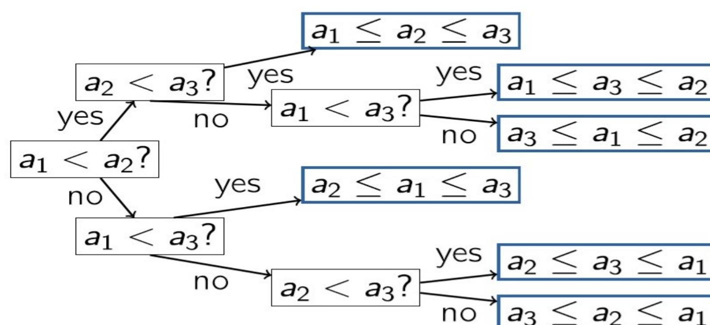
،

$$a_3$$

، ... ،

$$a_n$$

را داشته باشیم :



شکل ۱.۷: Decision Tree

۲.۷ نکات مثال

- در آخر این درخت به یک برگ میرسیم که مرتب شده ی

  a_1

  تا

  a_n

  وجود دارد.

- در هر نود این درخت یا شاخه های آن حالت هایی است که مثلا

  a_n

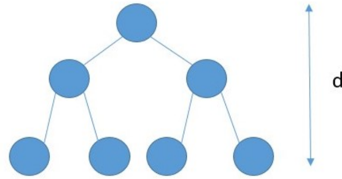
  ها می توانند نسبت به هم داشته باشند !

- در واقع حالت های متفاوت چیدن مثل جایگشت بدون تکرار است که کل حالت ها $n!$ است.
- تعداد مقایسه ها برابر است با عمق درخت. پس بیشترین عمق برابر است با حالت max در درخت.
- در حالت max تعداد برگ ها برابر است با 2^d البته درخت باید متوازن باشد .

۳.۷ اثبات اینکه n log n بهترین حالت است

۴.۷ Example

یک درخت به عمق ۲ چند حالت برگ دارد؟
پاسخ : میتواند حالت max باشد یا میتواند برگ هایش کمتر باشد.
عکس وسط و آخر یک حالت به حساب می آیند و حالت تکراری اند چون فقط تعداد برگ ها اینجا مهم است.
چند حالت دیگر هم برای این مثال هست و محدود به این مواردی که در بالا نشان داده شد نمی شود.



شکل ۲.۷: Tree

- 2^d>=n! →d>= Log n!

با تخمین این میتوانیم اوردر را بدست آوریم

- Log n! = log n +log n-1 + log n-2 + … + log 1

نصف این را نگه میداریم و بقیه را میریزیم دور و بعد یه تساوی برای I میزنیم

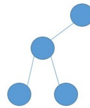
شکل ۳.۷: proof ۱

- Log n + log n-1 + … + log 1>= log n +…+ log n/2 >= log n/2 +log n/2 +…

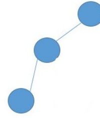
n/2 log n/2

از روی این اگر امگا بگیریم داریم
Ω(n log n)
حداقل مقایسه

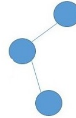
شکل ۴.۷: proof ۲



شکل ۵.۷: Tree ۱



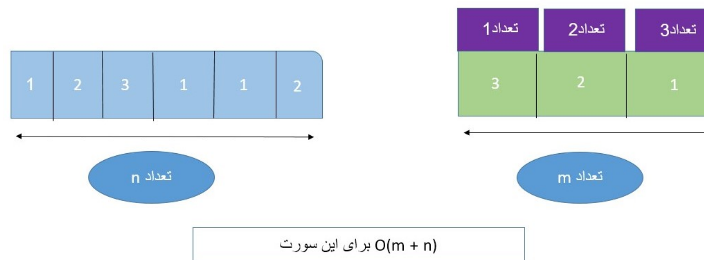
شکل ۶.۷: Tree ۲



شکل ۷.۷: Tree ۳

۵.۷ Sorting Small Integers

برای استفاده از این روش باید تعداد متغییر ها محدود باشد و باید متغییر ها را بشناسیم.
تعداد را با یک دور چک کردن می توان فهمید ، بعد به تعداد متغییر هایی که داریم از کوچک به بزرگ آن ها را مینویسیم مثلا : از کوچک ترین متغییر ۲ تا داریم اول دو تا از آن متغییر مینویسیم و بعد به ترتیب بقیه ی متغییر ها را مینویسیم.
برای استفاده از این روش حتما اعداد گسسته اند مانند : اعداد صحیح!! اگر پیوسته باشند جواب نمیدهد.



شکل ۸.۷: مثال

```
۱    private static void CoutSort(long[] a, long n)
۲    {
۳        long[] count = new long[m];
۴        for(int i=0; i < n ; i++)
۵        {
۶            count[a[i]] = count[a[i]] + 1;
۷        }
۸        long[] pos = new long[m];
```

```
        pos[0] = 1;
        for(int i=1; i< m; i++)
        {
            pos[i] = pos[i-1] + count[i-1];
        }
        for(int i=0; i< n; i++)
        {
            a[pos[a[i]]] = a[i];
            pos[a[i]] = pos[a[i]] + 1;
        }
    }
```

۶.۷ Stable Sort

سورتی است که اگر دو تا عنصر مقدارشان باهم برابر باشد اگر یکی قبل دیگری باشد ، ترتیبشان در بعد از سورت نیز رعایت میشود.

- 1 2 1 1 3 1 2
- 1 1 1 1 2 2 3

ترتیب 1 ها و 2 ها و 3 ها
به همان ترتیب اولیه است و
بعد از سورت تغییری نکرده
است

شکل ۹.۷: ۱ مثال

- 12
- 22
- 13

12
22
13

12
13
22

اول بر اساس یکان ها سورت میکنیم برای رقم دهگان باید دقت کنیم که کدام یک برای مثلا عدد 12 است و کدام برای عدد 13 اگر Stable
نباشد به صورت زیر مرتب میشود :
13
12
22

شکل ۱۰.۷: مثال ۲

۷.۷ Quick Sort

برای آرایه بهترین سورت است و حتی بهتر از Merge Sort ! در واقع روش عملکرد آن این گونه است که : یک خانه را آرایه انتخاب میکنیم و بعد جوری میچینیم که خانه هایی که دارای مقدار کوچک تر هستند سمت چپ و خانه هایی که دارای مقدار بزرگ تر هستند سمت راست قرار گیرند.
این روش به صورت بازگشتی است ، یعنی برای آن دو قسمت جدید باز باید این کار انجام شود تا تمام خانه ها مرتب شوند .

```
۱    private static void QuickSort(long[] a, long l, long r)
۲    {
۳        if(l >= r)
۴        {
۵            return;
۶        }
۷        long m = Partition(A,l,r);
۸        QuickSort(A,l,m-1);
۹        QuickSort(A,m+1,r);
۱۰   }
```

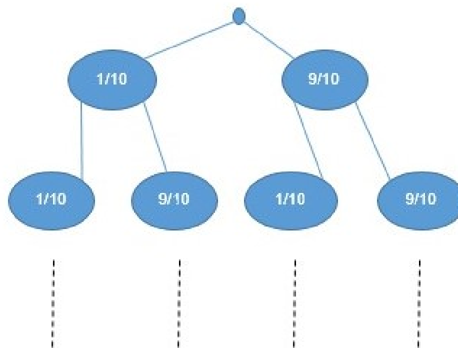
```
۱    private static long Partition(long[] a, long l, long r)
۲    {
۳        var pivot = a[l];
۴        var j =l;
۵        for(int i=l+1; i< r;i++)
۶        {
۷            if(a[i] <= pivot)
۸            {
۹                j = j+1;
۱۰               swap a[j] and a[i];
۱۱           }
۱۲       }
۱۳       swap a[l] and a[j];
۱۴       return j;
۱۵   }
```

۸.۷ چند نکته :

- در quick sort خود m را بررسی نمیکنیم ، چون در جایگاه نهاییش قرار گرفته است.
- pivot را خانه ای را در نظر میگیریم که جایگاهش تغییر نکند پس نمیشود در حالیکه داریم با اولین pivot مقایسه میکنیم برای دو تا بازه ی بعدی که هنوز همه ی خانه هایشان مشخص نشده است pivot در نظر بگیریم و آن ها را هم سورت کنیم .
- در بهترین حالت O(n log n) و در بدترین حالت $O(n^2)$ است .

$$T(n) = T(n/10) + T(9n/10) + O(n)$$

شکل ۱۱.۷: [؟] Balanced Partitions



شکل ۱۲.۷: Balanced Partitions Tree

منابع دیگر : [؟] [؟]

۹.۷ Random Pivot

در این حالت با Quick Sort عادی یک تفاوت وجود دارد آن هم این است که در این حالت خانه ی اول را انتخاب نمیکنیم و یکی از خانه های آرایه را به صورت رندوم انتخاب و بعد نسبت به آن جابه جا میکنیم .
با این کار احتمال $O(n^2)$ را کم میکنیم چون داده های متفاوتی را هر دفعه به آن میدهیم.
به طور متوسط داریم : O(n log n)

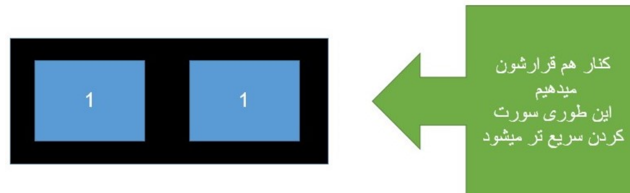
```
۱    private static void RandomizedQuickSort(long[] a, long l, long r)
۲    {
```

```
۳        if(l >= r)
۴        {
۵            return;
۶        }
۷        var k = random(l,r);
۸        swap a[l] and a[k];
۹        long m = Partition(A,l,r);
۱۰       QuickSort(A,l,m-1);
۱۱       QuickSort(A,m+1,r);
۱۲   }
```

منبع دیگر : [؟]

۱۰.۷ Equal Elements

اگر آرایه دارای خانه هایی با متغییر های یکسان باشد ، آن خانه ها را کنار هم قرار میدهیم و اگر لازم شد باهم جابه جایشان میکنیم .
با این کار سورت کردن سریع تر صورت میگیرد.
تئوری ساده و خودمونی : اگر یک بازه باشد که دارای عناصر برابر باشد مثل $p, p-1, n, n, n, k, k+1$ درPartition که در قسمت quick sort گفتیم به جای اینکه عدد وسط را بازگرداند ، باید یک بازه از جایگاه هایی که دارای عدد مساوی است را به ما بدهد ، و بعد فورها را برای دوتا بازه ی سمت راست و چپ باید جوری بنویسیم که بازه ی مساوی را دست نزند و از بعد آخرین خانه و از قبل اولین خانه ی بازه ی برگردانده شده شروع کند .



شکل ۱۳.۷: نمونه

```
private static void RandomizedQuickSort(long[] a, long l, long r)
{
    if(l >= r)
    {
        return;
    }
    var k = random(l,r);
    swap a[l] and a[k];
    var (m1,m2) = Partition3(a,l,r);
    QuickSort(A,l,m1-1);
    QuickSort(A,m2+1,r);
}
```

۱۱.۷ Sort Visualization

- این لینک انیمیشن سورت ها با داده هایی که از پیش تعیین شده است را نشان میدهد. لینک اول
- در این لینک میتوانید یکی از انواع سورت ها را انتخاب کنید و نحوه ی عملکرد آن ها را به صورت درخت ببینید.(داده ها را به صورت دستی وارد میکنید.) لینک دوم
- این لینک دارای کد سورت ها به زبان جاوا است و میتوانید به صورت انیمیشن نحوه ی عملکرد سورت ها را ببینید.(هم به صورت دستی داده وارد میکنید و هم دارای داده های از پیش تعیین شده است.) لینک سوم

جلسه ۸

Tail Recursion، برنامه نویسی پویا

سهراب نمازی نیا - ۱۳۹۸/۷/۲۲

جزوه جلسه ۸ام مورخ ۱۳۹۸/۷/۲۲ درس ساختمان‌های داده تهیه شده توسط سهراب نمازی نیا. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

۱.۸ Tail Recursion

طبق آنچه که جلسه گذشته در مورد الگوریتم Quick Sort گفته شد، در هر مرحله از این الگوریتم، با انتخاب Pivot مناسب، عناصر کوچکتر از Pivot را در سمت چپ آن و عناصر بزرگتر را نیز در سمت راست آن نگهداری میکردیم [؟]. اکنون میخواهیم مدل بهینه تری از Quick Sort را نسبت به جلسه گذشته بررسی کنیم. می خواهیم الگوریتم Quick Sort را به گونه ای پیاده سازی کنیم که تضمین شود در بدترین حالت ممکن Space Complexity آن برابر با log n است. در این روش ابتدا سمت چپ عنصر Pivot را به همان روش بازگشتی مرتب میکنیم. اما در سمت راست، به جای مجددا فراخوانی کردن تابع Quick Sort، اندیس شروع لیست را به اندیس میانی منتقل میکنیم و این عمل تا زمانی که اندیس اشاره گر به اول لیست از اندیس اشاره گر به آخر لیست کوچکتر است، ادامه می یابد. پس در واقع تا زمانی که شرط مورد نظر برقرار است، قسمت سمت چپ Pivot همانند حالت عادی الگوریتم Quick Sort به صورت بازگشتی انجام میشود. اما به جای فراخوانی مجدد تابع به صورت بازگشتی برای سمت راست، اندیس اشاره گر به ابتدای لیست را تغییر میدهیم. پس در واقع تا اینجا ما توانستیم یکی از روابط بازگشتی را از الگوریتم خود حذف کنیم. در زیر شبه کد مربوط به این الگوریتم را مشاهده میکنید:

```
long[] A;
while l < r do
    m ← partition(A, l, r);
    QuickSort(A, l, m - 1);
    l ← m + 1;
end
```
Algorithm 10: Tail Recursion

حال میخواهیم عمل بازگشتی را آگاهانه تر انجام دهیم. به این معنی که لزوما سمت چپ Pivot برای انجام رابطه بازگشتی انتخاب نشود. این بار باید از بین سمت راست و سمت چپ، بازه ای را که طول کوتاه تری دارد برای این کار انتخاب کنیم. برای درک بهتر به شبه کد زیر توجه فرمایید :

```
long[] A;
while l < r do
    m = partition(A, l, r);
    if (m - l) < (r - m) then
        QuickSort(A, l, m - 1);
        l ← m + 1;
    else
        QuickSort(A, m + 1, r);
        l ← m - 1;
    end
end
```
Algorithm 11: Tail Recursion

۱.۸

۲.۸ Intro Sort

در الگوریتم Quick Sort انتخاب Pivot مناسب نیز مهم است. بهترین Pivot عنصری است که بعد از Partition بندی، آرایه را به دو زیر مجموعه متعادل تبدیل کند. متعادل به این معنا است که هر دو زیرمجموعه تعداد اعضای نزدیک به هم داشته باشند. به همین دلیل برای اینکه Pivot از کوچکترین عنصر بودن و یا بزرگترین عنصر بودن فاصله بگیرد، آن را به این روش انتخاب میکنیم: سه عضو دلخواه آرایه را انتخاب میکنیم. عموما این سه عضو را عضو ابتدایی، میانی و انتهایی آرایه در نظر میگیرند. سپس عنصر میانی از لحاظ مقدار را از بین این سه عنصر به عنوان Pivot انتخاب میکنیم.

این الگوریتم که نمونه بهینه سازی شده از الگوریتم Quick Sort است را الگوریتم Intro Sort میگویند. یک بهینه سازی ممکن دیگر در این الگوریتم این است که اگر عمق درخت بازگشتی از مقدار خاصی بزرگتر شد، الگوریتم را تغییر داده و برای مرتب کردن داده ها از الگوریتم Heap Sort استفاده شود. [؟]

۳.۸ برنامه نویسی پویا

در روش تقسیم و حل دیدیم که میتوان یک مساله را به دو یا چند مسئله کوچکتر تقسیم کرد و با حل زیر مسئله ها، جواب مسئله نهایی را بدست آورد. اما مشکلی که وجود داشت این بود که گاهی مجبور بودیم که یک زیر

مسئله را که ممکن بود زیر مسئله تعداد زیادی از زیر مسئله های دیگر نیز باشد، چندین بار محاسبه کنیم که این موضوع از لحاظ Time Complexity به عنوان یک مشکل اساسی محسوب میشود. اما اگر ما مسئله ها را به ترتیبی حل کنیم که در هر مسئله تمام زیر مسئله های لازم برای آن از قبل حل و ذخیره شده باشند، دیگر به مشکل اشاره شده برنخواهیم خورد [؟]. این روش حل مسائل را برنامه نویسی پویا میگویند. به دو مثال زیر در مورد برنامه نویسی پویا توجه فرمایید :

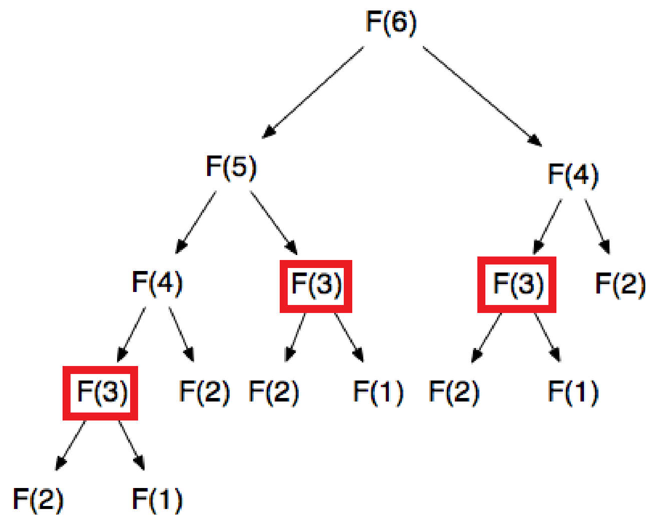
۱.۳.۸ دنباله فیبوناچی

دنباله فیبوناچی را در نظر بگیرید:

```
Fib(0) = 0, Fib(1) = 1
```

```
Fib[n] = Fib[n - 1] + Fib[n - 2]
```

اگر بخواهیم برای محاسبه جمله ای دلخواه از دنباله فیبوناچی به روش بازگشتی عمل کنیم، برای مقادیر بزرگ زمان بسیار زیادی برای محاسبه حاصل طول خواهد کشید. دلیل این موضوع این است که در درخت بازگشتی مربوطه، خیلی از مقادیر فیبوناچی بار ها محاسبه میشوند. به شکل زیر دقت کنید:
۱.۸



شکل ۱.۸: درخت بازگشتی فیبوناچی عدد ۶

برای رفع این مشکل باید جملات این دنباله را به ترتیبی بدست بیاوریم که در هر مرحله، جملاتی که برای بدست آوردن عدد فیبوناچی در آن مرحله نیاز داریم از قبل محاسبه و ذخیره شده باشند. بدست آوردن این ترتیب در برخی مسائل کمی دشوار می باشد. اما در مورد دنباله فیبوناچی به راحتی میتوان به کمک برنامه نویسی پویا این مسئله را حل کرد. یک آرایه را در نظر بگیرید. اگر جملات دنباله فیبوناچی را به ترتیب در این آرایه ذخیره کنیم، همواره مسئله های پیش نیاز برای ما از پیش حل شده خواهند بود. به شبه کد زیر دقت کنید:

```
long[] Fibs;
Fibs[0] = 0;
Fibs[1] = 1;
long i = 2;
while i <= n do
    Fibs[i] = Fibs[i - 1] + Fibs[i - 2];
    i++;
end
```
Algorithm 12: Fibonacci, dynamic programming

۲.۳.۸ مسئله خرد کردن پول

حال میخواهیم به بررسی مثالی دیگر از کاربرد برنامه نویسی پویا بپردازیم. فرض کنید انواع معینی سکه داریم و قرار است پول دلخواهی را با کمترین تعداد سکه های لازم خرد کنیم. برای حل این مسئله الگوریتم حریصانه لزوما به جواب درست منتهی نمیشود. به عنوان مثال فرض کنید سکه های ۵، ۱۰، ۲۰ و ۲۵ تومانی موجود است و قرار است که اسکناسی چهل تومانی را به کمک کمترین تعداد آن ها خرد کنیم. به شبه کد الگوریتم حریصانه برای حل این مسئله دقت کنید:

```
Change ← empty collection of coins
while money > 0 do
    coin ← largest denomination that does not exceed money
     add coin to Change
     money ← money − coin
end
return change;
```
Algorithm 13: greedy way to solve "Change Problem"

اگر بخواهیم به روش حریصانه این مسئله را حل کنیم، نهایتا به این جواب خواهیم رسید که به یک سکه ۲۵ تومنی، یک سکه ۱۰ تومانی و یک سکه ۵ تومانی برای خرد کردن این اسکناس لازم است. در حالی که همین مسئله را فقط با ۲ سکه ۲۰ تومانی میتوان حل کرد. پس الگوریتم حریصانه نمیتواند پاسخگوی این مسئله باشد.

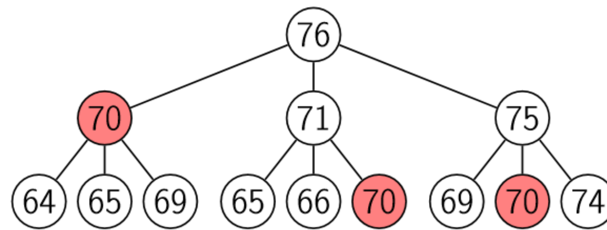
حال میخواهیم این مسئله را به روش تقسیم و حل بررسی کنیم. به کمک این روش میتوانیم بگوییم که در هر مرحله همه حالت های ممکن را محاسبه کرده و آن که به کمترین تعداد سکه لازم منجر میشود، همان جواب مسئله باشد. به عنوان مثال برای همان مسئله مطرح شده، به جای بدست آوردن جواب برای اسکناس ۴۰ تومانی، ابتدا فرض میکنیم یک بار از اسکناس ۲۵ تومانی استفاده کرده ایم و مسئله را برای پول باقیمانده حل میکنیم. این به این معنی است که جواب برای اسکناس چهل تومانی یکی بیشتر از جواب برای آن پول باقی مانده خواهد شد. اما این تنها یک حالت ممکن است. باید همین کار را برای سکه های دیگر هم به جای سکه ۲۵ تومانی انجام دهیم و نهایتا کمترین حاصل از میان جواب های بازگشتی بدست آمده، جواب نهایی مسئله برای خرد کردن اسکناس ۴۰ تومانی خواهد بود.

به شبه کد این مسئله به روش تقسیم و حل دقت فرمایید:

```
MinNumCoins ← ∞;
if money = 0 then
 | return 0
end
while i <= number of Coins do
 | if money >= Coins[i] then
 | | NumCoins ← RecursiveChange(money − Coins[i], coins)
 | | if NumCoins +1 < MinNumCoins then
 | | | MinNumCoins ← NumCoins +1;
 | | end
 | end
 | i ← i + 1;
end
return MinNumCoins;
```

Algorithm 14: Change problem, divide and conquer

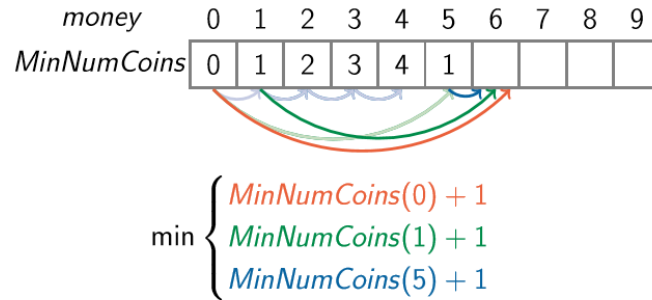
اگر اسکناس ورودی مسئله مقدار خیلی بزرگی داشته باشد، الگوریتم تقسیم و حل مشابه آنچه در مسئله فیبوناچی اتفاق افتاد به دلیل حل کردن مجدد مسئله های تکراری قادر نخواهد بود تا در زمان کوتاه به جواب نهایی برسد. به عنوان مثال فرض کنید سکه های ۱، ۵ و ۶ تومانی در اختیار داریم و میخواهیم یک اسکناس ۷۶ تومانی را با کمترین تعداد سکه ممکن به روش تقسیم و حل خرد کنیم. همانطور که در شکل میبینید، ما چندین بار این مسئله را برای یک اسکناس ۷۰ تومانی حل میکنیم که این کار از لحاظ زمانی برای مقادیر بزرگ، الگوریتم ما را با مشکل مواجه خواهد کرد.
۲.۸



شکل ۲.۸: درخت بازگشتی برای مسئله خرد کردن پول (اسکناس ۷۶ تومانی)

به همین دلیل به سراغ روش برنامه نویسی پویا برای حل این مسئله خواهیم رفت. در این روش، ترتیب پر کردن خانه های آرایه همانند مسئله فیبوناچی است. مطابق شکل زیر، میخواهیم جواب مسئله را برای یک اسکناس ۷ تومانی بدست آوریم و سکه های موجود ۱، ۵ و ۶ تومانی میباشند. فرض کنید ما تمام زیر مسئله های لازم برای پاسخ دادن به این مسئله را حل نموده ایم و جدول تا خانه شماره ۶ کامل شده است. حال بسته به این که کدام یک از سکه ها به عنوان اولین سکه انتخاب شود، مسئله به سه زیر مسئله ی از قبل حل شده تقسیم میشود. کمترین جواب از میان این سه مسئله را انتخاب کرده و با یک جمع میکنیم. حاصل جواب خانه شماره ۷ خواهد بود. این عمل را تا آخرین خانه ادامه میدهیم تا جواب نهایی مسئله بدست آید.

۳.۸



شکل ۳.۸: حل مسئله خرد کردن پول به روش برنامه نویسی پویا

به شبه کد مربوط به حل پویای این مسئه دقت فرمایید:

```
MinNumCoins(0) ← 0;
for m from 1 to money do
    MinNumCoins(m) ← ∞;
    for i from 1 to NumberOfCoins do
        if m > Coins[i] then
            NumCoins ← MinNumCoins(m − Coins[i]) + 1;
            if NumCoins < MinNumCoins[m] then
                MinNumCoins[m] ← NumCoins
            end
        end
    end
end
return MinNumCoins[money];
```

Algorithm 15: Change problem, dynamic programming

۴.۸ خلاصه

به طور خلاصه برای جمع بندی مطالب جلسه هشتم میتوان به موارد زیر اشاره کرد:

- یکی از روش های بهینه سازی الگوریتم Quick Sort، Tail Recursion نام دارد که در آن با کاهش یکی از دو رابطه بازگشتی، تضمین میشود که عمق درخت بازگشتی مربوطه از log n بیشتر نخواهد شد. در این روش به جای یکی از روابط بازگشتی، اندیس اشاره گر به ابتدای آرایه را در حلقه برنامه آپدیت میکنیم.
- یکی دیگر از روش های بهینه سازی الگوریتم Quick Sort، Intro Sort نام دارد که در آن با انتخاب تقریبی میانه به عنوان Pivot از غیر متعادل شدن تفاوت تعداد اعضای سمت چپ و راست

آن جلوگیری میکند.

- برنامه نویسی پویا در واقع به روش تقسیم و حل کمک میکند که یک زیر مسئله را برای بار های متوالی حل نکند. در این روش باید به ترتیبی مسائل را حل کنیم که زیر مسئله های ما همیشه از پیش حل شده باشند. مسئله فیبوناچی و خرد کردن اسکناس دو نمونه از کاربرد های برنامه نویسی پویا محسوب میشوند.

در جلسه آینده مسائل بیشتر و پیچیده تری از برنامه نویسی پویا بررسی خواهند شد.

جلسه ۱۱

برنامه ریزی پویا

سهند نظرزاده – ۱۳۹۸/۸/۵

۱.۱۱ مقدمه

برنامه ریزی پویا [۱] (این روش با نام برنامه‌نویسی پویا نیز شناخته می‌شود . منظور از برنامه‌نویسی، روشی خطی برای حل مسئله است، نه نوشتن کد در رایانه) همانند الگوریتم تقسیم و حل [۲] ، مسئله با ترکیب زیرمسئله ها حل می شود با این تفاوت که در الگوریتم تقسیم و حل زیر مسئله ها مستقل از هم بودند اما در برنامه ریزی پویا زیر مسئله ها کاملا مستقل نیستند و زیر مسئله ها ، زیر مسئله های مشترکی (زیر مسئله های هم پوشان) دارند یا زیرمسئله های یکسانی از مسیر های مختلفی حاصل می شوند که با این حال جوابشان یکسان است و در هر مرحله نیازی به محاسبه ی مجدد یک مسئله ی تکراری نیست و فقط یک بار آن را حل کرده و ذخیره می‌کنیم . معمولا برای حل مسائل بهینه سازی و شمارشی از برنامه ریزی پویا استفاده می شود که مسئله جواب یکتا ندارند اما فقط یک جواب یا اندازه ی بیشینه بودن یا کمینه بودن اهمیت دارد . قسمت مهم حل مسائل با این الگوریتم ترتیب حل زیر مسئله ها است که باید به شکلی صورت گیرد که همه زیر مسئله های یک مسئله قبل از آن حل شده باشند . چارچوب استانداردی برای حل عمومی مسائل برنامه ریزی پویا وجود ندارد . بلکه برنامه ریزی پویا فقط یک روش برخورد کلی (یا یک دید کلی) جهت حل دسته‌ای از مسائل ارائه می‌دهد.

۲.۱۱ مسائل بررسی شده

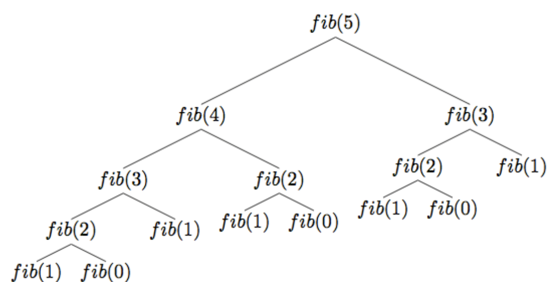
۱.۲.۱۱ Fibonacci

مسئله. الگوریتمی طراحی کنید که با الگوریتم برنامه ریزی پویا n امین عدد دنباله ی فیبوناچی را محاسبه کند.
حل. با توجه به تعریف دنباله فیبوناچی داریم

$\text{Fib(i)} = \text{Fib(i - 1)} + \text{Fib(i - 2)}\ ,\ \text{Fib}(0) = 0\ ,\ \text{Fib}(1) = 1$

برای $n = 5$ زیر مسئله ها به ترتیب شکل ۱.۱۱ صدا زده می شود

[۱] Programming Dynamic
[۲] Conquer and Divide



شکل ۱.۱۱: درخت زیر مسئله ها

با توجه به تکرار زیر مسئله ها (به عنوان مثال زیر مسئله Fib(2) سه بار تکرار شده است) الگوریتم برنامه ریزی پویا نسبت به الگوریتم تقسیم و حل بهینه تر است به همین دلیل هر زیر مسئله را یک بار حل و آن را ذخیره می کنیم. شبه کد ۱۶ آن به صورت زیر است

Data: n
Result: Fib_{n-1}
Declare Fib as an array
$Fib_0 = 0$
$Fib_1 = 1$
for $i \leftarrow 0$ *to n-1* **do**
| $Fib_i = Fib_{i-1} + Fib_{i-2}$
end

Algorithm 16: DP solution

۲.۲.۱۱ Change Money

مسئله. الگوریتمی طراحی کنید که با کمترین تعداد از سکه های $c_1, c_2, c_3, ..., c_m$ سنتی، money سنت را خرد کند.
حل.

الگوریتم حریصانه

با توجه به مثال (شکل ۲.۱۱) زیر جواب بدست آمده از الگوریتم حریصانه درست نیست زیرا برای خرد کردن ۴۰ سنت با سکه های ۵، ۱۰، ۲۰ و ۲۵ سنتی ، دو سکه ی ۲۰ سنتی بهینه تر از سکه های ۵ ، ۱۰ و ۲۵ سنتی است به همین دلیل الگوریتم حریصانه برای این مسئله کار آمد نیست.

40 cents = 25 + 10 + 5 = 20 + 20

Greedy is not Optimal

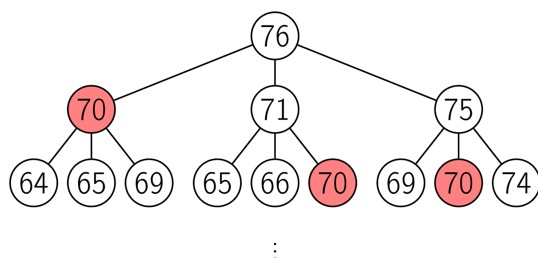
شکل ۲.۱۱: جواب حریصانه

الگوریتم بازگشتی

ابتدا باید رابطه ی بازگشتی مسئله را با کمک گرفتن از الگوریتم تقسیم و حل پیدا کنیم. برای این کار باید محیط مسئله اصلی را به نحوی کوچک تر کنیم تا به زیر مسئله هایی مشابه با مسئله ی اصلی برسیم. در این مسئله بر روی انتخاب شدن یا نشدن سکه i ام حالت بندی می کنیم.
در صورت انتخاب شدن سکه ی i ام که $i = 1, 2, 3, ... , m$ محیط مسئله از money به money - c_i تبدیل می شود و در صورت انتخاب نشدن سکه ی i ام در این مرحله به سراغ سکه های بعدی می رویم و بر روی آن ها حالت بندی می کنیم و در آخر زیر مسئله ای که با کمترین تعداد سکه حل شده است را به عنوان بهینه ترین زیر مسئله انتخاب می کنیم . این کار را تا زمانی که محیط مسئله به اندازه کافی ساده نشده ادامه می دهیم. رابطه بازگشتی به صورت زیر می شود.

$$MinNumCions(money) = Min \begin{cases} MinNumCions(money - c_1) + 1 \\ MinNumCions(money - c_2) + 1 \\ MinNumCions(money - c_3) + 1 \\ ... \\ MinNumCions(money - c_m) + 1 \end{cases} \tag{۱.۱۱}$$

طبق شکل ۳.۱۱ برای $money = 76$ و با سکه های $c_1 = 1, c_2 = 5, c_3 = 6$ سنتی، به دلیل محاسبات تکراری (در همین ابتدای کار زیر مسئله ی $money = 70$ سه بار تکرار شده است) الگوریتم بازگشتی نیز از لحاظ زمانی کار آمد نیست.



شکل ۳.۱۱: درخت زیر مسئله ها

شبه کد ۱۷ آن به صورت زیر است.

```
Data: money, coin
Result: MinNumCoins
if money = 0 then
 |  return 0;
end
MinNumCoins ← ∞
for i ← 0 to |cions| do
 |  if coin_i ≤ money then
 |   |  NumCoins ← MCRecursive(money − coin_i, coin)
 |   |  if NumCoins + 1 < MinNumCoins then
 |   |   |  MinNumCoins ← NumCoins + 1
 |   |  end
 |  end
end
return MinNumCoins;
```

Algorithm 17: MCRecursive

الگوریتم برنامه ریزی پویا

ابتدا باید رابطه بین مسئله و زیر مسئله ها را پیدا کنیم که همان رابطه ی بازگشتی ۱.۱۱ است.
در ادامه باید زیر مسئله ها را به ترتیبی حل و ذخیره کنیم که قبل از حل هر مسئله، زیر مسئله های آن حل و ذخیره شده باشد. طبق رابطه ی بازگشتی برای حل مسئله $\mathrm{money} = \mathrm{k}$ باید مسائل $\mathrm{money} = \mathrm{k} - c_i$ که $i = 1, 2, 3, \ldots, \mathrm{m}$ قبل از آن حل شده باشد. به همین دلیل از مسئله $money = 1$ شروع کرده و به ترتیب مسائل $money = 2$ و $money = 3$ الی $money = \mathrm{k}$ را حل و ذخیره می کنیم. برای ذخیره جواب زیر مسائل ، آرایه ی $MinNumCoins$ و به طول $\mathrm{k} + 1$ را تعریف می کنیم. برای راحتی کار جواب مسئله ی $money = 0$ را صفر فرض می کنیم $MinNumCoins_0 = 0$ (صفر سنت پول با صفر سکه خرد می شود).
برای درک بهتر برنامه ریزی پویا ، مسئله ی $money = 9$ و با سکه های $c_1 = 1, c_2 = 5, c_3 = 6$ سنتی را بررسی می کنیم. در ابتدا آرایه $MinNumCoins$ به شکل ۴.۱۱ زیر است.

money	0	1	2	3	4	5	6	7	8	9
MinNumCoins	0									

شکل ۴.۱۱: ابتدای کار

برای حل مسئله ی $money = 1$ باید جواب زیر مسئله های $money = 1 - 1 = 0$ ، $money = 1 - 5 = -4$ و $money = 1 - 6 = -5$ را بدانیم. دو زیر مسئله ی آخر به دلیل منفی بودن $(money - c_i < 0)$ ، قابل قبول نیستند. پس جواب این مسئله به صورت زیر می شود.

$$MinNumcoins_1 = MinNumCoins_0 + 1$$

و آرایه به شکل ۵.۱۱ زیر تغییر میکند.

money	0	1	2	3	4	5	6	7	8	9
MinNumCoins	0	1								

شکل ۵.۱۱: مرحله ی اول

مسئله های $money = 2$ ، $money = 3$ و $money = 4$ با روش مشابه ای حل می شوند. و آرایه به شکل ۶.۱۱ زیر تغییر می کند.

money	0	1	2	3	4	5	6	7	8	9
MinNumCoins	0	1	2	3	4					

شکل ۶.۱۱: مرحله ی چهارم

مسئله ی $money = 5$ با مسائل قبلی کمی متفاوت است. رابطه ی بازگشتی را برای آن می نویسیم تا

رابطه آن با زیر مسئله مشخص شود.

$$MinNumCions_5 = min \begin{cases} MinNumCions_{5-1} + 1 = MinNumCoins_4 + 1 \\ MinNumCions_{5-5} + 1 = MinNumCoins_0 + 1 \\ MinNumCions_{5-6} + 1 \end{cases}$$

(۲.۱۱)

زیر مسئله آخر به دلیل منفی بودن قابل قبول نیست ($money$ - $c_i < 0$).
حال باید با توجه به جواب زیر مسئله ها $MinNumCoins_0 = 0$ و $MinNumCoins_4 = 4$ از بین آن ها بهینه ترین جواب را انتخاب کرد.

$$MinNumCions_5 = min \begin{cases} MinNumCions_4 + 1 = 4 + 1 = 5 \\ MinNumCions_0 + 1 = 0 + 1 = 1 \end{cases} \quad (۳.۱۱)$$

پس جواب این مسئله به صورت زیر می شود.

$$MinNumCoins_5 = \min(5, 1) = 1$$

و آرایه به شکل ۷.۱۱ زیر تغییر می کند.

money	0	1	2	3	4	5	6	7	8	9
MinNumCoins	0	1	2	3	4	1				

شکل ۷.۱۱: مرحله ی پنجم

مسئله $money = 6$ را با روشی مشابه با مسئله ی $money = 5$ بررسی می کنیم. برای این کار به کمک رابطه ی بازگشتی ، رابطه ی میان مسئله و زیر مسائل را میابیم.

$$MinNumCions_6 = min \begin{cases} MinNumCions_{6-1} + 1 = MinNumCoins_5 + 1 \\ MinNumCions_{6-5} + 1 = MinNumCoins_1 + 1 \\ MinNumCions_{6-6} + 1 = MinNumCoins_0 + 1 \end{cases} \tag{۴.۱۱}$$

با توجه به جواب زیر مسائل داریم.

$$MinNumCions_6 = min \begin{cases} MinNumCions_5 + 1 = 1 + 1 = 2 \\ MinNumCions_1 + 1 = 1 + 1 = 2 \\ MinNumCions_0 + 1 = 0 + 1 = 1 \end{cases} \tag{۵.۱۱}$$

جواب این مسئله به صورت زیر بدست می آید.

$$MinNumCoins_6 = \min(2, 2, 1) = 1$$

آرایه به شکل ۸.۱۱ تغییر می کند.

money	0	1	2	3	4	5	6	7	8	9
MinNumCoins	0	1	2	3	4	1	1			

شکل ۸.۱۱: مرحله ی ششم

سایر مسایل با روشی مشابه ای حل می شوند. در آخر آرایه به شکل ۹.۱۱ زیر تبدیل می شود و پاسخ مسئله ی اصلی برابر با $MinNumCoins_{money} = MinNumCoins_9 = 4$ است.

money	0	1	2	3	4	5	6	7	8	9
MinNumCoins	0	1	2	3	4	1	1	2	3	4

شکل ۹.۱۱: انتهای کار

شبه کد ۱۸ به در زیر آمده است.

Data: $money$, $coin$
Result: $MinNumCoins[money]$
Declare $MinNumCoins$ as an array
$MinNumCoins_0 = 0$
for $i \leftarrow 1$ *to money* **do**
$MinNumCoins_i = \min(MinNumsCoins_{i-c_1}, MinNumCoins_{i-c_2}, MinNumCoins_{i-c_3}, ... , MinNumCoins_{i-c_m}) + 1$
end
return $MinNumCoins[money]$;

Algorithm 18: Dynamic solution

۳.۲.۱۱ Distance Edit

مسئله. الگوریتمی طراحی کنید که با کمترین تعداد از عملیات های اضافی کردن یک کاراکتر [۳] ، پاک کردن یک کاراکتر [۴] و تعویض یک کاراکتر با کاراکتری دیگر [۵] رشته ی A را به رشته ی B تبدیل کند.
حل. همانند مسئله ی قبل ابتدا رابطه میان مسئله و زیر مسائل را پیدا می کنیم. برای این کار رو آخرین کاراکتر هر دو رشته حالت بندی می کنیم.

$\mathrm{A} = a_1, a_2, a_3, \dots , a_n$
$\mathrm{B} = b_1, b_2, b_3, \dots , b_m$

بر روی a_n و b_m حالت بندی می کنیم:

حالت اول ($a_n = b_m$)
در این حالت به دلیل این که هر دو کاراکتر برابر هستند، برای تبدیل رشته ی اول به رشته ی دوم کافی است فرض کنیم که دو کاراکتر آخر وجود ندارند و سپس زیر مسئله ای که رشته های آن

$\mathrm{A} = a_1, a_2, a_3, \dots , a_{n-1}$
$\mathrm{B} = b_1, b_2, b_3, \dots , b_{m-1}$

هستند را حل کنیم و سپس کاراکتر ها را در انتها رشته ها قرار می دهیم (برای اضافی کردن کاراکتر ها به انتهای دو رشته نیازی به استفاده از عملگر ها نیست زیرا قبل از تبدیل مسئله به زیر مسئله در انتهای دو رشته وجود داشته اند) تا رشته ی اول به رشته دوم تبدیل شود. به دلیل این که در مسیر تبدیل رشته ی بدست آمده از حل زیر مسئله به رشته ی مسئله اصلی از عملگر ها استفاده نکردیم؛ جواب مسئله همان جواب زیر مسئله است.
به عنوان مثال برای مسئله ای که رشته ی اول آن $\mathrm{A} =$ "insert" و رشته ی دوم آن $\mathrm{B} =$ "edit" هستند؛ به دلیل برابر بودن کاراکتر آخر هر دو رشته ($a_6 = \mathrm{t}$ و $b_4 = \mathrm{t}$) کاراکتر ها را از انتهای دو رشته حذف می کنیم و زیر مسئله (کمترین تعداد از عملیات ها برای تبدیل رشته ی
$\mathrm{A} =$ "inser" به $\mathrm{B} =$ "edi") را حل می کنیم و سپس در کاراکتر ها را در انتهای دو رشته قرار می دهیم. به دلیل این که در مسیر تبدیل جواب زیر به جواب مسئله از عملگر ها استفاده نکردیم؛ جواب مسئله همان جواب زیر مسئله است.

---
[۳] Insertions
[۴] deletions
[۵] subtitutions

حالت دوم ($a_n \neq b_m$)

برای تبدیل کاراکتر a_n به کاراکتر b_n باید از عملگر ها استفاده کنیم. هر عملگر مسئله را به یک زیر مسئله مرتبط می کند.
بنابراین روی عملگر ها حالت بندی می کنیم:

عملگر تعویض کردن یک کاراکتر
برای تبدیل رشته ی اول به رشته ی دوم کافی است فرض کنیم که دو کاراکتر آخر وجود ندارند و سپس زیر مسئله ای که رشته های آن

$\mathrm{A} = a_1, a_2, a_3, ..., a_{n-1}$
$\mathrm{B} = b_1, b_2, b_3, ..., b_{m-1}$

هستند را حل کنیم و سپس کاراکتر آخر رشته ی اول a_n را با عملگر تعویض و با هزینه ی یک به کاراکتر آخر رشته ی دوم b_m تبدیل می کنیم و در انتهای رشته اول قرار می دهیم و کاراکتر آخر رشته ی دوم را انتهای همان رشته قرار می دهیم (برای اضافی کردن کاراکتر به انتهای رشته ی دوم نیازی به استفاده از عملگر ها نیست زیرا قبل از تبدیل مسئله به زیر مسئله در انتهای آن رشته وجود داشته است) تا رشته ی اول به رشته دوم تبدیل شود. به دلیل این که در مسیر تبدیل رشته ی بدست آمده از حل زیر مسئله به رشته ی مسئله اصلی، یک بار از عملگر تعویض استفاده شده است؛ جواب مسئله اصلی یک واحد بیشتر از زیر مسئله است.
به عنوان مثال برای مسئله ای که رشته ی اول آن "A = "edit و رشته ی دوم آن "B = "distance هستند؛ کاراکتر های آخر را از انتهای دو رشته حذف می کنیم و زیر مسئله (کمترین تعداد از عملیات ها برای تبدیل رشته ی "A = "edi به "B = "distanc) را حل می کنیم؛ سپس کاراکتر آخر رشته ی اول را با عملگر تعویض و با هزینه یک به کاراکتر آخر رشته ی دوم تبدیل می کنیم و در انتهای رشته ی اول قرار می دهیم و کاراکتر آخر رشته ی دوم را انتهای همان رشته قرار می دهیم (برای اضافی کردن کاراکتر به انتهای رشته ی دوم نیازی به استفاده از عملگر ها نیست زیرا قبل از تبدیل مسئله به زیر مسئله در انتهای آن رشته وجود داشته است) تا رشته ی اول به رشته دوم تبدیل شود.به دلیل این که در مسیر تبدیل رشته ی بدست آمده از حل زیر مسئله به رشته ی مسئله اصلی، یک بار از عملگر تعویض استفاده شده است؛ جواب مسئله اصلی یک واحد بیشتر از زیر مسئله است.

عملگر اضافه کردن یک کاراکتر
برای تبدیل رشته ی اول به رشته ی دوم کافی است فرض کنیم که کاراکتر آخر رشته دوم وجود ندارد و زیر مسئله ای که رشته های آن

$A = a_1, a_2, a_3, ..., a_n$
$B = b_1, b_2, b_3, ..., b_{m-1}$

هستند را حل کنیم؛ سپس کاراکتر آخر رشته ی دوم b_m را با عملگر اضافه کردن و با هزینه ی یک، به رشته ی بدست آمده از حل زیر مسئله اضافه می کنیم تا رشته ی اول به رشته دوم تبدیل شود. به دلیل این که در مسیر تبدیل رشته ی بدست آمده از حل زیر مسئله به رشته ی مسئله اصلی، یک بار از عملگر اضافه کردن استفاده شده است؛ جواب مسئله اصلی یک واحد بیشتر از زیر مسئله است.
به عنوان مثال برای مسئله ای که رشته ی اول آن $A =$ "edit" و رشته ی دوم آن $B =$ "distance" هستند؛ کاراکتر آخر رشته ی دوم $b_8 =$ e را از انتهای رشته دوم حذف می کنیم و زیر مسئله (کمترین تعداد از عملیات ها برای تبدیل رشته ی $A =$ "edit" به $B =$ "distanc") را حل می کنیم؛سپس کاراکتر آخر رشته ی دوم $b_8 =$ e را با عملگر اضافه کردن و با هزینه ی یک، به رشته ی بدست آمده از حل زیر مسئله اضافه می کنیم تا رشته ی اول به رشته دوم تبدیل شود. به دلیل این که در مسیر تبدیل رشته ی بدست آمده از حل زیر مسئله به رشته ی مسئله اصلی، یک بار از عملگر اضافه کردن استفاده شده است؛ جواب مسئله اصلی یک واحد بیشتر از زیر مسئله است.

عملگر پاک کردن یک کاراکتر
برای تبدیل رشته ی اول به رشته ی دوم کافی است کاراکتر آخر رشته اول a_n را با عملگر پاک کردن و با هزینه ی یک از آخر رشته اول پاک کنیم و زیر مسئله ای که رشته های آن

$A = a_1, a_2, a_3, ..., a_{n-1}$
$B = b_1, b_2, b_3, ..., b_m$

هستند را حل کنیم؛ به دلیل این که در مسیر تبدیل رشته ی مسئله اصلی به رشته ی بدست آمده از حل زیر مسئله، یک بار از عملگر پاک کردن استفاده شده است؛ جواب مسئله اصلی یک واحد بیشتر از زیر مسئله است.
به عنوان مثال برای مسئله ای که رشته ی اول آن $A =$ "edit" و رشته ی دوم آن $B =$ "distance" هستند؛ کاراکتر آخر رشته ی اول $a_4 =$ e را با عملگر پاک کردن و با هزینه ی یک از آخر رشته اول پاک کنیم و زیر مسئله (کمترین تعداد از عملیات ها برای تبدیل رشته ی $A =$ "edi" به $B =$ "distance") را حل می کنیم؛ به دلیل این که در مسیر تبدیل رشته ی مسئله اصلی به رشته ی بدست آمده از حل زیر مسئله، یک بار از عملگر پاک کردن استفاده شده است؛ جواب مسئله اصلی یک واحد بیشتر از زیر مسئله است.

برای باز نویسی رابطه ی میان مسئله و زیر مسائل آرایه دو بعدی D را به نحوی تعریف می کنیم که مقدار خانه ی $D_{i,j}$ برابر با کمترین تعداد عملیات هایی باشد که برای تبدیل رشته $A = a_1, a_2, a_3, ..., a_i$ به رشته ی $B = b_1, b_2, b_3, ..., b_j$ لازم است. رابطه ی میان مسئله و زیر مسائل به صورت زیر خلاصه می شود.

$$D_{i,j} = min \begin{cases} D_{i-1,j-1} & a_i = b_j \\ D_{i-1,j-1} + 1 & a_i \neq b_j \\ D_{i,j-1} + 1 & a_i \neq b_j \\ D_{i-1,j} + 1 & a_i \neq b_j \end{cases} \qquad (۶.۱۱)$$

حال با توجه به رابطه ی بدست آمده باید زیر مسائل را به ترتیبی حلو ذخیره کنیم که قبل از حل یک مسئله، زیر مسئله های آن مسئله حل و ذخیره شده باشد. طبق رابطه میان مسئله و زیر مسائل باید برای حل مسئله $D_{i,j}$ باید مسائل $D_{i-1,j-1}$ ، $D_{i,j-1}$ و $D_{i-1,j}$ قبل از آن حل و ذخیره شده باشند. بنابراین بر روی آرایه از بالا به پایین و از چپ به راست حرکت می کنیم.(در این مسئله حرکت در راستاهای افقی و عمودی نسبت به هم ارجعیت ندارند؛ مسائلی وجود دارند که تریتب راستای حرکت حلقه ی اول و راستای حلقه ی دوم مهم است.)
حال باید جواب زیر مسائلی را که به اندازه کافی کوچک هستند را در آرایه D ذخیره کنیم.(معمولا ردیف اول و ستون اول زیر مسائلی هستند که به اندازه کافی کوچک هستند.) اگر یکی از رشته ها هیچ کاراکتری نداشته باشد؛ زیر مسئله به اندازه ی کافی کوچک شده است که جواب آن را بدانیم.
حالت اول این است که رشته ی اول کاراکتری نداشته باشد؛ برای تبدیل رشته اول به رشته دوم با j کاراکتر باید j بار عملگر اضافه کردن یک حرف را روی رشته ی اول استفاده کنیم؛ تا رشته ی اول به رشته ی دوم تبدیل شود.
حالت دوم این است که رشته ی دوم کاراکتری نداشته باشد؛ برای تبدیل رشته اول با i کاراکتر به رشته دوم باید i بار عملگر پاک کردن یک حرف را روی رشته ی اول استفاده کنیم؛ تا رشته ی اول به رشته ی دوم تبدیل شود. روابط بالا را می توان به صورت زیر خلاصه کرد.

$$D_{0,j} = j \qquad j = 0, 1, 2, 3, ..., m \tag{۷.۱۱}$$

$$D_{i,0} = i \qquad i = 0, 1, 2, 3, ..., n \tag{۸.۱۱}$$

(۹.۱۱)

حال با توجه به رابطه ی به دست آمده و زیر مسائل حل شده به سراغ حل و ذخیره زیر مسائل پیچیده تر می رویم.
برای درک بهتر برنامه ریزی پویا ، مسئله ای با رشته های "A = "distance و "B = "editing را بررسی می کنیم. ابتدا زیر مسائلی که به اندازه کافی کوچک هستند را حل و در آرایه ذخیره می کنیم.
زمانی که هر دو رشته کاراکتری ندارند با صفر عملیات رشته ی اول به رشته دوم تبدیل می شود.
$D_{0,0} = 0$

زمانی که رشته ی اول کاراکتری ندارد و رشته ی دوم "B = "e است با اضافه کردن کاراکتر "e" به رشته ی اول، رشته ی اول به رشته دوم تبدیل می شود.
$D_{0,1} = 1$

زمانی که رشته ی اول کاراکتری ندارد و رشته ی دوم "B = "ed است با اضافه کردن کاراکتر "e" و "d" به رشته ی اول، رشته ی اول به رشته دوم تبدیل می شود.
$D_{0,2} = 2$

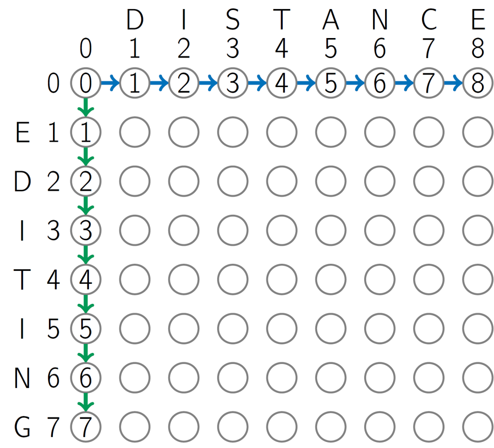
زیر مسائل $D_{0,j}$ که $j = 0, 1, 2, ..., m$ است؛ با منطقی مشابه حل می شوند.
$D_{0,j} = j$

زمانی که رشته اول "A = "d است و رشته دوم کاراکتری ندارد با پاک کردن کاراکتر "d" از رشته ی اول، رشته ی اول به رشته دوم تبدیل می شود.
$D_{1,0} = 1$

زمانی که رشته اول "A = "di است و رشته دوم کاراکتری ندارد با پاک کردن کاراکتر "d" و "i" از رشته ی اول، رشته ی اول به رشته دوم تبدیل می شود.
$D_{2,0} = 2$

زیر مسائل $D_{i,0}$ که $i = 0, 1, 2, ..., n$ است؛ با منطقی مشابه حل می شوند.
$D_{i,0} = i$

پس از حل و ذخیره مقادیر بالا آرایه به شکل ۱۰.۱۱ زیر در می آید.



شکل ۱۰.۱۱: ابتدای کار

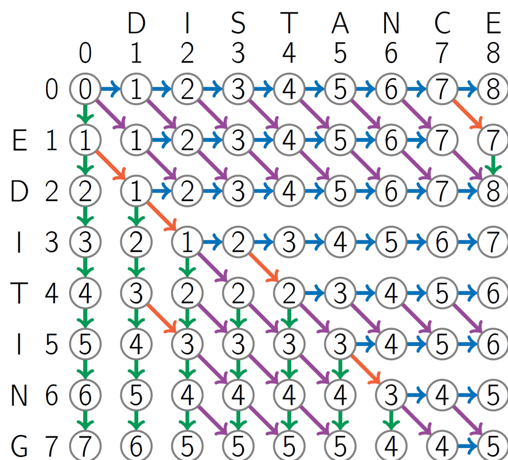
برای حل مسئله ی $D_{1,1}$ به دلیل اینکه کاراکتر های اول دو رشته برابر نیستند و طبق رابطه ی میان مسئله و زیر مسائل باید مقادیر زیر مسئله های $D_{0,0}$ ، $D_{1,0}$ و $D_{0,1}$ را بدانیم و از آن ها استفاده کنیم. جواب این مسئله به صورت زیر می شود.

$$D_{2,1} = min \begin{cases} D_{0,0} + 1 = 0 + 1 = 1 \\ D_{1,0} + 1 = 0 + 1 = 1 \\ D_{0,1} + 1 = 0 + 1 = 1 \end{cases}$$

برای حل مسئله ی $D_{2,1}$ به دلیل اینکه کاراکتر های اول دو رشته برابر هستند و طبق رابطه ی میان مسئله و زیر مسائل باید مقادیر زیر مسئله های $D_{1,0}$ ، $D_{1,1}$ و $D_{2,0}$ را بدانیم و از آن ها استفاده کنیم. جواب این مسئله به صورت زیر می شود.

$$D_{2,1} = min \begin{cases} D_{1,0} = 1D_{1,0} + 1 = 1 + 1 = 2 \\ D_{1,1} + 1 = 1 + 1 = 2 \\ D_{2,0} + 1 = 2 + 1 = 3 \end{cases}$$

زیر مسائل $D_{i,j}$ که $n ,... ,2 ,1 =$ و $j = 1, 2, ..., m$ است؛ با منطقی مشابه حل می شوند. پس از حل و ذخیره مقادیر بالا آرایه به شکل ۱۱.۱۱ زیر در می آید. جواب مسئله اصلی برابر است با مقدار $D_{n,m} = 5$ است.



شکل ۱۱.۱۱: انتهای کار

مراجع :

[؟]
[؟]
[؟]
[؟]

جلسه ۱۲

برنامه نویسی پویا-ادامه

یاسین عسکریان - ۱۳۹۸/۸/۶

۱.۱۲ کوله پشتی[۱]

مسئله کوله‌پشتی که با نام‌های Knapsack یا Rucksack مطرح می‌شود فرض کنید مجموعه‌ای از اشیا که هر کدام داری وزن و ارزش خاصی هستند در اختیار دارید. به هر شی تعدادی را تخصیص دهید به‌طوری‌که وزن اشیا انتخاب شده کوچکتر یا مساوی حدی از پیش تعیین شده، و ارزش آن‌ها بیشینه شود. علت نامگذاری این مسئله، جهانگردی است که کوله‌پشتی‌ای با اندازهٔ محدود دارد و باید آن را با مفیدترین صورت ممکن از اشیا پر کند. [؟]
از مثال های کاربردی دیگر آن می توان به تبلیغات تلویزیونی اشاره کرد فرض کنید شما مسئول تبلیغات یک شبکه ی تلویزیونی هستید،که قصد دارید تبلیغات را در حین پخش یک سریال پرطرفدار پخش کنید خب با توجه به زمانی که در اختیار دارید مثلا ۵ دقیقه باید از مجموعه ای از پیشنهادات تبلیغاتی گزینه هایی را انتخاب کنید تا بیشترین سود را برای شبکه داشته باشد
این نوع مسئله به دو حالت کلی تبدیل می شود که در ادامه به بررسی آن ها خواهیم پرداخت.

۱.۱.۱۲ کوله پشتی کسری[۲]

فرض کنید شما یک کوله پشتی دارید و می خواهید از یک خواربار فروشی اجناسی را انتخاب کنید که مجموع حجم شان از حجم کوله پشتی شما کمتر یا مساوی آن باشد و مجموع ارزش این اجناس در بیشترین حالت ممکن باشد خب مثلا اگر یک کیلو زعفران به ارزش صد هزار تومان نیم کیلو دارچین به ارزش هفتاد هزار تومان دارید و حجم کوله پشتی شما سیصد گرم باشد شما می توانید **کسری** از هرکدام از اجناس را انتخاب و قیمت آن ها را محاسبه کنید تا مجموع قیمت آن بیشینه شود
خب برای حل این مسئله از الگوریتم حریصانه [۳] استفاده می کنیم که در جلسه چهارم به بحث و بررسی آن پرداختیم.

---

[۱] Knapsack
[۲] Fractional Knapsack
[۳] Greedy Algorithms

۲.۱۲ کوله پشتی گسسته [۴]

فرض کنید شما یک کوله پشتی با حجم مشخص دارید و میخواهید آن رو با شمش طلا،نقره و برنز پر کنید تا مجموع ارزش آن بیشترین حالت ممکن باشد می دانیم که نمی توانیم کسری از یک شمش طلا را برداریم پس یا باید یک شمش طلا به کوله پشتی اضافه کنیم یا اصلا آن را اضافه نکنیم
این مسئله به دو حالت با تکرار و بدون تکرار تبدیل می شود در حالت با تکرار با تو جه به مثال بالا ما می توانیم چند شمش طلا برداریم ولی در حالت بدون تکرار حداکثر یک شمش طلا. لطفا به شکل زیر توجه کنید :

Example
$30 $14 $16 $9
6 3 4 2
$30 $16
w/o repeats 6 4 total: $46
$30 $9 $9
w repeats 6 2 2 total: $48
$30 $14 $4.5
fractional 6 3 1 total: $48.5

شکل ۱.۱۲: مقایسه حالت های مختلف مسئله کوله پشتی

۱.۲.۱۲ کوله پشتی با تکرار [۵]

برای حل این مسئله باید ابتدا بیشینه ی ارزش برای حجم ها کم تر از حجم کوله پشتی خودمان را بدست آوریم
به مثال زیر توجه کنید

---

[۴] Discrete Knapsack
[۵] Knapsack with Repetitions

Example: $W = 10$

$30	$14	$16	$9
6	3	4	2

0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0

شکل ۲.۱۲: کوله پشتی با تکرار(۱)

خب ما چهار جنس مختلف با حجم و ارزش های متفات داریم و حجم کوله پشتی ما نیز ده است خب ابتدا برای حجم صفر شروع به محاسبه بیشنه ارزش میکنیم و با توجه به اجناس در حجم صفر بیشینه نیز صفر است و همینطور برای حجم یک، خب برای حجم دو ما یک جنس به ارزش ۹ دلار و حجم دو داریم و جنس دیگری را نمی توانیم در داخل آن قرار دهیم پس بیشینه ی ارزش در حجم دو برابر با ۹ دلار می شود حال به سراغ حجم سه می رویم در این حجم ما دو جنس به حجم های دو و سه داریم خب اگر حجم دو را انتخاب کنیم ۹ دلار به ارزش آن اضافه شده و دو واحد از حجم آن کاسته می شود خب یک واحد حجم باقی می ماند برای پر کردن آن حجم به بیشینه ارزش در حجم یک رجوع کرده و مقدار آن که برابر با صفر هست را با ۹ دلار جمع می کنیم حال در حالت بعدی جنسی که حجم آن سه و ارزش آن ۱۴ دلار هست را انتخاب کرده که در اینصورت حجم باقی مانده برابر صفر می شود حال ارزش به دست آمده در این دو حالت را با هم مقایسه کرده و ارزش بیشتر را برای حجم سه انتخاب می کنیم که برابر است با ۱۴ دلار و به همین ترتیب ارزش ها را تا حجم ۱۰ حساب می کنیم

Example: $W = 10$

$30	$14	$16	$9
6	3	4	2

0	1	2	3	4	5	6	7	8	9	10
0	0	9	14	18	23	30	32	39	44	48

شکل ۳.۱۲: کوله پشتی با تکرار(۲)

در اینصورت الگوریتم حل مسئله کوله پشتی با توانایی تکرار اجناس به صورت زیر است

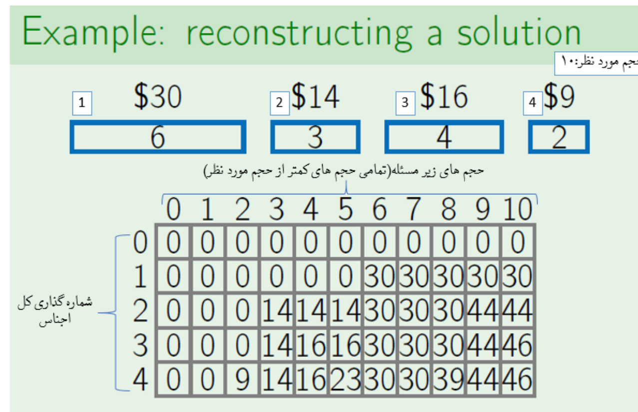
```
private static long Knapsack(long W, long v)
{
    long[] value = new long[W];
    value[0] = 0;
    for(value w =1; w < W; w++)
    {
        value[w] = 0;
        for(var i=1; i < n; i++)
        {
            if( wi <= w)
            {
                var val = vlue[w - wi] + vi;
                if(val > value(w)
                    value[w] = val;
            }
        }
    }
    return value[W];
}
```

حال اگر بخواهیم بدانیم که چه اجناسی را انتخاب کردیم تا به این بیشینه رسیدیم و ما آرایه زیرمسئله،حجم های کوچک تر از حجم مورد نظر،را داریم از اخرین خانه ی ارایه شروع میکنیم ابتدا حجم تمام اجناس را به ترتیب از حجم خانه ی آخر کم می کنیم خب تفاضل بدست آمده خود نیز یکی از زیر مسئله های ماست پس به سراغ خانه ای با شماره ی تفاضل بدست آمده میرویم اگر جنسی که ما انتخاب کردیم یکی از اجناسی باشد که در کوله پشتی قرار گرفته است باید جمع ارزش آن جنس با ارزش قرار گرفته در خانه ی بدست آمده برابر با ارزش نهایی شده باشد در غیراینصورت به سراغ جنس بعدی میرویم اگر جنس انتخابی درست باشد همین مراحل را برای آن خانه ی بدست آمده در آرایه تکرار میکنیم و آنقدر تکرار میکنیم تا به خانه ای با ارزش صفر برسیم

۲.۲.۱۲ کوله پشتی بدون تکرار [۶]

در این حالت زیرمسئله تنها حجم های کوچک تر از حجم مورد نظر نیست بلکه با این دید به مسئله نگاه می کنیم که پر کردن تمامی حجم ها ابتدا با جنس شماره یک سپس شماره یک و دو و همینطور الی آخر انجام شود. پس ما به یک آرایه دو بعدی نیاز داریم به شکل زیر توجه کنید

[۶] Knapsack without Repetitions



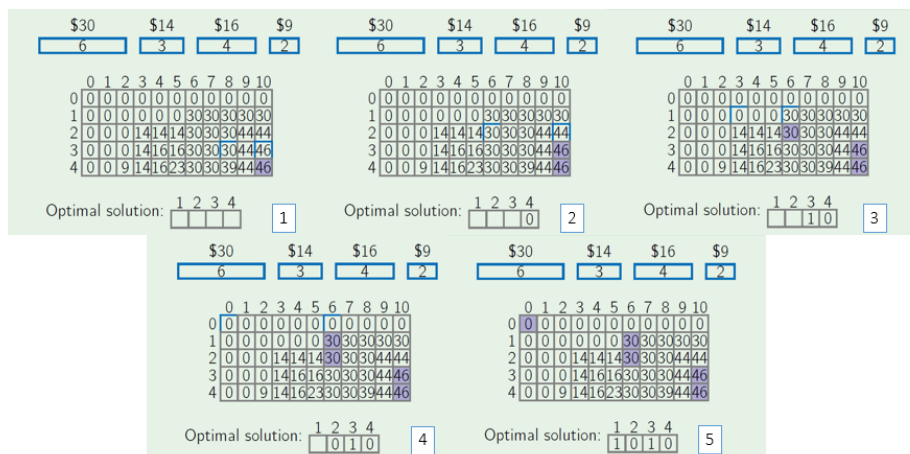
	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

شکل ۴.۱۲: بازسازی راه حل کوله پشتی بدون تکرار

با تو به شکل بالا اگر اسم آرایه دو بعدی M باشد خانه ی M(i,j) بیانگر آن است که کوله پشتی ای با حجم i و با استفاده از اجناس با شماره ی کوچک تر مساوی j یعنی [j,j-1,j-2,j-3,...,0] حداکثر M(i,j) ارزش در آن جای گرفته است
و الگوریتم آن را نیز می توانید در شکل زیر مشاهده کنید

```
۱   private static long Knapsack(long W, long v)
۲   {
۳       initialize all value(0,j) = 0;
۴       initialize all value(w,0) = 0;
۵       for(var i=1; i < n; i++)
۶       {
۷           for (var w=1; w < W ; w++)
۸           {
۹               value[w,i] = value[w, i-1];
۱۰              if (wi < w)
۱۱              {
۱۲                  var val = value[w - wi, i-1] + vi;
۱۳                  if(value[w,i] < val)
۱۴                      value[w,i] = val;
۱۵              }
۱۶          }
۱۷      }
۱۸      return value[W, n];
۱۹  }
```

حال اگر ما آن آرایه دو بعدی را داشته باشیم و بخواهیم بدانیم چه اجناسی را انتخاب کرده ایم ابتدا به آخرین خانه یعنی M(imax,jmax) مراجعه می کنیم این خانه بیانگر آن است که آیا ما از جنس jmax استفاده کرده ایم یا خیر اگر استفاده کرده باشیم باید جمع ارزش خانه ای با حجم (W-w(jmax)) با value(jmax) برابر با M(imax,jmax) شود در غیر اینصورت یعنی از جنس با شماره jmax استفاده نکرده ایم برای درک بیشتر به شکل زیر توجه کنید



شکل ۵.۱۲: کوله پشتی بدون تکرار

۳.۱۲ قرار دادن پرانتز [۷]

در این مسئله ورودی های ما متشکل از یک دنباله اعداد و یک دنباله از چهار عمل اصلی ریاضی است به این که بین هردو عدد حتما یک عملگر وجود دارد و ما وظیفه داریم تا با پرانتز گذاری مناسب بیشینه یا کمینه ی جواب نهایی این عبارت را بدست آوریم
فرض کنید عبارتی مثل $5 - 8 + 7 \times 4 - 8 + 9$ و می خواهیم بیشینه این عبارت را در حالت زیر بدست آوریم
$(5 - 8 + 7) \times (4 - 8 + 9)$

[۷] Parentheses Placing

Example: $(5-8+7)\times(4-8+9)$

$$\begin{aligned} \min(5-8+7) &= (5-(8+7)) = -10\\ \max(5-8+7) &= ((5-8)+7) = 4\\ \min(4-8+9) &= (4-(8+9)) = -13\\ \max(4-8+9) &= ((4-8)+9) = 5 \end{aligned}$$

شکل ۶.۱۲: بررسی مثال برای مسئله قرار دادن پرانتز

با توجه به شکل بالا در می یابیم که بیشینه عبارت $(5-8+7)\times(4-8+9)$ برابر است با ضرب دو عدد منفی یعنی (-13 × -10) که برابر است با 130 خب همانطور که مشاهده می کنید ما برای بدست آوردن بیشینه این عبارت ابتدا باید کمینه دو بخش از آن را بدست آوریم پس همانطور که از این مثال متوجه شدید برای بدست آوردن بیشینه عبارت ما باید کمینه و بیشینه بخش های مختلف را بدست آوریم
اگر E یک زیرعبارت [۸] به صورت

$$E = d(i)op(i)...d(j)op(j)$$

در این صورت داریم
M(i,j)= maximum value of E
m(i,j)= minimum value of E
و رابطه بازگشتی آن ها به صورت زیر است

$$M(i,j) = \max_{i\le k\le j-1}\begin{cases} M(i,k) & op_k & M(k+1,j)\\ M(i,k) & op_k & m(k+1,j)\\ m(i,k) & op_k & M(k+1,j)\\ m(i,k) & op_k & m(k+1,j)\end{cases}$$

$$m(i,j) = \min_{i\le k\le j-1}\begin{cases} M(i,k) & op_k & M(k+1,j)\\ M(i,k) & op_k & m(k+1,j)\\ m(i,k) & op_k & M(k+1,j)\\ m(i,k) & op_k & m(k+1,j)\end{cases}$$

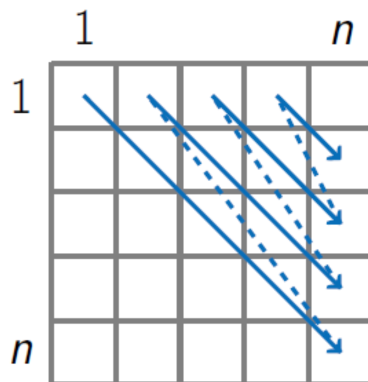
پس ما به دو آرایه دو بعدی یکی برای کمینه و دیگری برای بیشنه نیاز داریم در نهایت الگوریتم محاسبه کمینه و بیشیته هر زیرعبارت به صورت زیر است

[۸] subexpression

```
MinAndMax(i, j)
min ← +∞
max ← −∞
for k from i to j − 1:
   a ← M(i, k)   op_k   M(k + 1, j)
   b ← M(i, k)   op_k   m(k + 1, j)
   c ← m(i, k)   op_k   M(k + 1, j)
   d ← m(i, k)   op_k   m(k + 1, j)
   min ← min(min, a, b, c, d)
   max ← max(max, a, b, c, d)
return (min, max)
```

شکل ۷.۱۲: الگوریتم محاسبه کمینه و بیشینه

خب برای حل مسئله ابتدا به سراغ ساده ترین زیرعبارت می رویم یعنی هر عدد بدون هیچ عملگری (i=j) در این زیرعبارت ها حاصل هر زیر عبارت برابر است با خود زیرعبارت یعنی خود عدد. زیرعبارت بعدی عبارتی است با دو عدد و یک عملگر بین آن ها (|j-i|=1) حالت بعدی برابر است با سه عدد و دو عملگر بین آنها (|j-i|=2) در این حالت این زیر عبارت مه به دو بخش به صورتی که یک بخش که شامل دو عدد و یک عملگر است با یک عملگر به بخش دیگر که یک عدد است که این دو بخش را نیز قبلا حل کرده ایم و همینطور الی آخر با توجه به این تعاریف آرایه های دوبعدی ما به صورت زیر خواهد بود



حال به بررسی یک مثال می پردازیم

Example: $5 - 8 + 7 \times 4 - 8 + 9$

	1	2	3	4	5	6
1	5	-3	-10	-55	-63	-94
2		8	15	36	-60	-195
3			7	28	-28	-91
4				4	-4	-13
5					8	17
6						9

m

	1	2	3	4	5	6
1	5	-3	4	25	65	200
2		8	15	60	52	75
3			7	28	20	35
4				4	-4	5
5					8	17
6						9

M

خب دو خانه مشخص شده در آرایه های دوبعدی کمینه و بیشینه یعنی $M(2,4)$ و $m(2,4)$ را مشاهده می‌کنید این خانه ها بیانگر زیرعبارت $8+7\times4$ یعنی از عدد شماره دو تا عدد شماره چهار خب این زیرعبارت شامل دو بخش می باشد که به صورت $4\times(7+8)$ یا $8+(7\times4)$ می باشد که در حالت اول حاصل $7+8$ را قبلا محاسبه کرده ایم که برابر است با ۱۵ در خانه های $M(2,3)$ و $m(2,3)$ و حاصل این عبارت را در عدد ۴ ضرب می کنیم که این عدد هم در جدول ما هست خانه های $M(4,4)$ و $m(4,4)$ پس حاصل زیرعبارت اول ما برابر با 15×4 یعنی ۶۰ می شود حال به سراغ حل زیر عبارت بعد می رویم زیرعبارت (7×4) که بیشینه و کمینه این عبارت را می توانیم در خانه های $M(3,4)$ و $m(3,4)$ پیدا کنیم که این دو حالت کمینه و بیشینه باهم برابر و مساوی ۲۸ می باشند و حاصل این عبارت با ۸ جمع شده و برابر با ۳۶ می باشد پس مقایسه این دو در می یابیم که خانه $M(2,4)=60$ و $m(2,4)=36$ خواهد بود در نهایت الگوریتم حل مسئله پرانتزگذاری به صورت زیر است

```
۱    private static long Parentheses(long[] d, string[] op)
۲    {
۳        long[] m  = new long[d.Length];
۴        long[] M  = new long[d.Length];
۵        for(var i=1; i < d.Length ; i++)
۶        {
۷            m[i,i] = d[i];
۸            M[i,i] = d[i];
۹        }
۱۰       for(var s = 1 ; s < d.Length-1; s++)
۱۱       {
۱۲           for(var i=1; i < d.Length-s ; i++)
۱۳           {
۱۴               var j = i + s;
۱۵               m[i,j] , M[i,j] = MinAndMax[i,j];
۱۶           }
۱۷       }
۱۸       return M[1,d.Length];
۱۹   }
```

جلسه ۱۳

برنامه نویسی پویا — ساختار داده ای آرایه

غزل زمانی نژاد - ۱۳۹۸/۸/۱۱

جزوه جلسه ۱۳ام مورخ ۱۳۹۸/۸/۱۱درس ساختمان‌های داده تهیه شده توسط غزل زمانی نژاد.

۱.۱۳ مسئله ی پیدا کردن بیشترین مقدار یک عبارت ریاضی با پرانتزگذاری:

اصل ایده ی این مسئله روش تقسیم و حل است اما اگر به روش تقسیم و حل پیاده سازی شود، برخی از مقادیر چندین بار محاسبه شده اند. پس بهتر است به کمک برنامه نویسی پویا پیاده سازی شود.
روش حل: ابتدا دو ماتریس n*n ، یکی برای بیشترین مقادیر در یک محدوده و دیگری برای کمترین مقادیر تشکیل می دهیم. سپس همه ی حالات پرانتزگذاری را امتحان می کنیم و دو ماتریس را با بیشترین و کمترین مقادیر پر می کنیم. بدین صورت که:
قطر اصلی هر دو ماتریس را با همان اعداد پرمی کنیم. سپس برای پر کردن سایر خانه ها، با توجه به شماره ی آن خانه، حالات مختلف پرانتزگذاری پیش می آید که هر یک از حالت ها می تواند ۴ مقدار داشته باشد. به دلیل تقارن هر یک از این دو ماتریس تنها پر کردن قطر اصلی و خانه های بالای آن کافی است. برای توضیح بیشتر چگونگی پر کردن ماتریس ها، به بررسی مثال زیر می پردازیم:

Example: $5 - 8 + 7 \times 4 - 8 + 9$

5	-3	-10	-55	-63	-94
	8	15	36	-60	-195
		7	28	-28	-91
			4	-4	-13
				8	17
					9

m

5	-3	4	25	65	200
	8	15	60	52	75
		7	28	20	35
			4	-4	5
				8	17
					9

M

شکل ۱.۱۳: مثال نحوه پر کردن ماتریس ها

به طور مثال برای پر کردن خانه (2,5) ، ۳ حالت پرانتزگذاری زیر پیش می آید:

1. (2) + (3,4,5)

```
• Max (2,2) + Max (3,5) = 8 + 20
• Max (2,2) + Min (3,5) = 8 + -28
• Min (2,2) + Max (3,5) = 8 + 20
• Min (2,2) + Min (3,5) = 8 + -28
```

2. (2,3) * (4,5)

```
• Max (2,3) * Max (4,5) = 15 * -4
• Max (2,3) * Min (4,5) = 15 * -4
• Min (2,3) * Max (4,5) = 15 * -4
• Min (2,3) * Min (4,5) = 15 * -4
```

3. (2,3,4) – (5)

```
• Max (2,4) - Max (5,5) = 60 - 8
• Max (2,4) - Min (5,5) = 60 - 8
• Min (2,4) - Max (5,5) = 36 - 8
• Min (2,4) - Min (5,5) = 36 - 8
```

در نهایت برای پر کردن خانه (2,5) ماتریس مقادیر مینیمم، کمترین مقدار ۱۲ حالت بالا، و برای پر کردن خانه (2,5) ماتریس مقادیر ماکسیمم، بیشترین مقدار حالات بالا را در نظر می گیریم.

شبه کد این مسئله به صورت زیر است:

```
Data: i,j
Result: min,max
 min ← ∞
 max ← −∞
 for k from i to j-1 do
   a ← M(i, k) op_k M(k + 1, j)
   b ← M(i, k) op_k m(k + 1, j)
   c ← m(i, k) op_k M(k + 1, j)
   d ← m(i, k) op_k m(k + 1, j)
   min ← min(min, a, b, c, d)
   max ← max(max, a, b, c, d)
end
return min,max
```

Algorithm 19: MinAndMax(i, j)

```
Data: M,m,n
Result: M(1,n)
for i from 1 to n do
  m(i, i) ← d_i, M(i, i) ← d_i
end
for s from 1 to n-1 do
  for i from 1 to n-s do
    j ← i + s
    m(i, j), M(i, j) ← MinAndMax(i, j)
  end
end
return M(1,n);
```

Algorithm 20: Parentheses(d_1 op_1 d_2 op_2 . . . d_n)

پیچیدگی زمانی الگوریتم ۱، حداکثر n است.

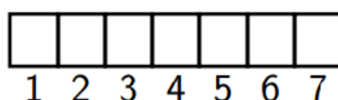
پیچیدگی زمانی الگوریتم ۲، حداکثر n^3 است.

۲.۱۳ ساختار داده: آرایه

آرایه بخشی از حافظه اصلی است که پیوسته بوده (یعنی عناصر آن در حافظه پشت هم قرار می گیرند.) و دارای اندازه مشخص می باشد. دسترسی به هر عنصر آرایه از $O(1)$ است. چون با pointer arithmetic میتوان محل هر عنصر را محاسبه کرد.

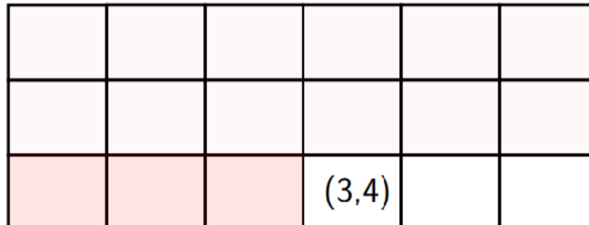
Constant-time access

$$\text{array_addr} + \text{elem_size} \times (i - \text{first_index})$$



شکل ۲.۱۳: دسترسی به عناصر در آرایه یک بعدی

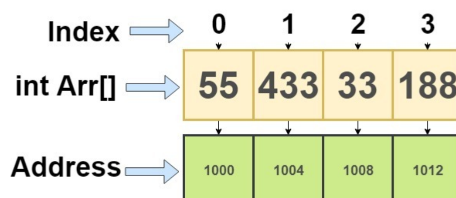
Multi-Dimensional Arrays



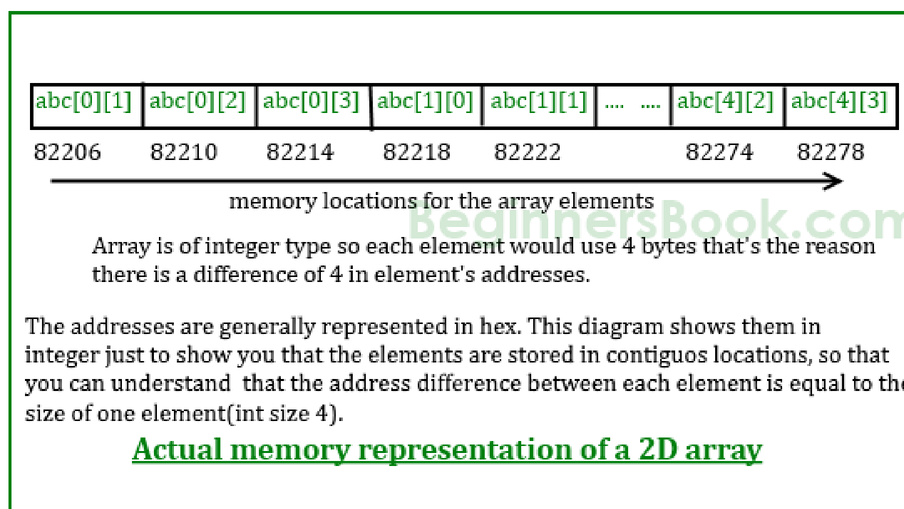
$$\text{array_addr} + \text{elem_size} \times ((3-1) \times 6 + (4-1))$$

شکل ۳.۱۳: دسترسی به عناصر در آرایه چند بعدی

برای محاسبه مقدار حافظه ای که هر آرایه اشغال می کند، باید سایز آرایه را در مقدار حافظه ای که هر المان آرایه به خود اختصاص می دهد ضرب کنیم.
به طور مثال، حافظه ای که int array[100] اشغال می کند، برابر است با:
$100 * \text{sizeof(int)} = 400$



شکل ۴.۱۳: عناصر یک آرایه یک بعدی به همراه آدرس هایشان
[؟]



شکل ۵.۱۳: عناصر یک آرایه چند بعدی به همراه آدرس هایشان
[؟]

۳.۱۳ عملیات اضافه یا حذف کردن یک عنصر در آرایه:

- پیچیدگی اضافه کردن عنصر به انتهای آرایه $O(1)$ است چون اگر در انتهای آرایه جای خالی داشته باشیم، می توان یک عنصر اضافه کرد. پاک کردن از انتها نیز به همین شکل است.
- پیچیدگی اضافه کردن عنصر به ابتدای آرایه از $O(n)$ است چون باید همه عناصر را یکی به جلو shift بدهیم. برای پاک کردن از ابتدا، باید بعد از پاک کردن اولین عنصر، بقیه عناصر را یکی به عقب shift بدهیم که این کار در $O(n)$ انجام می شود.
- اضافه یا حذف کردن از وسط آرایه نیز در $O(n)$ انجام می شود.

Summary

- Array: contiguous area of memory consisting of equal-size elements indexed by contiguous integers.
- Constant-time access to any element.
- Constant time to add/remove at the end.
- Linear time to add/remove at an arbitrary location.

شکل ۶.۱۳: خلاصه مبحث آرایه

جلسه ۱۴

لیست پیوندی

محمد مصطفی رستم خانی – ۱۳۹۸/۸/۱۳

جزوه جلسه ۱۴ام مورخ ۱۳۹۸/۸/۱۳ درس ساختمان‌های داده تهیه شده توسط محمد مصطفی رستم خانی. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

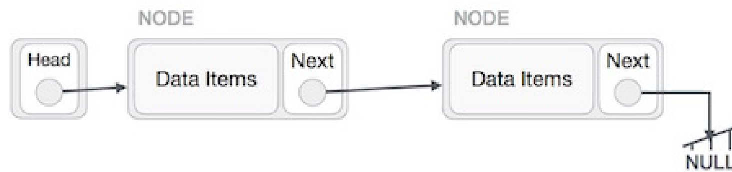
۱.۱۴ لیست پیوندی

نوعی ساختمان داده است که اعضای آن در جا های مختلفی از حافظه هستند و الزامی ندارد که پشت سر هم باشند. هر عضو از این ساختمان داده دارای حداقل دو ویژگی است. یکی مقدار آن و دیگری اشاره گری به عضو بعدی(و یا هم بعدی و هم قبلی) دنباله است. ما در این نوع ساختمان داده باید آدرس عنصر اول را نگه داریم و از آنجایی که هر عضو به عضو بعدی خود اشاره می کند و آخرین عنصر دارای اشاره گری به null است، اینگونه می توان به تمام عناصر این دنباله دسترسی پیدا کرد. این ویژگی لیست پیوندی باعث می شود که بتواند تعداد عناصر آن مختلف باشد و از آن کم کرد یا به آن اضافه کرد.گاهی اوقات اشاره گر به آخرین عنصر نیز نگه داری می شود. انواع لیست پیوندی عبارتند از:

- لبست پیوندی یک طرفه
- لیست پیوندی دو طرفه
- لیست پیوندی حلقوی

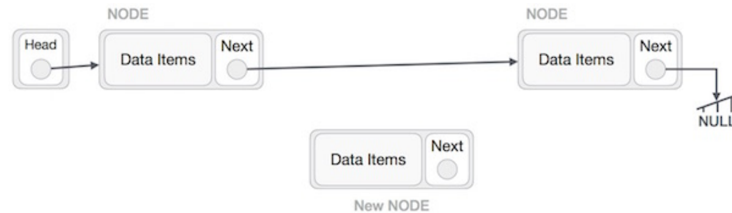
۲.۱۴ لیست پیوندی یک طرفه:

یک لیست پیوندی یک طرفه (Singly-linked list) دنباله ای از عناصر داده ای به نام گره (node) است که ترتیب خطی آنها توسط اشاره گرها تعیین می گردد. عناصر لیست تنها می توانند به ترتیب از ابتدای لیست تا انتها مورد دسترسی قرار بگیرند. هر گره آدرس گره بعدی را شامل می شود که به این صورت امکان پیمایش از یک گره به گره بعدی فراهم می شود. برای رسم لیست پیوندی گره ها به صورت مستطیل هائی پشت سرهم رسم می شوند که توسط فلش هائی بهم متصل شده اند. مقدار ثابت NULL برای علامت گذاری انتهای لیست در اشاره گر آخرین گره ذخیره می شود. لیست توسط یک اشاره گر Head که آدرس اولین گره لیست را در خود ذخیره می کند قابل دسترس است. بقیه عناصر توسط جستجوی خطی بدست می آیند.



شکل ۱.۱۴: لیست پیوندی

۳.۱۴ درج کردن:

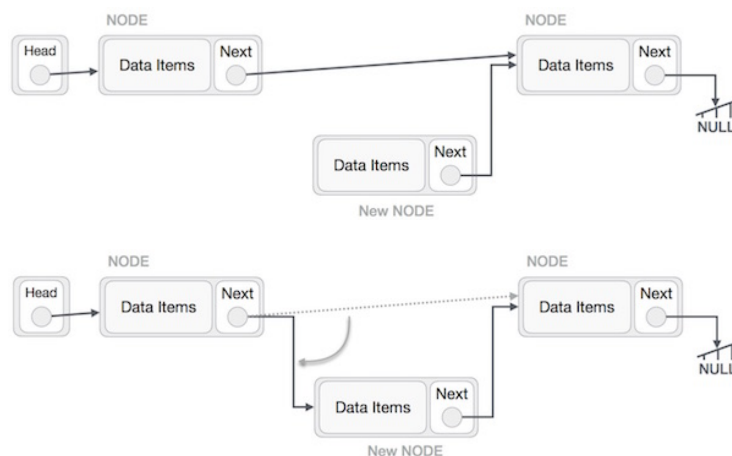


تصور کنید که می‌خواهیم یک گره B (گره جدید) را بین گره A (گره چپ) و گره C (گره راست) درج کنیم. در این صورت B باید به C به عنوان next اشاره کند:

```
NewNode.next -> RightNode;
```

این عملیات به صورت زیر خواهد بود:

حال گره سمت چپ باید به گره جدید اشاره کند:



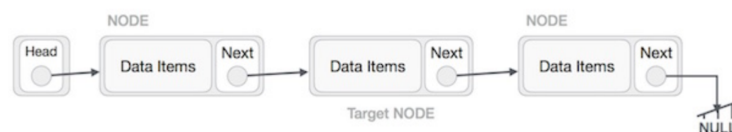

```
LeftNode.next -> NewNode;
```

بدین ترتیب گره جدید در میان دو گره قبلی قرار می‌گیرد. لیست جدید به صورت زیر خواهد بود:

اگر بخواهیم گرهی را در ابتدای لیست اضافه کنیم نیز مراحل مشابهی را طی می‌کنیم. زمانی که می‌خواهیم گرهی را در انتهای لیست درج کنیم، گره ما قبل آخر باشد به گره جدید اشاره کند و گره جدید به یک مقدار null اشاره می‌کند.

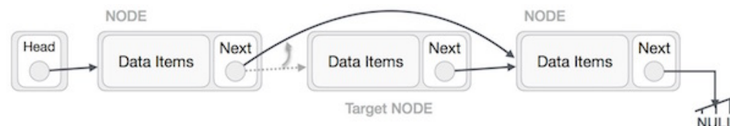
۴.۱۴ حذف کردن

ابتدا گره هدف که می‌خواهیم حذف کنیم را با استفاده از الگوریتم‌های جستجو می‌یابیم.



گره چپ (قبلی) گره هدف اینک باید به گره بعد از گره هدف اشاره کند:

```
LeftNode.next -> TargetNode.next;
```



با این کار پیوندی که به گره هدف وجود داشت از بین می‌رود. حال با استفاده از کد زیر موردی که گره هدف به آن اشاره می‌کرد را نیز حذف می‌کنیم:

```
TargetNode.next -> NULL;
```

```
PushBack(key):
node=new node
node.key=key
node.next=nil
if tail==nil then
|  head=tail=nil
else
|  tail.next=node
|  tail=node
end
```

Algorithm 21: PushBack singly-linked list

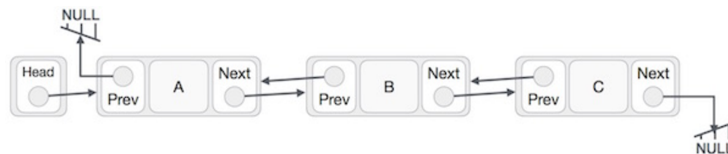
```
PopBack():
if head==nil then
|  ERROR:Empty list
end
if head==tail then
|  head=tail=nil
else
|  p=head
|  while p.next.next != nil do
|  |  p=p.next
|  end
|  p.next=nil
|  tail=p
end
```

Algorithm 22: PopBack singly-linked list

```
AddAfter(node,key):
node2=new node
node2.key=key
node2.next=node.next
node.next=node2
if tail==node then
 |  tail=node2;
end
```
Algorithm 23: AddAfter singly-linked list

۵.۱۴ لیست پیوندی دو طرفه

لیست پیوندی دو طرفه نوعی از لیست پیوندی است که حرکت در هر دو جهت یعنی به سمت جلو یا عقب در آن امکان‌پذیر است. یعنی هر عنصر علاوه بر آدرس عنصر بعدی به آدرس عنصر قبلی نیز اشاره می کند.




```
PushBack(key):
node=new node
node.key=key
node.next=nil
if tail==nil then
 |  head=tail=node
 |  node.prev=nil
else
 |  tail.next=node
 |  node.prev=tail
 |  tail=node
end
```
Algorithm 24: PushBack Doubly-linked list

```
PopBack():
if head==nil then
|   ERROR:Empty list
end
if head==tail then
|   head=tail=nil
else
|   tail=tail.prev
|   tail.next=nil
end
```

Algorithm 25: PopBack Doubly-linked list

```
AddAfter(node,key):
node2=new node
node2.key=key
node2.next=node.next
node2.prev=node
node.next=node2
if node2.next != nil then
|   node2.next.prev=node2
end
if tail==node then
|   tail=node2
end
```

Algorithm 26: AddAfter Doubly-linked list

```
AddBefore(node,key):
node2=new node
node2.key=key
node2.next=node
node2.prev=node.prev
node.prev=node2
if node2.prev != nil then
|   node2.prev.next=node2
end
if head==node then
|   head=node2
end
```

Algorithm 27: AddAfter Doubly-linked list

[؟] [؟]. [؟].

جلسه ۱۵

صف و پشته

محمد صدرا خاموشی فر – ۱۳۹۸/۷/۱۸

جزوه جلسه ۱۵ام مورخ ۱۳۹۸/۷/۱۸ درس ساختمان‌های داده تهیه شده توسط محمد صدرا خاموشی فر. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

اگه خواستید پیاده سازی NET. روبرای STACK ببینید به سایت referencesource.microsoft.com[] مراجعه کنید

system.collections.genereic.stack را سرچ کنید.

- stack
- queue

تعریف پشته: نوع داده ای Abstract که عملگر های push ، key key،top وpop را دارد.

تعریف: Abstract نوع داده (ADT) نوعی (یا کلاس) برای اشیاء است که رفتار آنها توسط مجموعه ای از ارزش و مجموعه عملیات تعریف می شود.

```
Push (key) : add key to collection
Key top():returns most recently-added key
Empty():are there any elements?
Key pop():removes and return recently-added key
```

برای تشخیص درست بودن ترتیب پرانتز ها میتوان از stack استفاده کرد

```
Data: string
Result: return IsBalanced
Stack stack;
while not at end of string do
    if char in ['[' ,'('] then
        stack.Push(char);
    else
        if stack.Empty() then
            return false;
        else
            top=stack.Pop();
            if top='[' and char!=']' or top='(' and char!=')' then
                return false;
            else
                return stack.Empty();
            end
        end
    end
end
```

Algorithm 28: IsBlalnced code

ما میتوانیم stack را با Data Type های مختلف مثل array،Queue،Linkedlist نمایش بدهیم

با آرایه:

برای push() به انتهای آرایه اضافه میشود . برای اینکه ببینیم خالی هست یا نه از Empty() استفاده میکنیم . نمیتوان resize کرد آرایه را و اگه ارایه پر باشد و بخواهیم یه عنصر اضافه کنیم Error میدهد.

با لینک لیست:

با داشتن Head میتوان به ابتدای linkedlist اضافه push)) کرد.

در stack تمام عملیات ها با پیچیدگی زمانی ۱ انجام میشود.

و همچنین در stack هر عنصری که آخر وارد بشود اول هم خارج میشود.

صف :

نوع داده ای با متد های زیر :

```
Enqueue (key) : add key to collection
Dequeue():removes and return the last recently-added
Empty():are there any elements?
```

همه ی متد ها با پیچیدگی زمانی ۱ انجام میشوند.

و از قاعده ی FIFO پیروی میکند.

یعنی اینکه هر عنصری که آخر وارد میشود اول از بقیه هم خارج میشود.

پیاده سازی صف با لینک لیست:
برای پیادسازی با LinkedList با داشتن tail و head به انتهای linkedlist اضافه میکنیم.هنگام اضافه کردن به انتهای لیست اضافه کرده و هنگام برداشتن از ابتدای لیست بر میداریم . به اسلاید های جلسه ۱۵ صفحه ی ۱۲۲ مراجعه شود.

پیاده سازی با آرایه:
برای پیادسازی با array یه آرایه و دوپوینتر read (به ابتدای ارایه اشاره کرده)و پوینتر write (به آخرین خونه ای که جدیدا اضافه شده اشاره میکند) در نظر گرفته.هر بار که Enqueue میکنیم write را یه واحد جلو برده و هر بار که Dequeue میکنیم read را یه واحد جلوبرده.به اسلاید های جلسه ۱۵ صفحه ی ۱۶۴ مراجعه شود.

درخت :
از درخت برای کار های مختلفی استفاده میشود(مناطق جغرافیایی و ...).
تعریف درخت:یه درخت یا خالیه یا مجموعه ای از راس هاست که هر کدام یه کلید دارند با یه لیست از بچه ها.
ریشه:بالاترین راس درخت
بچه :یال مستقیمی که از والد خود دارد
جد:پدر یا پدر پدر یا ...
نوه:بچه یا بچه ی بچه یا ...
خواهر برادری:بچه هایی که والد مشترک دارند
برگ:راسی که بچه ندارد
گره داخلی:راسی که برگ نباشد
لول(level) :۱+تعداد یال های بین ریشه و راس
ارتفاع:حداکثر عمق زیر درخت راس معیین شده و دور ترین برگ

هر راس شامل یه کلید و لیستی از بچه هاش وپدرش است.در درخت دودویی هر راس شامل کلید و راس سمت چپ و راس سمت راست و راس پدر است.
برای محاسبه ارتفاع :

```
Data: tree
Result: return Height
Height(tree):
if tree==null then
 |  return 0;
else
 |  return 1+Max(Height(tree.left),Height(tree.right));
end
```

Algorithm 29: Height code

براس محاسبه سایز درخت :

```
Data: tree
Result: return Size
Size(tree):
if tree==null then
 | return 0;
else
 | return 1+Size(tree.left)+Size(tree.right);
end
```

Algorithm 30: Size code

پیمایش درخت:هر راسی را حداقل یه بار مشاهده کنیم. و دو نوع داریم

پیمایش اول عمق: ما قبل از کاوش در زیر یک درخت خواهر و برادر، کاملاً یک زیر درخت را بپیماییم

پیمایش اول سطح: درخت را لول به لول پیمایش میکنیم.

```
Data: tree
Result: traversal
leveltraversal(tree);
if tree is null;
 then
 | return;
else
 | Queue q;
 | q.Enqueue(tree);
 | while q is not null do
 | | node=q.Dequeue();
 | | print(node);
 | | if node.left!=null;
 | |  then
 | | | q.Enqueue(node.left);
 | | else
 | | | if node.right;
 | | |  then
 | | | | q.Enqueue(node.right)
 | | | else
 | | |
 | | | end
 | | end
 | end
end
```

Algorithm 31: Traversal code

مدل های پیمایش اول عمق:

InOrder
PreOrder
PostOrder

در روش Inorder برای هر راس ابتدا بچه ی سمت چپ را نوشته بعدش خودش را نوشته بعدش بچه ی سمت راست.

```
Data: tree
Result: print inorder traversal
InorderTraversal(tree):
if tree is null then
 | return ;
else
 | InOrderTraversal(tree.left) ;
 | Print(tree.key) ;
 | InOrderTraversal(tree.right);
end
```

Algorithm 32: InOrderTRaversal code

در روش preorder برای هر راس ابتدا خودش را نوشتهو سپس بچه چپ را نوشته بعدش بچه ی سمت راست را نوشته .

```
Data: tree
Result: print PreOrder traversal
PreOrderTraversal(tree):
if tree is null then
 | return ;
else
 | Print(tree.key) ;
 | PreOrderTraversal(tree.left) ;
 | PreOrderTraversal(tree.right);
end
```

Algorithm 33: PreOrderTRaversal code

در روش ابتداpostorder بجه ی سمت چپ وسپس بچه سمت راست را نوشته وسپس خودش را نوشته.

```
Data: tree
Result: print PostOrder traversal
PostOrderTraversal(tree):
if tree is null then
 | return ;
else
 | PostOrderTraversal(tree.left);
 | PostOrderTraversal(tree.right);
 | Print(tree.key);
end
```

Algorithm 34: PostOrderTRaversal code

Dfs برای هر جور درختی میشه به جز inorder traversal اما BFS برای هر نوع درختی میتوان اجرا کرد چه باینری باشد چه باینری نباشد.

جلسه ۱۶

آرایه های پویا و اصل سرشکن کردن

هادی شیخی - ۱۳۹۸/۸/۲۰

جزوه جلسه ۱۶ام مورخ ۱۳۹۸/۸/۲۰ درس ساختمان‌های داده تهیه شده توسط هادی شیخی. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند.

۱.۱۶ آرایه های پویا

یکی از مشکلات آرایه های عادی اندازه ثابت آنهاست. آرایه های پویا نوعی ساختمان داده با قابلیت تغییر سایز هستند. یک آرایه پویا معمولا با ایجاد یک آرایه ابتدایی در حافظه ایجاد میشود. عناصر آرایه پویا (اگر با ساختمان داده دیگری پیاده سازی نشده باشد) به صورت پیوسته در حافظه قرار دارند و هنگامی که دیگر جایی برای ذخیره عناصر نبود با توجه به پیاده سازی، حافظه جدیدی با اندازه بزرگتر برای ذخیره سازی تخصیص میدهیم و همه مقادیر قبلی را در آن مقداردهی میکنیم.

```
Pushback(value):
 if size = capacity then
   allocate newArray[2 * capacity];
   for i from 0 to size - 1 do
     newArray[i] = arrary[i];
   end
   free array;
   array = newArray;
   capacity = 2 * capacity;
end
array[size] = value;
size = size + 1;
```

Algorithm 35: Add a new item to dynamic array

۱.۱.۱۶ نرخ رشد

نرخ رشد در ارایه های پویا به عوامل متعددی وابسته است مانند اهمیت و پیچیدگی فضا و زمان، الگوریتم های تخصیص حافظه و ... نرخ رشد ایده آل برای ایجاد آرایه های جدید عدد طلایی ۱.۶۱۸۰۳۳ است که برای ساده سازی محاسبات در اکثر پیاده سازی ها نرخ رشد را ۲ در نظر گرفته اند.

۲.۱.۱۶ پیچیدگی زمان و کارایی

پیچیدگی زمان ارایه های پویا مشابه ارایه های عادی است:

- مقدار دهی یا تغییر عنصر در مکان خاصی از لیست (constant time)
- پیمایش لیست (linear time)
- وارد کردن یا حذف کردن عنصر در میان لیست (linear time)
- وارد یا حذف کردن عنصر در آخر لیست (constant amortized time)

نکته مهم که باید در هنگام کار با لیست ها و ارایه ها در نظر گرفت اینست که هنگام پاس دادن آنها به توابع باید آدرس ابتدای آنها را پاس داد نه کل لیست را، در غیر اینصورت پیچیدگی زمان الگوریتم برای لیست های به اندازه کافی بزرگ بالا میرود و الگوریتم کارایی خود را از دست میدهد.

۲.۱۶ اصل سرشکن

برای بیان اصل سرشکن روش ها متفاوتی ارائه شد به طور خلاصه درباره آن بحث شد.

۱.۲.۱۶ بیان میانگین

برای محاسبه هزینه هر عمل در دنباله ای از اعمال میتوان مجموع هزینه کل را بر تعداد اعمال انجام شده تقسیم کرد و هزینه میانگین برای انجام هر عمل را بدست آورد. با فرض c به عنوان هزینه میانگین برای هر عمل میتوان نوشت :

$$c = \frac{Cost(AllOperations)}{n}$$

حال از این متد برای محاسبه هزینه میانگین برای اضافه کردن عنصر جدید در یک آرایه پویا استفاده میکنیم. اینگونه بیان میکنیم که از آنجایی که نرخ رشد ما ۲ و اندازه ابتدایی آرایه ۱ است، برای عنصر هایی که در جایگاه بعدی توان های دو هستند باید یک آرایه جدیدی ساخته شود و طبق کد هایی که قبل تر ارائه شد برای ساخت آرایه جدید با طول n به طور تقریب باید تعداد n عملیات انجام شود.

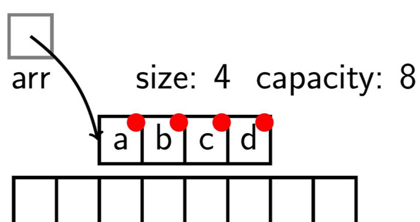
$$\text{let} c_i = \text{insertion i'th of cost}$$

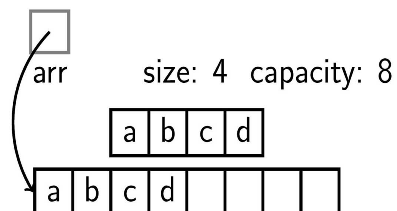
$$c_i = 1 + \begin{cases} i-1 & \text{2 of power a is 1 - i if} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n + \sum_{j=1}^{\lfloor \log_2^n \rfloor} 2^j}{n} = \frac{O(n)}{n} = O(1)$$

۲.۲.۱۶ بیان صندوقدار

اگر انجام هر عمل واحد را به عنوان خرج کردن یک سکه در نظر بگیریم ، با اضافه کردن یک عنصر یک سکه خرج میکنیم، حال فرض کنید در حین اضافه کردن عنصر جدید ما سه سکه خرج کنیم یعنی یکی برای اضافه کردن و دو سکه برای مکان جدید اضافه شده و یکی از مکان های نیمه اول آرایه ذخیره کنیم . برای ساختن آرایه جدید از سکه های ذخیره شده استفاده میکنیم.





شکل ۱.۱۶: ذخیره کردن و استفاده کردن از سکه ها

جلسه ۱۷

صف اولویت دار

حسن صبور – ۱۳۹۸/۸/۲۵

جزوه جلسه ۱۷ام مورخ ۱۳۹۸/۸/۲۵ درس ساختمان‌های داده تهیه شده توسط حسن صبور. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهشمند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

۱.۱۷ تعریف

صف اولویت‌دار (یا صف اولویتی – Priority Queue) از جمله ساختمان داده‌های بسیار پرکاربرد است. در صف عادی از تکنیک FIFO – مخفف First In First Out – استفاده می‌شود. در این تکنیک، مثل یک صف نانوایی، داده‌ها به ترتیب ورود پشت سر هم در صف قرار می‌گیرند. بنابراین اولین داده‌ی ورودی، اولین داده‌ی خروجی نیز خواهد بود. اما در صف اولویت‌دار برای هر داده، اولویتی – نه لزوما منحصربفرد – مشخص می‌شود. صف اولویت را می‌توان به اورژانس یک بیمارستان تشبیه کرد که هر بیمار با شدت بیماری بیشتر اولویت بیشتری برای رسیدگی دارد. سیستم عامل کامپیوتر هم برای مدیریت پردازش‌ها از صف‌های اولویت‌دار استفاده می‌کند.

به عنوان مثال، فرض کنید پردازش‌های زیر در انتظار اختصاص CPU به خود هستند:

شماره پردازش	۱	۲	۳	٤	۵	٦
اولویت	٤	۲	۱	۳	۵	٤

صف انتظار CPU یک صف اولویت‌دار است. در نتیجه CPU در اولین فرصت ممکن ابتدا پردازش شماره‌ی ۳ را انجام می‌دهد. سپس پردازش شماره‌ی ۲ و ...

تذکر: روش‌های زمان‌بندی CPU جهت انجام پردازش‌های مختلف یکی از بحث‌های جذاب و در عین حال مهم مبحث سیستم عامل است. بررسی تمامی روش‌های زمان‌بندی و مزایا و معایب آنها خارج از بحث

فعلی ما است.

برای پیاده‌سازی صف اولویتی عموما از آرایه استفاده می‌شود. من در اینجا سه روش مختلف را شرح می‌دهم.

۲.۱۷ پیاده‌سازی با استفاده از آرایه‌ی نامرتب

در این روش زمانی که داده‌ای وارد صف می‌شود، همچون صف عادی در انتهای آن قرار می‌گیرد. به عنوان نمونه، داده‌های مثال زمان‌بندی CPU به صورت زیر در آرایه قرار می‌گیرند:

	1	2	3	4	5	6
شماره پردازش	1	2	3	4	5	6
اولویت	4	2	1	3	5	4

هر عنصر آرایه ساختمانی مرکب از دو عنصر شماره‌ی پردازش و اولویت آن است. هر پردازش جدید به انتهای صف اضافه می‌شود که از مرتبه‌ی O(۱) است:

	1	2	3	4	5	6	7
شماره پردازش	1	2	3	4	5	6	7
اولویت	4	2	1	3	5	4	3

اما زمانی که قرار است پردازشی از آن خارج شود، باید تک تک عناصر بررسی شوند، تا پردازشی با بیشترین اولویت انتخاب شود. این فرآیند از مرتبه‌ی O(n) است.

۳.۱۷ پیاده‌سازی با استفاده از آرایه‌ی مرتب

در این روش بر خلاف روش قبل، آرایه بر اساس اولویت‌ها مرتب شده است.

	1	2	3	4	5	6
شماره پردازش	5	6	1	4	2	3
اولویت	5	4	4	3	2	1

زمانی که داده‌ای وارد صف می‌شود، بر اساس اولویت خود در محل مناسب قرار می‌گیرد:

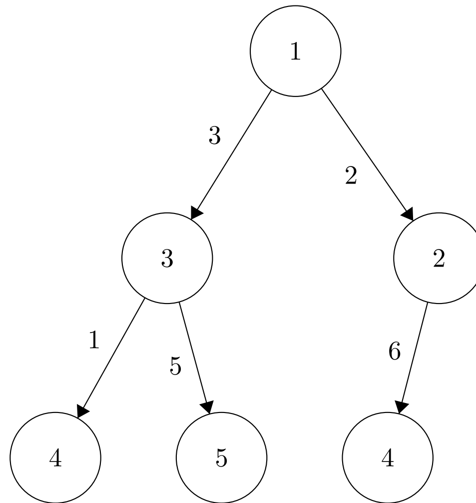
	1	2	3	4	5	6	7
شماره پردازش	5	6	1	7	4	2	3
اولویت	5	4	4	3	3	2	1

در این حالت پردازش با بیشترین اولویت همواره در انتهای صف قرار دارد و هزینه‌ی استخراج آن O(۱) است. این مسئله در مقایسه با آرایه‌ی نامرتب یک برتری است. اما در این روش هزینه‌ی درج O(n) است که در مقایسه با روش قبلی بدتر است.

در کل می‌توان گفت روش آرایه‌ی مرتب و نامرتب هم‌ارز یکدیگر بوده و از لحاظ عملکرد تفاوت چندانی با هم ندارند.

۴.۱۷ پیاده‌سازی با استفاده از آرایه‌ی نیمه‌مرتب

در این روش داده‌ها بر اساس اولویت آنها در یک درخت min–heap وارد می‌شوند:



اعداد داخل گره‌ها اولویت و اعداد خارجی شماره‌ی پردازش هستند. درخت فوق در نمایش آرایه‌ای به این صورت خواهد شد:

در یک درخت min–heap عنصری با کوچکترین کلید همواره در ریشه قرار دارد. در نتیجه عمل حذف گره ریشه از درخت min–heap، کوچکترین عنصر آن را به ما می‌دهد. این عمل بر اساس بحث‌های پیشین از مرتبه‌ی O(logn) است. عمل درج نیز در min–heap از همین مرتبه است.

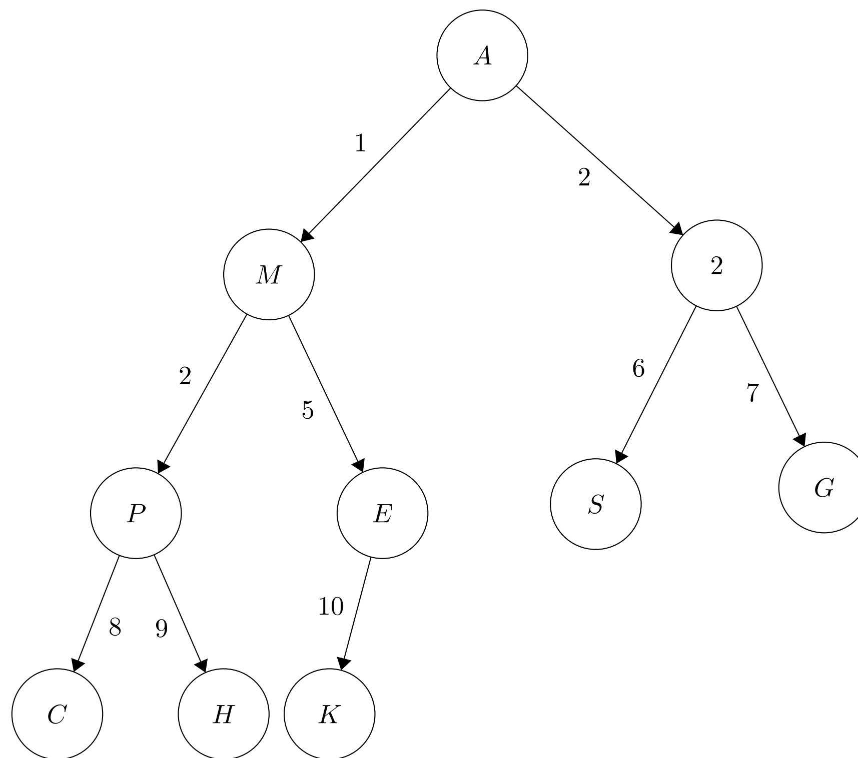
عملیات درج و حذف روی یک صف اولویتی که با استفاده از آرایه‌ی مرتب یا نامرتب ساخته شده باشد، روی هم رفته از مرتبه‌ی اجرایی n هستند. اما در روش آرایه‌ی نیمه‌مرتب این مرتبه به logn کاهش می‌یابد. پس می‌توان گفت که روش درخت هیپ برای پیاده‌سازی صف اولویتی کارآیی بسیار بهتری دارد.

	1	2	3	4	5	6
شماره پردازش	3	4	2	1	5	6
اولویت	1	3	2	4	5	4

۵.۱۷ heap

۱.۵.۱۷ تعریف

یک درخت دودویی کامل است، هرگاه تمامی سطوح درخت به غیر از احتمالا آخرین سطح پر بوده و برگ‌های سطح آخر از سمت چپ قرار گرفته باشند.
به یک مثال دقت کنید:



همانطور که مشاهده می‌کنید، تمامی سطوح درخت به غیر از آخرین سطح به طور کامل پر و همه‌ی برگ‌های سطح آخر نیز در سمت چپ درخت هستند. در واقع تمامی برگ‌های درخت دودویی کامل در دو سطح آخر آن قرار دارند.

۲.۵.۱۷ نمایش درخت دودویی کامل

نمایش با استفاده از لیست پیوندی و آرایه دو شکل مشهور نمایش درخت دودویی در ساختمان داده‌ها است. در حالت عادی انتخاب یکی از این دو روش برای نمایش بهینه و با مصرف حافظه‌ی کمتر بسته به چیدمان عناصر درخت دارد. به عنوان مثال، در درخت‌های مورب روش نمایش با آرایه بدترین بازدهی و بیشترین مصرف حافظه را دارد. اما در درخت دودویی کامل این روش در مقایسه با روش لیست پیوندی بسیار بهینه‌تر است.

در روش استفاده از آرایه برای نمایش درخت دودویی، گره‌های درخت مطابق شکل فوق با شروع از ریشه و در هر سطح از چپ به راست به ترتیب شماره‌گذاری شده و مقدار هر کدام از گره‌ها با توجه به شماره‌ی آن در یکی از خانه‌های آرایه قرار می‌گیرد. برای درخت فوق داریم:

1	2	3	4	5	6	7	8	9	10
A	M	Q	P	E	S	G	C	H	K

در آرایه‌ی متناظر درخت دودویی کامل، از همه‌ی عناصر به صورت کامل استفاده شده و هیچ حافظه‌ی هرزی وجود ندارد (چرا؟). به همین خاطر این روش نمایش برای درخت کامل مناسب است.

فرض کنیم توابع Parent Left، و Right شماره‌ی یک گره را گرفته و به ترتیب شماره‌ی گره والد، فرزند چپ و فرزند راست را برگرداند. در این صورت با توجه به شکل فوق:

Right(i)=۲i+۱ ، Left(i)=۲i ، Parent(i)=[i/۲]

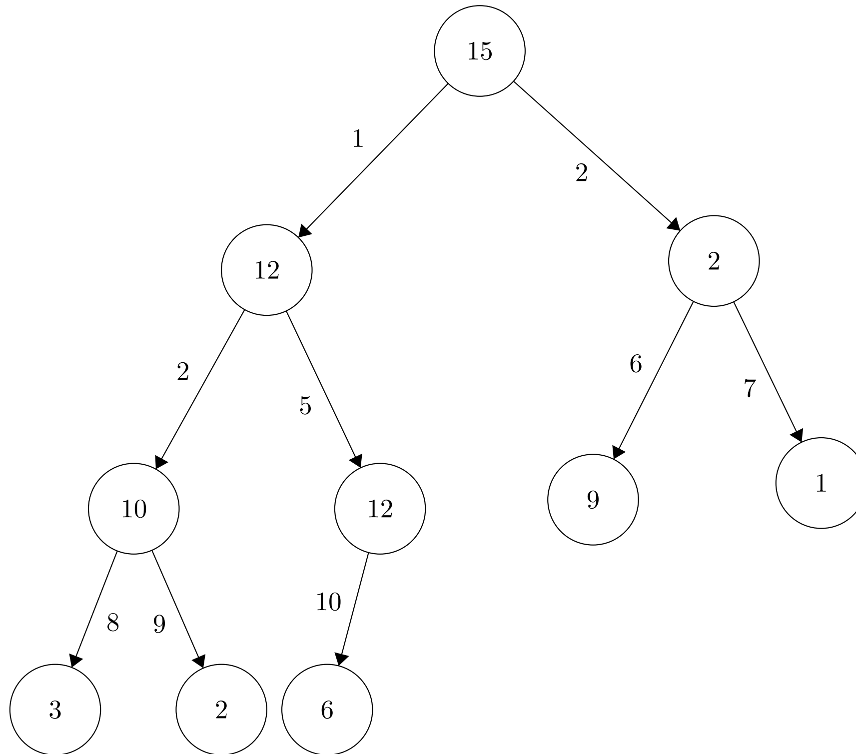
که منظور از [] جزء صحیح (کف) عدد است.

به عنوان مثال، در مورد گره شماره‌ی ۳ می‌توان نوشت:

Parent(۳)=[۳/۲]=۱،Left(۳)=۲×۳=۶،Right(۳)=۲×۳+۱=۷

۳.۵.۱۷ max heap

درخت دودویی کاملی است که مقدار هر گره بیشتر یا مساوی فرزندان خود است.

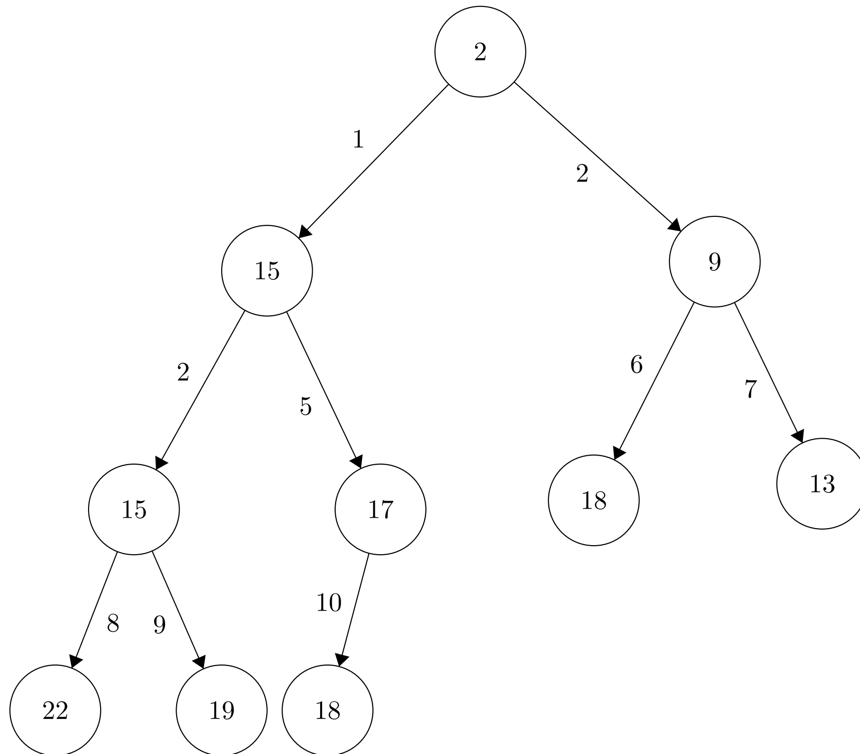


و نمایش آرایه‌ای:

	1	2	3	4	5	6	7	8	9	10
Max Heap	15	12	11	10	12	9	1	3	2	6

۴.۵.۱۷ min heap

درخت دودویی کاملی است که مقدار هر گره کمتر یا مساوی فرزندان خود است.



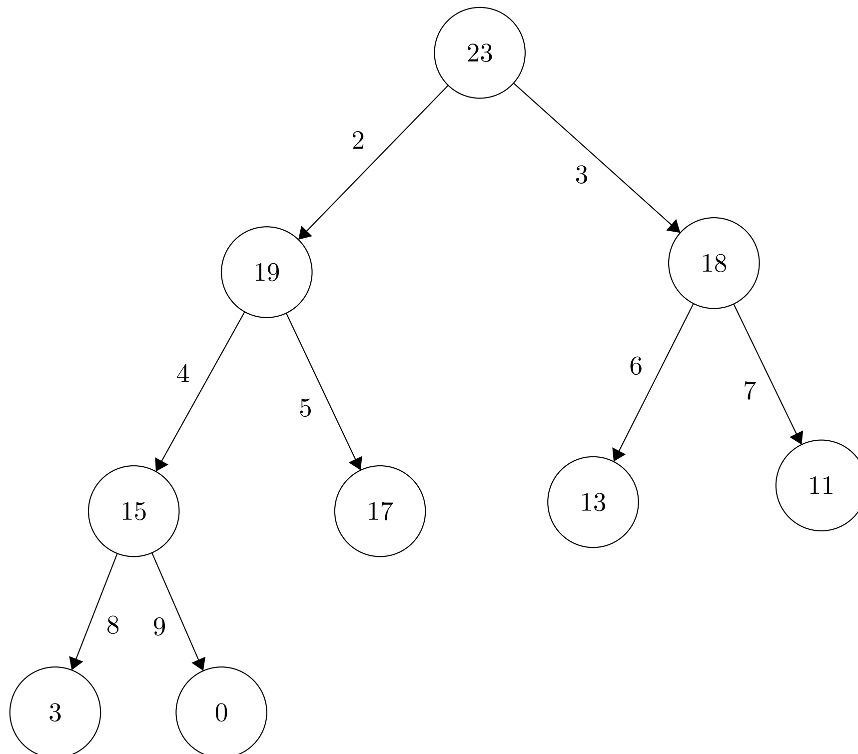
۵.۵.۱۷ ساختن heap

ساختن یک درخت heap در واقع وارد کردن متوالی گره‌ها در آن است. برای وارد کردن یک گره به درخت heap، طی دو مرحله به صورت زیر عمل می‌کنیم:

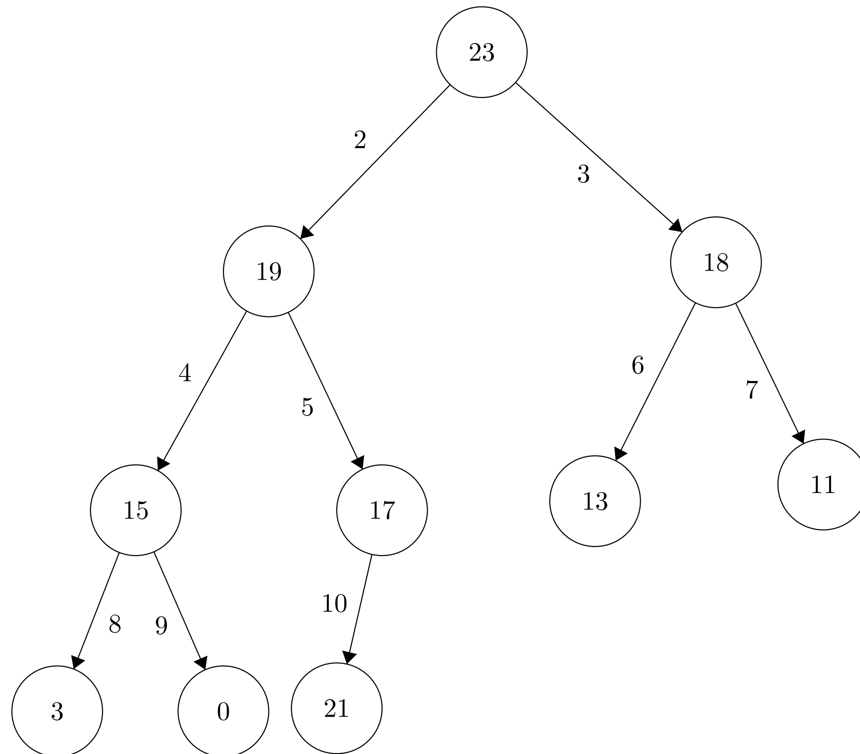
۱- گره مفروض را در محلی از درخت که شرط کامل بودن آن به هم نخورد (بدون در نظر گرفتن شرط max-heap یا min-heap بودن) درج می‌کنیم.

۲- اگر گره مذکور بر اساس موقعیت خود در درخت، شرط max-heap یا min-heap بودن را نقض نکند، نیاز به انجام کاری نیست و عملیات درج تمام شده است. در غیر اینصورت، با جابجا کردن گره با والد خود، درخت جدیدی حاصل می‌شود که باید مرحله‌ی ۲ در مورد آن تکرار شود.

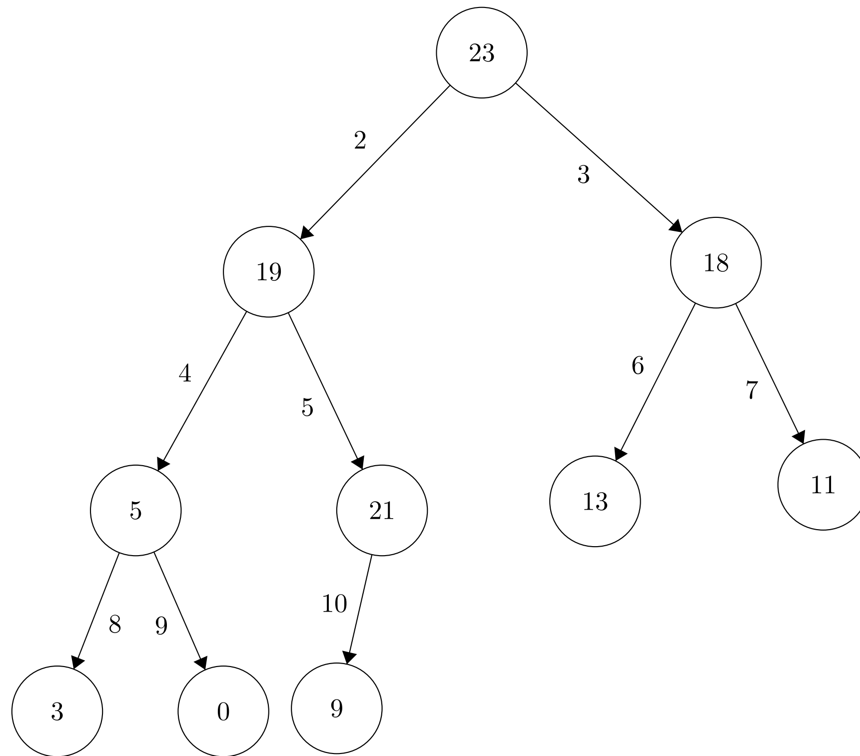
به عنوان مثال، فرض کنید یک درخت max-heap به فرم زیر داریم:



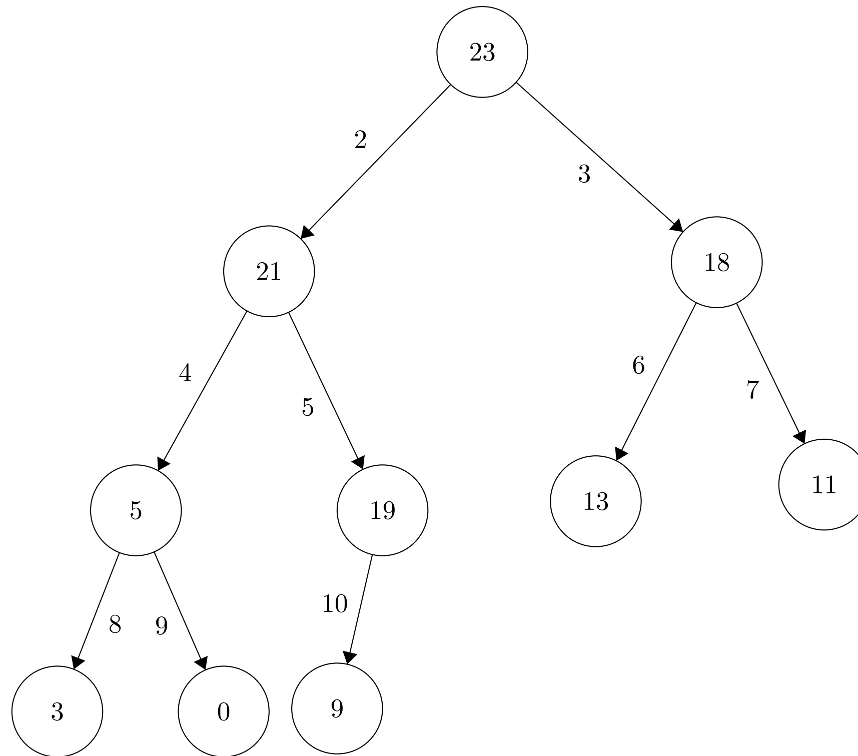
حال می‌خواهیم گرهی با مقدار ۲۱ را به درخت اضافه کنیم. برای اینکار در مرحله‌ی اول گره مذکور را به محلی که شرط کامل بودن درخت نقض نشود وارد می‌کنیم. این محل سمت چپ‌ترین فضای آزاد آخرین سطح درخت است:



با درج این گره، شرط max–heap بودن نقض می‌شود. چرا که مقدار گره شماره‌ی ۱۰ از والد خود یعنی گره شماره‌ی ۵ بیشتر است. پس با توجه به دستورالعمل مرحله‌ی دوم، مقدار دو گره را جابجا می‌کنیم:



با این عمل، باز هم شرط max-heap بودن برآورده نمی‌شود. گره‌های شماره‌ی ۵ و ۲ این شرط را نقض کرده‌اند. پس باز هم با تکرار مرحله‌ی دوم مقدار این دو گره را با هم جابجا می‌کنیم:



حال شرط max-heap بودن برقرار بوده و عملیات درج گره تمام می‌شود.
با توجه به این مثال می‌توان مرحله‌ی دوم عملیات درج را اینگونه بیان کرد:
۲- گره درج شده را با والدهای خود تا جایی‌که شرط max-heap یا min-heap بودن برقرار شود جابجا می‌کنیم.

۶.۵.۱۷ برنامه‌نویسی درج گره در درخت heap

در اینجا کد مربوط به درج گره در درخت max-heap را می‌آورم که با یک تغییر جزئی همین کد برای درخت min-heap هم قابل استفاده است.

همانطور که بحث شد، بهترین روش نمایش درخت heap استفاده از آرایه است. در مورد درخت max-heap اولیه فوق داریم:

	1	2	3	4	5	6	7	8	9
Max Heap	23	19	18	5	9	13	11	3	0

با اضافه کردن گره ۲۱ و طی کردن مراحل دوگانه درج گره:

	1	2	3	4	5	6	7	8	9	10
Max Heap	23	19	18	5	9	13	11	3	0	21

	1	2	3	4	5	6	7	8	9	10
Max Heap	23	19	18	5	21	13	11	3	0	9

	1	2	3	4	5	6	7	8	9	10
Max Heap	23	21	18	5	19	13	11	3	0	9

در قسمت قبلی رابطه‌ی ریاضی بین اندیس‌های والد و فرزند بیان شده است. بر اساس این رابطه و توضیحات فوق، تابع درج گره با مقدار v در یک درخت max-heap که در حال حاضر n عنصر (گره) دارد در زبان C++ به این صورت خواهد بود:

```
۱   void push(int heap[], int &n, int v)
۲   {
۳       int i,temp;
۴       heap[++n] = v;
۵       for(i = n; i > 1 && heap[i] > heap[i/2]; i/=2){
۶           temp = heap[i];
۷           heap[i] = heap[i/2];
۸           heap[i/2] = temp;
۹       }
۱۰  }
```

تذکر ۱: اندیس آرایه‌ها در زبان برنامه‌نویسی C++ از صفر شروع می‌شود. اما در اینجا برای راحتی کار و هماهنگ شدن با روش شماره‌گذاری درخت دودویی کامل، از اولین خانه - یعنی خانه‌ی شماره‌ی صفر - برای نمایش درخت heap استفاده نشده است.

تذکر ۲: در این تابع پارامتر n به صورت مرجع تعریف شده است که مختص زبان برنامه‌نویسی C++ بوده و در زبان C وجود ندارد. متغیرهای مرجع در یک نوشته به طور کامل توضیح داده شده است.

نکته: روش درج گره جدید در درخت heap در ظاهر شباهت‌هایی به درج گره جدید در درخت جستجوی دودویی (Binary Search Tree) دارد. اما این دو اختلاف‌های مشخصی دارند:

نکته: با توجه به قطعه کد بالا، مرتبه‌ی اجرایی عمل درج در درخت heap از مرتبه‌ی O(logn) است.

۷.۵.۱۷ حذف گره از درخت Heap

حذف گره از درخت هیپ عموما از ریشه‌ی آن صورت می‌گیرد. حذف گرهی غیر از گره ریشه، ممکن است هزینه‌ای معادل ساخت مجدد درخت تحمیل کند. چرا که با حذف یک گره غیر ریشه و جایگزین کردن گرهی دیگر با آن، نه تنها شرط heap بودن که شرط درخت کامل بودن هم ممکن است نقض شود. اکثر کاربردهای این نوع درخت نیز تنها با حذف گره از ریشه سر و کار دارند.

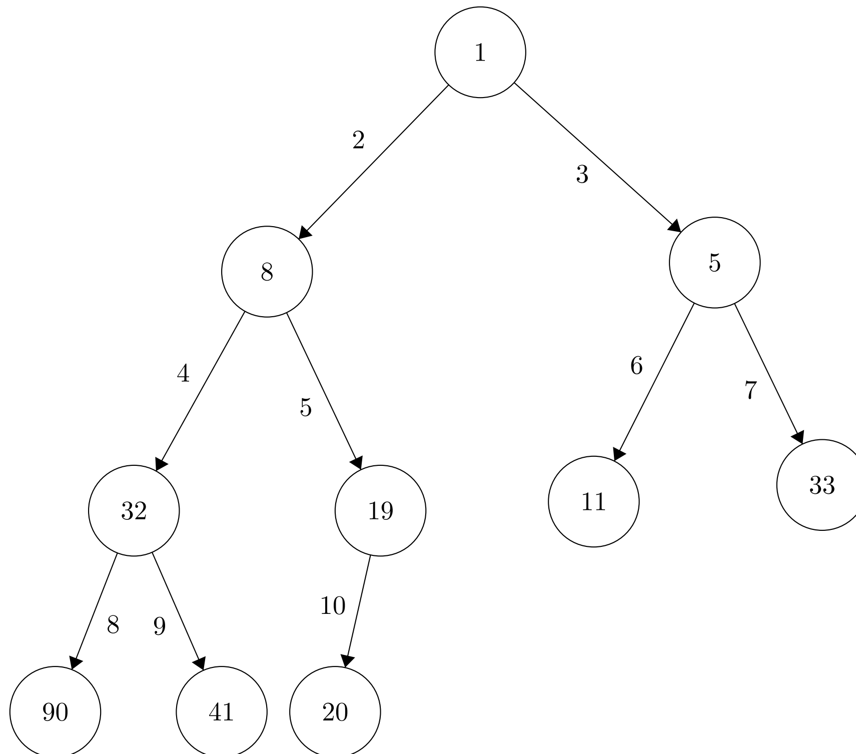
برای حذف گره ریشه‌ی درخت دو مرحله زیر را اجرا می‌کنیم:

۱- گره ریشه را حذف و سمت راست‌ترین برگ سطح آخر را جایگزین آن می‌کنیم.

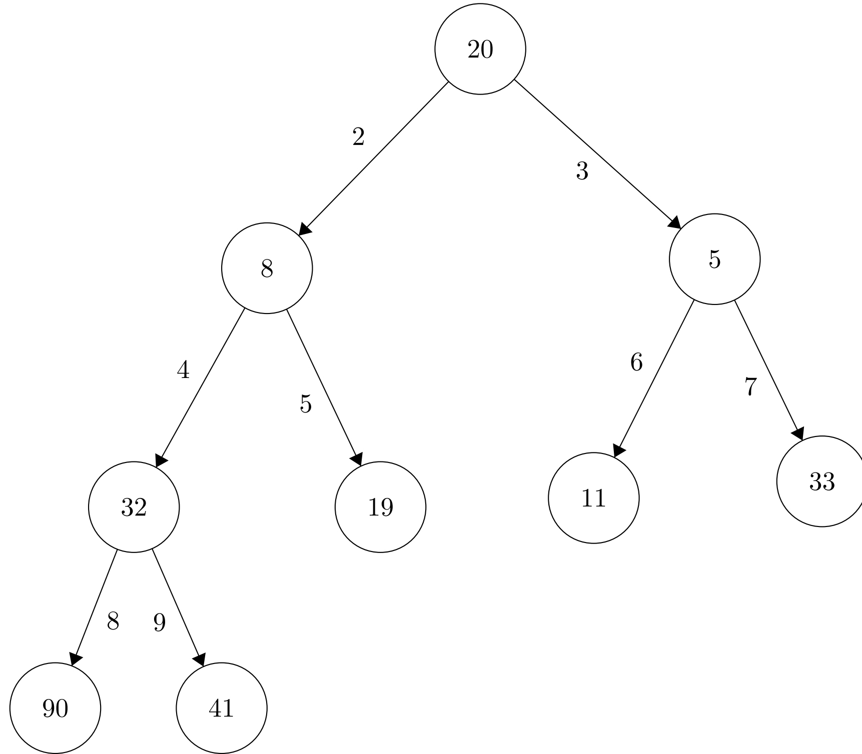
۲- در صورتی که گره درج شده جدید شرط heap بودن را نقض نکند عملیات حذف تمام می‌شود. در غیر اینصورت این گره با فرزند مناسب جایگزین شده و این مرحله برای درخت جدید مجددا اجرا می‌شود.

با اجرای مرحله‌ی اول و جایگزین کردن آخرین گره آخرین سطح درخت، شرط کامل بودن درخت پایدار می‌ماند. اما عموما شرط heap بودن نقض می‌شود. در مرحله‌ی دوم، گره تازه وارد را با یکی از فرزندان خود جایگزین می‌کنیم، تا به شرط heap بودن نزدیک شویم. اما کدام فرزند؟ پاسخ را با یک مثال مشخص می‌کنم.

فرض کنید قصد داریم گره ریشه‌ی درخت min-heap زیر را حذف کنیم:

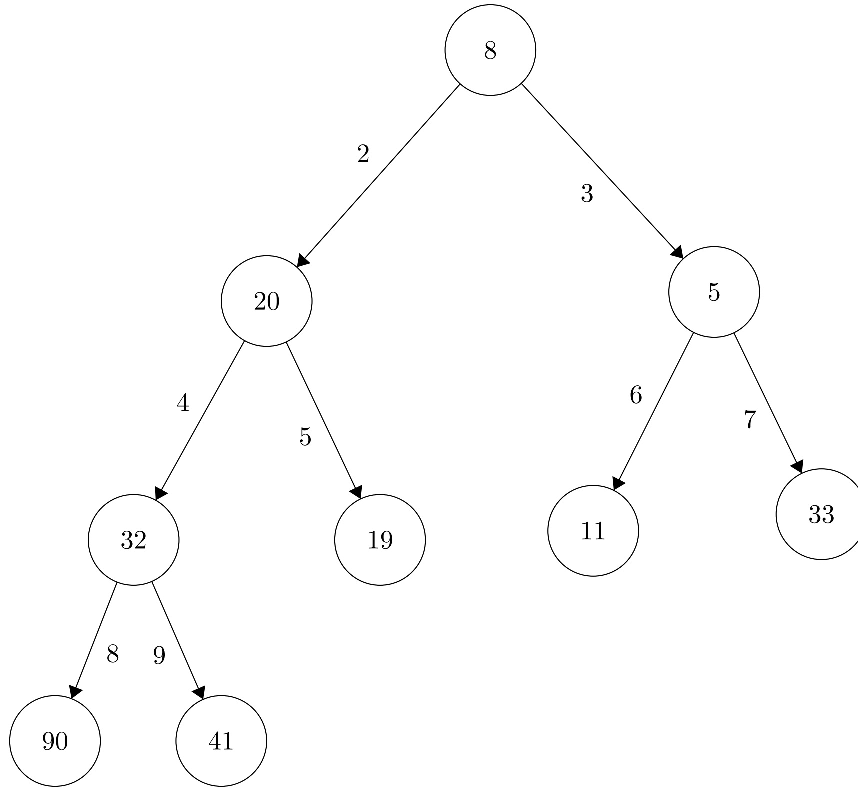


مرحله‌ی اول را اجرا کرده و گره شماره‌ی ۱۰ را جایگزین ریشه می‌کنیم:



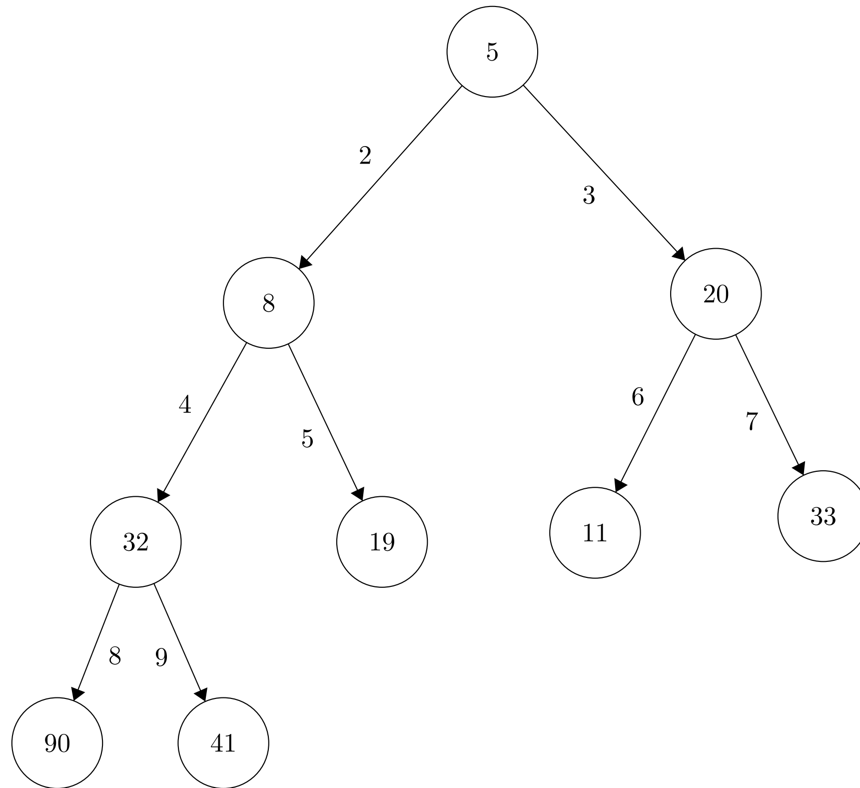
شرط درخت کامل بودن همچنان برقرار است. اما درخت فعلی min-heap نیست. چرا که ریشه از هر دو فرزند خود بزرگتر است. حال مطابق مرحله‌ی دوم باید یکی از فرزندان را با والد جابجا کنیم.

اگر فرزند چپ با مقدار ۸ را انتخاب کنیم:



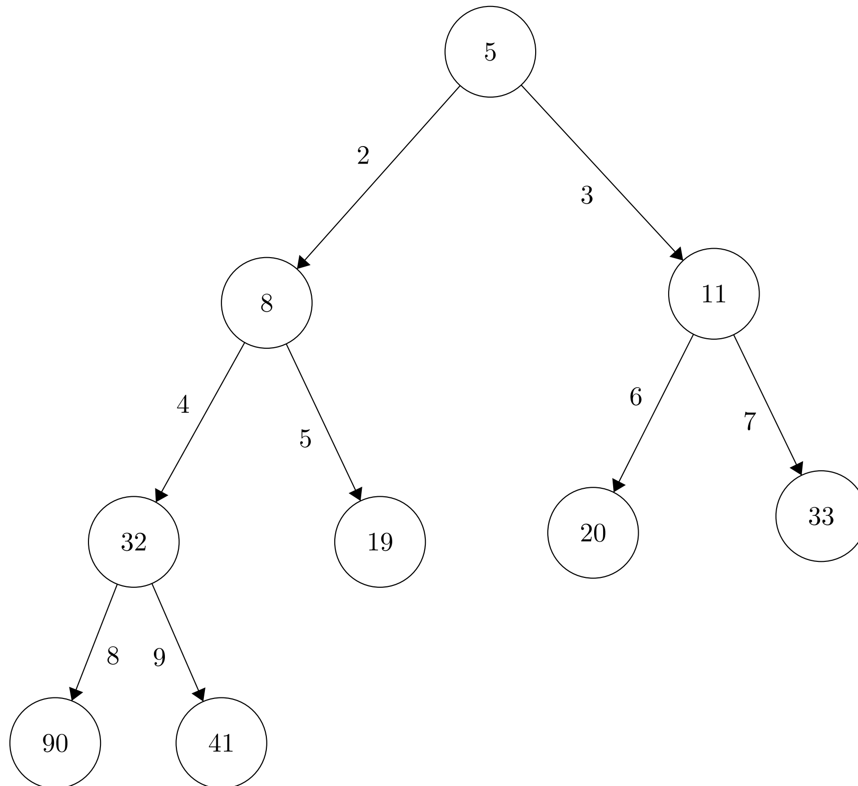
در این حالت، علاوه بر بحث مکان درست گره شماره‌ی ۲ با مقدار ۲۰، مشکل دیگری هم داریم: گره شماره‌ی ۱ و ۳ هم شرط min-heap را نقض می‌کنند.

اگر فرزند راست را انتخاب می‌کردیم:



در این حالت لااقل گره‌های شماره‌ی ۱ و ۲ مشکلی ندارند و تنها دغدغه‌ی ما محل درست گره شماره‌ی ۳ خواهد بود. پس نتیجه اینکه: در درخت min-heap، فرزندی را جایگزین والد می‌کنیم که مقدار کوچکتری داشته باشد. این مسئله در مورد max-heap به صورت عکس است. یعنی فرزندی را در درخت max-heap جایگزین می‌کنیم که مقدار بیشتری دارد.

اما هنوز گره شماره‌ی ۳ شرط min-heap بودن را نقض می‌کند. پس با تکرار مرحله‌ی دوم و با توجه به نتیجه‌گیری فوق، این گره را با گره شماره‌ی ۶ جابجا می‌کنیم:



به این ترتیب شرط min–heap بودن نیز برقرار شده و عملیات حذف گره به اتمام می‌رسد.

۸.۵.۱۷ برنامه‌نویسی حذف گره از درخت heap

این عملیات برای درخت فوق در نمایش آرایه‌ای به فرم زیر خواهد شد:

	1	2	3	4	5	6	7	8	9	10
Min Heap	2	8	5	32	19	11	33	90	41	20

	1	2	3	4	5	6	7	8	9
Min Heap	20	8	5	32	19	11	33	90	41

	1	2	3	4	5	6	7	8	9
Min Heap	5	8	20	32	19	11	33	90	41

	1	2	3	4	5	6	7	8	9
Min Heap	5	8	11	32	19	20	33	90	41

بر اساس روابط ریاضی بین شماره‌ی اندیس گره‌های والد و فرزند، تابع pop برای حذف گره ریشه به این ترتیب خواهد بود:

```
int pop(int heap[], int &n)
{
    int i =1,result,temp,min;
    result = heap[1];
    heap[1] = heap[n--];
    while(2 * i <= n){
        min = 2*i;
        if(min + 1 <= n && heap[min + 1] < heap[min])
            min++;
        temp = heap[i];
        heap[i] = heap[min];
        heap[min] = temp;
        i = min;
    }
    return result;
}
```

جلسه ۱۸

Disjoint Sets

امید میرزاجانی - ۱۳۹۸/۸/۲۰

جزوه جلسه ۱۸ام مورخ ۱۳۹۸/۸/۲۰ درس ساختمان‌های داده تهیه شده توسط امید میرزاجانی. این ساختمان داده، کاربرد های بسیاری دارد که یک از آنها تشخیص دادن هم گروه بودن تو راس در یک گراف است. به طور مثال وقتی میخواهیم ببینیم آیا از راس a به راس b مسیر هست یا خیر، میتوانیم از این ساختمان داده استفاده کنیم.

۱.۱۸ عملیات ساپورت شده Disjoint Sets

اعمال ویژه این ساختمان داده به شرح زیر است:

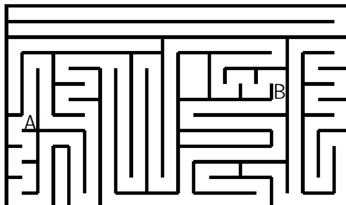
- MakeSet(x): که عنصر x را به یک مجموعه خاص مرتبط میکند. در واقع به آن عنصر یک ID ویژه میدهد.
- Find(x): ID آن عنصر را به عنوان خروجی میدهد. از این متد میتوان فهمید که آیا دو عنصر a،b داخل یک مجموعه قرار میگیرند یا خیر.

```
Data: Two elements (a , b)
Result: Check are these elements are in one set or not
initialization;
if Find(a)==Find(b) then
 | Return "yes";
else
 | Return "no";
end
```

- Union(x،y): این متد دو مجموعه را با هم ادغام میکند. در واقع بعد از اجرا شدن این متد، همه عناصری که با x هم گروه بودند؛ با تمام عناصری که هم گروه y اند، به یک محموعه تبدیل میشوند.

۲.۱۸ یک مثال از Disjoint Sets

فرض کنید میخواهیم پیدا کنیم آیا از A میتوان به B رفت یا خیر؟



شکل ۱.۱۸: یک مثال معروف!

ابتدا برای هر خانه از این شکل متد Makeset را فراخوانی میکنیم تا ID منحصر به فرد آن شکل بگیرد. سپس به ازای هر خانه با استفاده از متد Find چک میکنیم که هر خانه با کدام ۴ خانه مجاورش در یک گروه قرار دارد. سپس همانطورکه در بالا توضیح داده شد، چک میکنیم که آیا دو عنصر A ، B در یک مجموعه قرار گرفته اند یا خیر.

۳.۱۸ پیاده سازی

این ساختمان داده را میتوان به صورت آرایه [۱] یا لینکدلیست [۲] پیاده کرد.

۱.۳.۱۸ با آرایه

همانطورکه واضح است برای پیاده سازی این ساختمان داده ابتدا باید سه متو ذکر شده در بالا را پیاده سازی کرد. برای پیاده سازی ابتدا یک آرایه در نظر میگیریم که هر خانه اش شامل ID آن عنصر است. هم چنین ID هر عنصر، مقدار کوچکترین عضو آن مجموعه است. اگر این آرایه را smallest بنامیم داریم:

Makeset(i) : اندیس شماره i آرایه smallest را برابر i قرار میدهد. (این متد با O(۱) قابل بررسی است.)

Find(i) : smallest[i] را بر میگرداند. (این متد با O(۱) قابل بررسی است.)

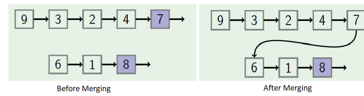
Union(i،j) : بین ID های i ، j کوچکترین ID را minID مینامیم. سپس در آرایه smallest پیمایش میکنیم و هر خانه ای که برابر Find(i) یا Find(j) بود، برابر minID قرار میدیم. (این متد با O(n) قابل بررسی است، زیرا با یک حلقه تمامی اعضای smallest را پیمایش کردیم و در هر بار یک عمل(Find) با اردر ۱ انجام دادیم.

۲.۳.۱۸ با لینکدلیست

در این نوع از پیاده سازی هر عضو را داخل یک لینکد لیست میگذاریم.
و همچنین برای متد Find آخرین عضو آن لینکد لیست را در نظر میگیریم.

[۱] Array
[۲] LinkedList

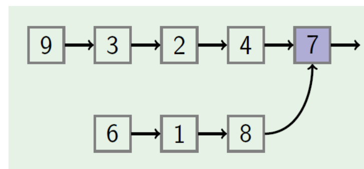
همچنین برای ادغام دو مجموعه که در اینجا همان ادغام کردن دو لینکدلیست است، اشاره گر آخرین عضو مجموعه اول را برابر اولین عضو مجموعه دوم قرار میدهیم.



شکل ۲.۱۸: ادغام سازی

۳.۳.۱۸ نوع دیگری از لینکدلیست

این روش با روش قبل شباهت بسیاری دارد با این تفاوت که برای ادغام کردن، مانند شکل عمل میکنیم.



شکل ۳.۱۸: ادغام سازی

اینگونه در نهایت یک شکلی مانند گراف جهت دار و بدون دور داریم و میتوان آن را به درخت تشبیه کرد که ID هر عضو ریشه ی درخت مرتبط است. این نوع پیاده سازی متد Union را بهبود میدهد ولی برای متد Find با O(n) انجام میدهد زیرا برای رسیدن به ID مورد نظر باید تا آخر لینکدلیست مورد نظر پیمایش کنیم.
ابتدا یک آرایه parent در نظر میگیریم که در آن هر عضو آرایه برابر پدر آن عضو است.
Makeset(i) : اندیس شماره i آرایه parent را برابر i قرار میدهد. (این متد با O(۱) قابل بررسی است.)

```
Data: i
while i!=parent(i) do
  | i=parent(i);
end
return i;
```

Algorithm 36: Find(i)

(این متد با O(tree height) قابل بررسی است.)
Union : برای ادغام دو مجموعه ریشه مجموعه اول را فرزند ریشه ی محموعه دوم قرار میدهیم. اما باید توجه داشته باشیم که برای یک درخت، هر چه ارتفاع آن کمتر باشد به برنامه ما بهبود میبخشد زیرا هر اطلاعاتی که بخواهیم از درخت بگیریم در زمان کمتری انجام میشود.
پس درختی را فرزند دیگری قرار میدهیم که در نتیجه ی کار ارتفاع درخت نهایی کمتر شود.

جلسه ۱۹

ادامه ی Hash + Disjoint sets Table

آرمین غلام پور - ۱۳۹۸/۹/۲

۱.۱۹ Disjoint sets

Disjoint sets : مجموعه هایی که اشتراکی با هم ندارند.
عملیات های موجود برای این ساختار داده :

- makeset : برای ساختن یک مجموعه ی جدید با یک عضو و با یک id مخصوص به همان مجموعه

```
MakeSet(x)
  smallest[i] <-- i
```

- find : پیدا کردن id یک عضو (پیدا کردن روت مجموعه ای که این عضو در آن قرار دارد)

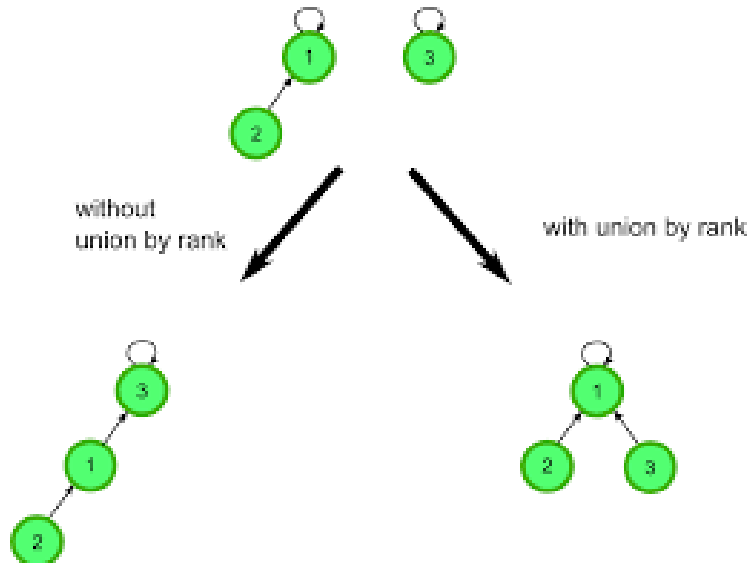
```
Find(x)
  return smallest[i]
```

- union(i,j) : مجموعه های شامل دو عضو i و j را پیدا میکند. سپس id تمامی عضو های یک مجموعه را به id مجموعه دوم تغییر میدهد.

```
Union(i, j)
i.id <-- Find(i)
j.id <-- Find(j)
if i.id = j.id:
  return
m <-- min(i.id, j.id)
for k from 1 to n:
  if smallest[k] in <i.id, j.id>:
    smallest[k] <-- m
```

union by rank : به روشی از union کردن گفته میشود که اردر آن از حالت عادی بهتر است و مقدار آن O(logn) میشود. در این روش root مجموعه ی کوچکتر را بچه ی root مجموعه ی بزرگتر میکنیم.
در این الگوریتم اثبات میشود که ارتفاع درخت حداکثر logn میشود.(اثبات با استقرا و استفاده از اینکه درختی که ارتفاع k دارد حداکثر ۲ به توان k نود دارد)

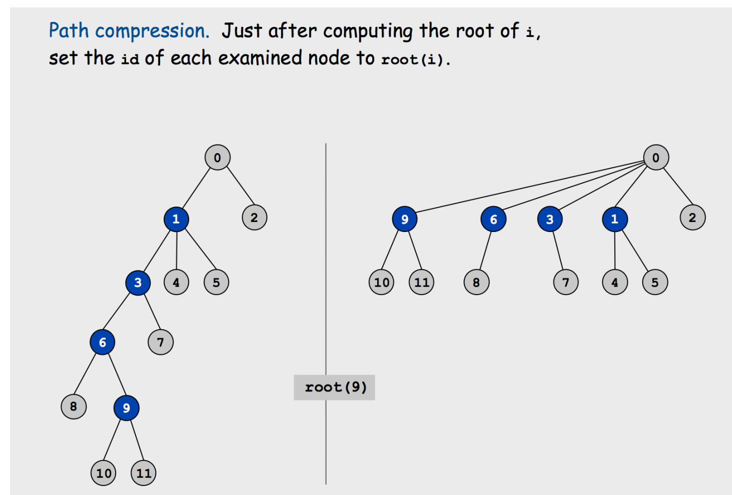
برای اینکه اردر عملیات ها کمتر شود از روش compression path استفاده میکنیم.



شکل ۱.۱۹: Union By Rank
[؟]

path compression : در این روش هر بار که متد find را صدا میزنیم، نود مورد نظر و تمامی نود هایی که در مسیر پیدا کردن root آن مجموعه هستند را، بچه ی root میکنیم. سر آخر آرایه ی parents را update میکنیم.
در این روش ارتفاع درخت ها کم و اردر عملیات نیز کمتر میشود. با این کار اردر عملیات ثابت (log*n) میشود. (log*n برابر است با تعداد دفعاتی که باید از n ، log بگیریم تا به ۱ برسیم. این عدد تا ۲ به توان ۶۵۰۰۰ حداکثر برابر ۵ خواهد شد)

```
Find(i)
if i != parent[i]:
  parent[i] <-- Find(parent[i])
return parent[i]
```



شکل ۲.۱۹: Path Compression
[؟]

موارد استفاده disjoint sets [؟] :

- برای دسته بندی مولفه های همبندی یک گراف بدون جهت
- برای تشخیص دادن دور در گراف
- برای محاسبه ی minimun spanning tree در الگوریتم kruskal
- در تولید و یا حل مساله های شامل maze
- پیدا کردن دوست های مشترک در روابط اجتماعی

visualizations for disjoint set + union by rank + path compression [؟]

۲.۱۹ Hash Table

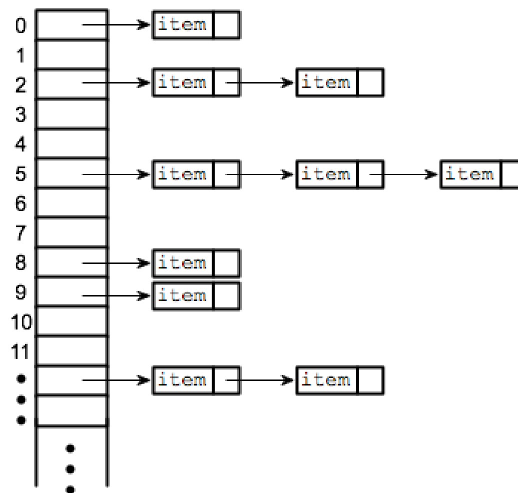
هر المان در یک hash table دارای دو ویژگی است :

- کلید
- مقدار

hash table یک ساختار داده ای است که داده ها را در غالب آرایه ای که خانه هایش بصورت لینک لیست هستند نگه داری میکند.

به اینصورت که برای هر داده به وسیله ی یک hash function یک عدد درست میکند که داده را هر چه که باشد (string, int, char, ...) به یک int یا long متناظر میکند.این عدد در واقع اندیس خانه ای از آرایه است که داده مورد نظر در لینک لیست موجود در آن خانه وجود دارد.

یک hash function خوب آن است که تعداد conflict ها در آن داده ها مینیم باشد.
conflict : یعنی دو داده ی متفاوت دارای مقدار hash code یکسان باشند.این باعث میشود که در آن خانه از آرایه که اندیسش برابر عدد بدست آمده است، دو داده (یا بیشتر) ذخیره شود.
اگر به conflict بخوریم باید در لینک لیست مورد نظر داده های جدید را به طوری ذخیره کنیم که هر خانه از لینک لیست هم دارای مقدار و هم کلید داده های کانفلیکت خورده باشد تا بتوانیم بعدا تفاوت داده های آن لینک لیست را متوجه شویم.



شکل ۳.۱۹: Hash Table
[؟]

موارد استفاده ی hash table :

- blockchain
- file system
- digital signature
- phone book

visualization for Hash Table [؟]

جلسه ۲۰

جدول هش / hash table

نگار زین العابدین - ۱۳۹۸/۹/۴

جزوه جلسه ۲۰ام مورخ ۱۳۹۸/۹/۴درس ساختمان‌های داده تهیه شده توسط نگار زین العابدین. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

۱.۲۰ نکته هایی از مطالب قبل

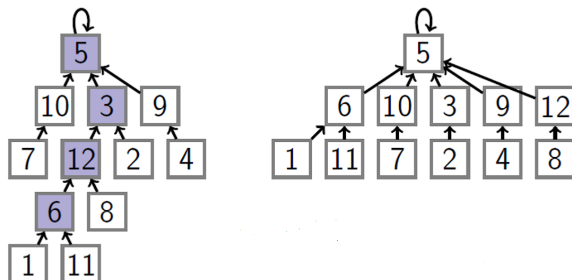
- در مبحث path copression ،وقتی از find استفاده می کنیم ، ساختار درخت ما عوض می شود.در حالی که تنها زمانی این اتفاق می افتاد ، (rank ما تغییر می کرد) موقع merge بود.چون وقتی به سمت ریشه حرکت می کنیم ، همه ی راس های سر راه را بچه ریشه قرار می دهیم.در نتیجه ارتفا ع کم تر می شود.(موقع find دست به rank نمی زنیم.)
- وقتی که n تا node داشته باشیم ، اون node هایی که rank شون k باشه ، حداقل ۲ به توان k تا node داریم.

۲.۲۰ function hash

- هش فانکشن به چه دردی می خورد؟

۱ Block Chain

۲ در زبان های برنامه نویسی ، برای hash function ، data structure هایی پیاده سازی شده است.برای مثال در زبان سی شارپ، dictionary و یا در جاوا ، hash map می باشد.



۳ serach کردن پوشه:به این صورت که وقتی نام پوشه را وارد می‌کنیم ، نام تبدیل به هش شده و سپس مکان پوشه جست و جو می شود.

۴ برای امضا رقمی کردن

۵ یکی دیگر از کاربرد های hash function ، این است که وقتی تلفن زنگ می خورد ، متوجه این بشویم که شماره متعلق به کیست. اگر دفترچه تلفن گوشی یک hash table داشته باشد ، با استفاده از هش شماره متوجه می شویم که شماره متعلق به چه کسی است و هم چنین بالعکس.(اسم را داریم و شماره اسم را می خواهیم.)

۳.۲۰ Direct Addressing

- همان طور که در بالا توضیح داده شد ، در ذخیره سازی شماره تلفن ها و پیدا کردن نام شماره مورد نظر ، می توانیم از روش Direct Addressing استفاده کنیم . در این روش برای این که به صورت بهینه کار ذخیره سازی و جست و جو را انجام دهیم ، می توانیم شماره را تبدیل کنیم به یک عدد مثلا ۷ رقمی و سپس یه آرایه به اندازه ۱۰ به توان ۷ درست کنیم و بعد به مکان آن عدد ۷ رقمی در آرایه برویم که آن جا نام شماره ذخیره شده است.

```
۱    private static string GetName(string phoneNumber)
۲    {
۳        var index = long.Parse(phoneNumber);
۴        return phoneBookArray[index];
۵    }
```

```
۱    private static void SetName(string phoneNumber, string name)
۲    {
۳        var index = long.Parse(phoneNumber);
۴        phoneBookArray[index] = name;
۵    }
```

*یکی از مشکلاتی که این روش دارد این است که فضای زیادی را در بر می گیرد.پس باید به دنبال روش دیگری بود. البته می توان از ساختمان داده هایی که تا الان خوانده ایم ، استفاده کنیم مانند Link List,Sorted Array,... . اما ممکن است که پیچیدگی محاسباتی بعضی از عملگر ها در پیاده سازی آن ها با استفاده از این ساختمان داده ها زیاد شود که ما این را نمی خواهیم.

- ایده ای در این باره وجود دارد ، این گونه است که برای اندازه آرایه خود یه مقدار ثابت در نظر بگیریم و سپس هر کدام از شماره ها را در خانه ای از آرایه خود قرار دهیم.در حقیقت وقتی hash function را روی داده خود صدا میزنیم ، یک عدد به دست می آوریم.حال داده خود را در خانه ای از آرایه قرار می دهیم که شماره اش با باقی مانده عدد به دست آمده بر اندازه آرایه (که ثابت در نظر گرفتیم) برابر باشد.

- در این ایده ، باید تلاش کرد که تا حد امکان تعداد collision ها کم تر باشد.

۴.۲۰ collision

- تعریف: اگر دو تا داده یا key متفاوت داشته باشیم که hash شون باهم برابر باشد ، در واقع به collision برخورده ایم.باید سعی کنیم که طوری hash function را پیاده سازی کنیم که تا حد امکان تعداد collision ها کم تر باشد.

۵.۲۰ Map

- در map تابع های زیر را داریم:
  - HasKey(object): چک می کند که آیا این کلید را داریم یا نه.در حقیقت تبدیل به هش کرده و سپس می رود به خانه مورد نظر و می بیند که آیا وجود دارد یا خیر.
  - Get(object) : value مناسب با کلید را به ما می دهد.
  - Set(object,value): معادل کلید را مقدار value قرار می دهد.
- implementation
  - در hash table ما آرایه ای از Link List ها داریم.
  - به پیاده سازی های زیر لطفا توجه کنید.
- پیچیدگی محاسباتی hash table ، اگر m اندازه آرایه و n مجموع تمام چیزهایی که در آرایه است باشد ، n+m می باشد.

```
۱   private static bool HashKey(object obj)
۲   {
۳       var chain = Chains[hash(obj)];
۴       foreach(var key,value in chain)
۵       {
۶           if(key == obj)
۷               return true;
۸       }
۹       return false;
۱۰  }
```

```
private static long Get(object obj)
{
    var chain = Chains[hash(obj)];
    foreach(var key,value in chain)
    {
        if(key == obj)
            return value;
    }
    return null;
}
```

```
private static void Set(object obj, long value)
{
    var chain = Chains[hash(obj)];
    foreach(var pair in chain)
    {
        if(pair.key == obj)
        {
            pair.value = value;
            return;
        }
    }
    chain.Append((obj,value));
}
```

۶.۲۰ Has Key

- می توان به جای hash table ، hash set داشت.ولی در این حالت ما فقط می خواهیم ببینیم که آیا key داده شده را داریم یا نه.
- به پیاده سازی های زیر لطفا توجه کنید.

```
private static void Add(object obj)
{
    var chain = Chains[hash(obj)];
    foreach(var key in chain)
    {
        if(key == obj)
            return;
    }
    chain.Append(obj);
}
```

```
private static void Remove(object obj)
{
    if(!Find(obj)):
        return;
    var chain = Chains[hash(obj)];
    chin.Erase(obj);
}
```

```
private static bool Find(object obj)
{
    var chain = Chains[hash(obj)];
    foreach(var key in chain)
    {
        if(key == obj)
            return true;
    }
    return false;
}
```

۷.۲۰ منابع بیش تر

- هش فانکشن چیست و چگونه یک هش فانکشن خوب انتخاب کنیم؟
https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-functio
- source reference
https://referencesource.microsoft.com/#q=hash

جلسه ۲۲

جدول هش

محتبی نافذ – ۱۳۹۸/۱۱/۹

جزوه جلسه ۲۲ام مورخ ۱۳۹۸/۱۱/۹ درس ساختمان‌های داده تهیه شده توسط محتبی نافذ. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

۱.۲۲ مقدمه ای بر تابع هش

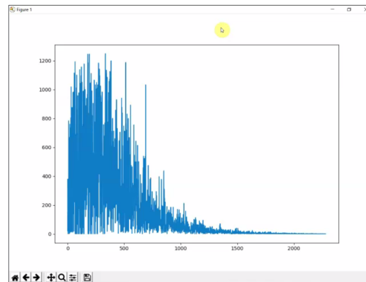
با توجه به مطالب جلسه گذشته این جلسه با ذکر چند نمونه تابع هش و مقایسه ی ان ها باهم مطالب را ادامه میدهیم
فرض کنید یک مجموعه از ۶۰۰ هزار رشته در اختیار داریم و میخواهیم یک تابع هش برای هش کردن این رشته ها پیاده سازی کنیم تا بتوانیم در یک جدول هش آن ها را دلخواه نگهداری کنیم
در اینجا چند تابع هش را بیان و نمودار مقدار هش (hash value)یکتا بر اساس تعداد collision هر hash value را رسم و توضیح میدهیم.

- یک ایده ی ساده جمع اعداد اسکی کاراکتر های موجود در رشته است

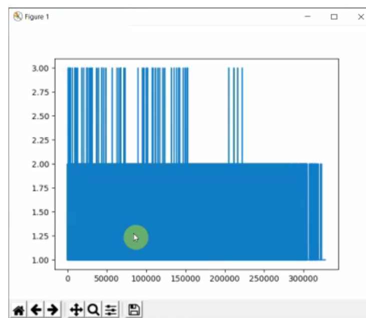
همانطور که می بینیم ۶۰۰ هزار رشته به حدود ۲۴۰۰ hash value یکتا متناظر شده که برای مثال در ۵۰۰ hash value حدود ۱۱۰۰ collisionرخ داده

- یک ایده ی دیگر هش کردن به صورت hash valueهای رندوم میباشد

همانطور که می بینیم این هش تعداد collisionبه شدت کم و پخش شده ای دارد که این عالی است اما یک ایراد بزرگ دارد این مزیت را از بین میبرد
زمانی که دو بار یک رشته را به آن میدهیم دو hash valueمتفاوت به ما میدهد پس در جدول هش ما هر مقداری را Insertکنیم نمیتوانیم دوباره ان را بازیابی کنیم



شکل ۱.۲۲: نمودار هش جمع کد اسکی



شکل ۲.۲۲: نمودار هش کاملا رندوم وار

آیا میتوانید یک تابع هش خوب برای رشته مثال بزنید؟

حال فرض کنید تعداد زیادی شماره تلفن داده شده و میخواهیم یک تابع هش برای متناظر نمودن ان ها به خانه های جدول هش پیاده سازی کنید.

- میتوان سه رقم اول هر شماره را به عنوان مقدار هش در نظر گرفت.

اما در این صورت همه ی کسانی که در یک منطقه زندگی میکنند ده یک خانه متناظر میشوند

- یک ایده ی دیگر هش کردن به صورت hash valueهای رندوم میباشد

همانطور که اشاره کردیم در این صورت مقادیر قابل باز یابی نیستند.

آیا میتوانید یک تابع هش خوب برای اعداد مثال بزنید؟

۲.۲۲ ویژگی های تابع هش خوب

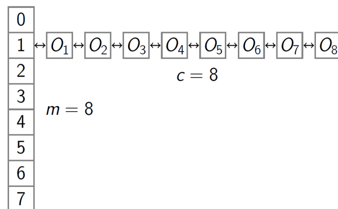
- یک مقدار را همیشه به یک hash valueمتناظر کند

- مقدار هش سریع محاسبه شود
- در تعداد مقادیر زیاد مقادیر را به صورت توزیع شده ای به هش های گوناگون متناظر کند
- collision آن کم باشد

نکته : باید توجه داشت که هر تابع هشی با توجه به یک مجوعه دیتا ممکن است خوب باشد یا بد اما ما باید طوری تابع هش پیاده سازی کنیم که عموما خوب باشد

۳.۲۲ denied of service attack (DOS)

DOS یک نوع حمله سایبری برای از از کار انداختن یا کند کردن پاسخگویی یک سرور است به طوری که کاربران به صفحه ی وب یا اینترنت یا ایمیل و ... دسترسی نداشته باشند.
اساس کار این حمله ایجاد collision زیاد است.
هر تابع هشی را میتوان ورودی هایی را داد که در نهایت همه ی ان ها به یک مقدار هش شوند شاید این کار سخت باشد اما امکان پذیر است



شکل ۳.۲۲: نوعcollisionبرای dos attack

در پارس کردن فایل ها در سرور ها بسیار از فایل jsonاستفاده میشود این فایل نوعی جدول هش است حمله کننده اگر بتواند تعداد زیادی collisionپیدا کند میتواند درکار سرور اخلال ایجاد نماید سرور با جستجو کردن در این LinkListدچار مشکل خوهد شد. شرط این کار این است که حمله کننده تعداد زیاد رشته از قبل به سرور بدهد و خروجی هش را ببیند و بتواند تعدادی رشته ی collisionدار پیدا نماید در سال ۲۰۱۱ شرکت مایکروسافت برای حل این مشکل امنیتی از UseRandomizedStringHashAlgorithmبرای غیر قابل پیش بینی کردن collisionها در پیاده سازی جدول هش استفاده کرد. که در ادامه توضیح خواهیم داد.

۴.۲۲ Universal Family

ایده : یک مجموعه از توابع هش داریم و هر دفعه به طور رندوم یکی از این توابع را انتخاب میکنیم
فرض کنید یک مجموعه ی Uداریم که شامل ۰،۱،۲،....،m-۱ است
حال hرا یک تابع هش در نظر بگیرید که هر ورودی را به عددی عضو Uمتناظر میکند.
حال فرض کنید Hیک مجموعه از این توابع هش h است یعنی:

```
{ { 1-m, ..., 2, 1, 0 } <- U : h } = H
```

H, Universal family است اگر برای دو ورودی مختلف احتمال collision کم تر از 1/m است

```
P[h(x)=h(y)]<=1/m
```

x , y عضو U و x != y

۵.۲۲ Load Factor

به نسبت تعداد مقادیر ذخیره شده در جدول هش به سایز کل جدول هش load factor می گویند
e = n / m
نکته : اگر جدول هش یک universal familyبود میانگین زمان دسترسی به طولانی ترین زنجیره (chain)از order O(1 + e) میباشد
کاربرد دیگر load factorبرای مدیریت حافظه است
به این صورت که سعی میشود این نسبت بین ۵.۰ تا ۹.۰ باشد اگر بیشتر شد باید برای جلوگیری از collisionسایز جدول هش را افزایش داد

[1] که شبه کد Rehash کردن در زیر امده است. ؟؟

```
Data: T
Result: new table
Rehash( T ):
loadFactor = T.numberOfKeys/T.size;
if not at end of this document then
    Create new Tnew of size, Tsize * 2;
    Choose hnew with cardinality of Tnew size;
    while T not empty do
        Insert object in Tnew using new hnew;
    end
end
T = Tnew, h = hnew;
```

Algorithm 37: How to Rehash hash table?

مرتبه ی زمانی :O(n)
مرتبه زمانی تحلیل زمانی : O(1)

۶.۲۲ یک تابع هش universal family برای اعداد

```
{ m mod ( p mod (ax+b) ) = hp(x) } = Hp
```

p یک عدد رندوم اسست.
a, b هر عددی بین صفر تا p-1 هستند
مثال : a = 34, b= 2, p=10 000 019 فرض کنید عدد مورد نظر x= 1 482 567باشد
(۲ + ۱۴۸۲۵۶۷ * ۳۴) mod ۱۰۰۰۰۰۱۹ = ۴۰۷۱۸۵

[1] pseudocode

۴۰۷۱۸۵ mod ۱۰۰۰ = ۱۸۵
۱۸۵ =h(x)

۷.۲۲ یک تابع هش universal family برای رشته ها

تیپ کلی این مجموعه از توابع هش :

```
{ p mod sum(S[i]pow(x,i)) = hp(s) } = Pp
```

i از صفر تا اندازه ی رشته منها ی یک می باشد. S نام رشته ورودی است
p یک عدد اول ثابت و xبین یک تا p-1 است.
این هش به Polynomial Hashing معروف است.

[۲] شبه کد PolyHash ؟؟

```
Data: S,p,x
Result: hash
hash = 0;
for i from |S| -1  down to 0 do
|  hash = ( hash * x + S[i] ) mod p;
end
return hash;
```

Algorithm 38: How to implement PolyHash hash function ?

مثال: |S| = ۳
۰ = hash
S[۲] mod p = hash
S[۲]x + S[۱] mod p = hash
S[۲]pow(x،۲) + S[۱]x + S[۰] mod p = hash

۸.۲۲ یافتن یک زیر رشته در یک رشته

سوال : یک زیررشته `P (pattern)` و یک رشته ی `T (Text)` داده شده و باید تمام ایندکس هایی که زیررشته در رشته قرار دارد را خروجی دهید.

راه حل معمولی :
مرحله ی اول یک تابع برای تشخیص برابری دو رشته مینویسیم:
[۳] شبه کد AreEqual ؟؟

---

[۲] pseudocode
[۳] pseudocode

```
Data: S1,S2
Result: boolean(true or false)
if |S1| != |S2| then
|  return False;
end
for i from 0 to |S1| -1  do
|  if  S1[i] != S2[i] then
|  |  return False;
|  end
end
return hash;
```
Algorithm 39: Are Equal tow string ?

زمان اجرای کد بالا O(|P|) است.
مرحله بعد پیاده سازی راه حل است :

[۴] شبه کد FindSubStringNative ؟؟

```
Data: T,P
Result: positions
positions = empty list;
for i from 0 to |T| - |P| do
|  if AreEqual(T[i..i+|P|-1], P) then
|  |  positions.Append(i);
|  end
end
return positions;
```
Algorithm 40: positions of pattern in text

زمان اجرای کد بالا O(|T||P|) است.
اما راه حل بهینه تر استفاده از هشینگ است.

۹.۲۲ الگوریتم RabinKarp

ایده : به جای مقایسه خود زیر رشته های رشته اصلی با الگو میدانیم اگر مقدار هش ان ها برابر نبود پس قطعا باهم برابر نیستند.
ولی اگر برابر بودند چاره ای جز مقایسه کاراکتر به کاراکتر نیست.
الگوریتم اولیه :

[۵] شبه کد initial RabinKarp ؟؟

---
[۴] pseudocode
[۵] pseudocode

```
Data: T,P
Result: positions
p = big prime , x = random(1, p-1);
positions = empty list;
pHash = PolyHash(P, p, x);
for i from 0 to |T| - |P| do
    tHash = PolyHash( T[i..i+|P|-1], p, x);
    if pHash != tHash then
        continue;
    end
    if AreEqual(T[i..i+|P|-1],P ) then
        positions.Append(i);
    end
end
return positions;
```

Algorithm 41: positions of pattern in text

هشدار : زمانی که است که دو زیر رشته باهم برابر باشند که در این صورت زمان اجرا بیشتر میشود ولی یادمان باشد که تعداد collision در یک هشuniversal family به شدت کم است

نکته : زمان اجرای الگوریتم بالا هم تقریبا همان O(|T||P|) است. برای سریع تر شدن چه کنیم؟

ایده RabinKarpزیر رشته های متوالی یک رشته مقدار هش شان بسیار شبیه هم بوده و میتوان ارتباطی برای زود تر حساب شدن هش ها پیدا کرد.

توجه کنید که تابع هش مورد استفاده PolyHash می باشد
Pp = { hp(s) = sum(S[i]pow(x،i)) mod p }

Consecutive substrings

$T =$ b e a c h

encode(T) = | 1 | 4 | 0 | 2 | 7 | $|P| = 3$

$h(\texttt{"ach"}) = 0+2x+7x^2$

$h(\texttt{"eac"}) = 4+0+2x^2$

$H[2] = h(\texttt{"ach"}) = 0 + 2x + 7x^2$

$H[1] = h(\texttt{"eac"}) = 4 + 0x + 2x^2 =$
$= 4 + x(0 + 2x) =$
$= 4 + x(0 + 2x + 7x^2) - 7x^3 =$
$= xH[2] + 4 - 7x^3$

شکل ۴.۲۲: ارتباط بین مقدار هش زیر رشته های متوالی

در نهایت رابطه ی بازگشتی زیر بین زیر رشته های متوالی برداشت میشود:

```
H[i] = x*H[i+1] + ( T[i] - (T[i+|P|] * pow(x, |P|)) mod p] )
```

که pow(x, |P|) را میتوان یکبار حساب و بعدا فقط استفاده نمود
برای شروع کد زنی باید قبل ااز حل تابعی پیاده سازی شود که با روش سریع و با استفاده از رابطه ی باگشتی کل هش زیر رشته ها را محاسبه کند.

شبه کد[۶] PreComputeHashes ؟؟

```
Data: T,|P|,p,x
Result: H
H = array of length |T| - |P| + 1 ;
S = T[|T| - |P| ... |T|-1];
H[|T| - |P|] = PolyHash(S, p, x);
y = 1;
for i from 1 to |P| do
 | y = (y*x) mod p;
end
for i from |T|-|P|-1 down to 0 do
 | H[i] = (x*H[i+1]+T[i]-y*T[i+|P|] mod p ;
end
return H;
```

Algorithm 42: hash of substring

حال ربین کارپ اصلی را بازنویسی میکنیم:

شبه کد[۷] main RabinKarp ؟؟

```
Data: T,P
Result: positions
p = big prime , x = random(1, p-1);
positions = empty list;
pHash = PolyHash(P, p, x);
H = PreComputeHashes(T, |P|,p,x);
for i from 0 to |T| - |P| do
 | if pHash != H[i] then
 | | continue;
 | end
 | if AreEqual(T[i..i+|P|-1],P ) then
 | | positions.Append(i);
 | end
end
return positions;
```

Algorithm 43: positions of pattern in text

زمان اجرای الگوریتم بالا از مرتبه ی زمانی O(|T|+(q+1)*|P|) است که چون معمولا q کوچک است پس مرتبه ی آن خیلی کم تر از O(|T|*|P|)
[؟] [؟] [؟] [؟]

[۶] pseudocode
[۷] pseudocode

جلسه ۲۳

Binary Search Trees

فاطمه امیدی - ۱۳۹۸/۹/۱۶

۱.۲۳ what is Binary Search Tree

نوعی ساختمان داده است که بصورت درخت است؛ باینری است، یعنی هر node حداکثر دو بچه دارد، و هر node از تمام node های سمت چپش بزرگتر و از تمام node های سمت راستش کوچک است.
این ساختمان داده برای بعضی کاربرد ها، مثل پیدا کردن داده های بین دو داده خاص بسیار مناسب است.

۲.۲۳ Operations

۱.۲.۲۳ RangeSearch

```
private static List<Node> RangeSearch(long x, long y, node r)
{
    List<Node> l = new List<Node>();
    Node n = Find(x,r);
    while(n.key <= y)
    {
        if(n.key >= x)
            l.append(n);
        n = Next(n);
    }
    return l;
}
```

۲.۲.۲۳ Find

```
private static Node Finde(long k, node r)
{
    if(r.Key == k)
        return R;
    else  if(r.Key > k)
    {
        if(r.Left != null)
            return Find(k,r.Left);
        return R;
    }
    else
        return Find(k,r.Right);
}
```

۳.۲.۲۳ Next

```
private static Node Next(node n)
{
    if(n.Right != null)
        return LeftDescendant(n.Right);
    else
        return RightAncestor(n);
}
```

```
private static Node LeftDescendant(node n)
{
    if(n.Left is null)
        return n;
    else
        return LeftDescendant(n.left);
}
```

```
۱    private static Node RightAncestor(node n)
۲    {
۳        if(n.Key < n.Parent.Key)
۴            return n.Parent;
۵        else
۶            return RightAncestor(n.parent);
۷    }
```

۴.۲.۲۳ Insert

```
۱    private static void Insert(long k, node r)
۲    {
۳        var p = Find(k,r);
۴        Add new node with key k as child of p
۵    }
```

۵.۲.۲۳ Delete

```
۱    private static void Delete(node n)
۲    {
۳        if(n.Right is null )
۴            Remove n, promote n.left;
۵        else
۶        {
۷            var x = Next(n);
۸            Replace n by x , promte x.Right;
۹        }
۱۰   }
```

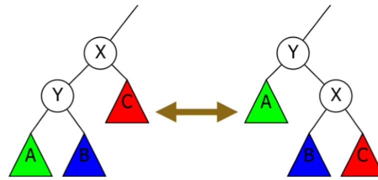
۳.۲۳ Order

پیچیدگی محاسباتی تمام عملگر های بالا به اندازه ارتفاع درخت است.

در نتیجه باید درخت را جوری تعریف کنیم تا درخت Balanced باشد و اندازه سمت چپ و سمت راست برابر باشد تا ارتفاع log n باشد و پیچیدگی محاسباتی بهترین مقدار شود. برای این کار میتوانیم از دو متد AVL tree و Splay tree استفاده کنیم.

۴.۲۳ AVL Tree

با استفاده از Left Rotation Right Rotation سعی میکنیم درخت را balanced نگه داریم.



$A < Y < B < X < C$

۵.۲۳ Splay Tree

با این فرض که احتمال آنکه یک node که آخرین بار فراخوانی شده را باز بخواهیم فراخوانی کنیم بیشتر است، بعد از هربار find ، آن node را به ریشه نزدیک تر میکنیم.
and a good wetsite [؟].

جلسه ۲۴

درخت AVL

هژار آزیز - ۱۳۹۸/۹/۱۸

جزوه جلسه ۲۴ام مورخ ۱۳۹۸/۹/۱۸ درس ساختمان‌های داده تهیه شده توسط هژار آزیز.

۱.۲۴ مروری بر مباحث جلسه گذشته

در جلسه گذشته با مفهوم درخت‌های دودویی و چگونگی کارکرد آن‌ها آشنا شدیم. همچنین یاد گرفتیم که چگونه تابع‌های Find، Next، Insert Search و Delete را پیاده‌سازی کنیم. با وجود اینکه درخت‌های دودویی یکی از بهینه‌ترین ساختمان‌های داده هستند، اگر به همان شیوه‌ی ساده و ابتدایی خود پیاده‌سازی شوند، با مشکلاتی روبه‌رو خواهند شد. یکی از این مشکلات، به‌هم‌خوردن تعادل درخت است که در این جلسه به بحث درمورد و راه‌حل‌های آن می‌پردازیم.

۲.۲۴ مقدمه‌ای بر درخت AVL

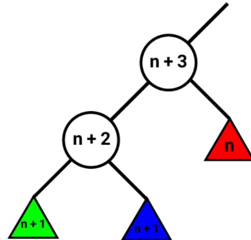
درخت AVL نام خود را از اول نام دو مخترع خود به نام‌های Adelson−Velsky و Landis گرفته است. این درخت در علوم کامپیوتر یک درخت خودمتوازن‌کننده است که اولین نوع از این ساختمان داده است.

۳.۲۴ درخت AVL و پیاده‌سازی آن

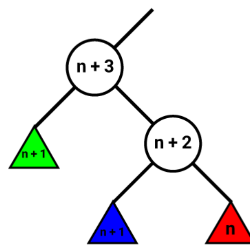
درخت AVL مزیت‌های زیادی دارد که یکی از آن‌ها کاهش ارتفاع درخت برای کاهش پیچیدگی محاسباتی عملیات‌های انجام شده بر روی درخت دودویی است. شرط اصلی برقراری خاصیت درخت AVL این است که تفاوت ارتفاع زیردرخت‌های چپ و راست هر گره در درخت حداکثر برابر یک باشد. اگر این شرط برقرار نباشد، درخت دارای خاصیت AVL نخواهد بود. این خاصیت در نهایت باعث خواهد شد که پیچیدگی محاسباتی همه‌ی عملیات‌های که در درخت دودویی AVL انجام می‌شود از $O(Log(n))$ خواهد بود.

۱.۳.۲۴ درخت AVL با مثال

به درخت‌های دودویی دو شکل زیر نگاه کنید. درخت شکل ۱.۲۴ از قاعده‌ی AVL پیروی نمی‌کند. درخت شکل ۲.۲۴ صحیح شده‌ی درخت اولی است و از چرخش درخت شکل ۱.۲۴ به سمت راست به‌دست آمده‌است.



شکل ۱.۲۴: یک نمونه از درخت دودویی که قاعده‌ی AVL در آن رعایت نشده است.



شکل ۲.۲۴: درخت فوق از چرخش درخت شکل ۱.۲۴ به سمت راست بدست آمده است.

۲.۳.۲۴ پیاده‌سازی و شبه‌کد درخت AVL

همان‌طور که می‌دانید در هربار اضافه‌کردن یک گره جدید به درخت دودویی، درخت تغییر می‌کند. بنابراین هربار با اضافه‌کردن یک گره جدید باید درخت را دوباره متعادل(Rebalance) کنیم.

```
initialization;
AVLInsert(k, R)
Insert (k, R);
N = Find (k, R);
Rebalance (N);
return;
```

Algorithm 44: AVL Tree Insertion

حالا که مرحله اضافه کردن گره را پیاده‌سازی کردیم، به سراغ حذف یک گره از درخت می‌رویم. در این مرحله نیز درخت تغییر می‌کند و باید دوباره درخت را متعادل کنیم تا خاصیت AVL برقرار باشد.

```
initialization;
AVLDelete(N)
Delete (N);
M = Parent of node replacing N;
Rebalance (M);
return;
```

Algorithm 45: AVL Tree Deletion

برای پیاده‌سازی دقیق‌تر شبه‌کد بالا، به اسلایدهای اصلی درس مراجعه کنید.

جلسه ۲۷

پیمایش در گراف

ملیکا احمدی رنجبر و رضا علیدوست - ۱۳۹۸/۹/۳۰

۱.۲۷ نمایش گراف و انواع گراف

هرگراف شامل دو عنصر گره (node) و یال (edge) است . سه نوع نمایش گراف موجود است :

- لیست یال list of edge شامل تمام یال های بین دو گره
- ماتریس مجاورت adjacency matrix ماتریسی از ۰ و ۱ که اگر یال بین دو گره باشد ۱ و اگر نه ۰ میگذاریم
- لیست مجاورت adjacency list لیست گره های متصل به هر گره عملیات های متفاوت سرعت های متفاوتی در هر نوع نمایش دارند و این بستگی به نوع گراف دارد. دو نوع گراف موجود است :
  - dense graph تعداد یال های زیادی دارد $|E|$ هم ارز $|V|^2$
    sparse graph تعداد یال های کمی دارد $|E|$ هم ارز $|V|$

۲.۲۷ پیمایش گراف

پیمایش گراف در حالت کلی به معنی گذشتن و دیدن تمام راس‌های گراف است و معمولا پیمایش از طریق یال‌های موجود درگراف انجام می‌شود. معمولا پیمایش گراف به تنهایی ارزش خاصی ندارد و هدف از پیمایش، محاسبه یا پیدا کردن چیز خاصی است.

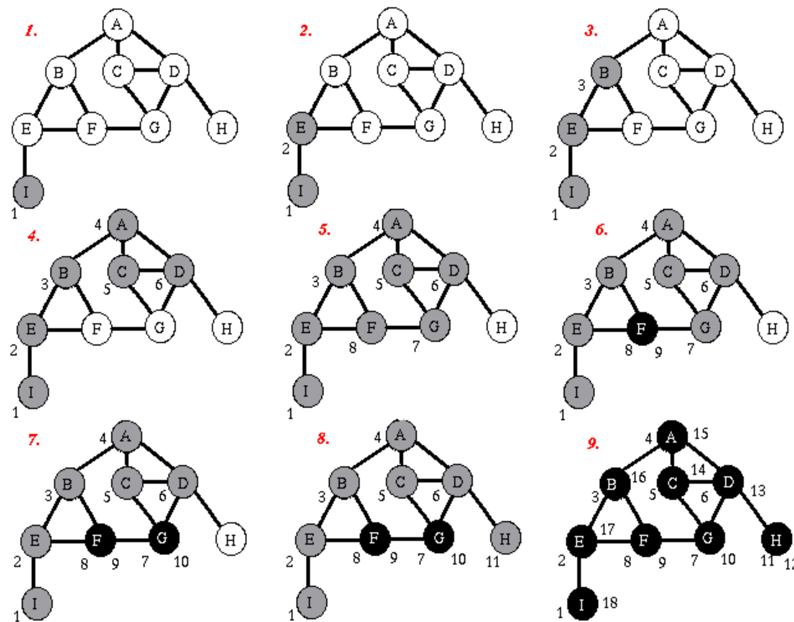
۱.۲.۲۷ Explore

با Explore کردن گره v درگراف G همه گره‌هایی که گره v به آن دسترسی دارند را پیدا میکنیم. به عبارتی همه‌ی گره‌هایی که با گره v دریک مولفه همبندی هستند را بدست می‌آوریم. الگوریتم به این شکل است که ابتدا یک راس مانندv انتخاب میکنیم و آن را ریشه می‌نامیم. ریشه را علامت‌گذاری می‌کنیم. سپس یک راس دلخواه علامت نخورده مجاور با v را انتخاب می‌کنیم و آن را u می‌نامیم. u را یکی از بچه‌های v می‌کنیم، سپس u را علامت

می‌زنیم. حال همین الگوریتم را روی u از ابتدا اجرا میکنیم. الگوریتم گفنه شده زمانی به بن‌بست می‌خورد که به ازای راسی مانند u ، تمام همسایه‌هایش علامت خورده باشند. در این صورت به راس پدر برمی‌گردیم و دوباره همین الگوریتم را از ادامه اجرا می‌کنیم. برنامه زمانی متوقف می‌شود که به ریشه برگشته باشیم و تمام همسایه هایش علامت خورده باشند که در این صورت می‌گوییم الگوریتم پایان یافته است. شبه‌کد ۴۷ آن به صورت زیر است :

```
Result: Explore(V)
visited(V) ← true
for (V, W) ∈ E do
    if not visited(W) then
        Explore(W);
    end
end
```

Algorithm 46: Explore(V)



شکل ۱.۲۷: Explore(V)

۲.۲.۲۷ جست‌وجوی اول عمق (DFS)

جست‌وجوی عمق اول که به DFS(DepthFirstSearch) معروف است در واقع الگوریتمی برای پیمایش کل گراف است. اگر گراف همبند نباشد Explore(V) تنها راس‌های مولفه همبندی ریشه را پیمایش می‌کند پس برای پیمایش روی تمام راس‌ها باید Explore به ازای هر راس علامت نخورده تکرار شود.شبه‌کد ۴۸ آن به صورت

```
Result: DFS(V)
for allv ∈ V do
 | visited[v] ← false
end
for v ∈ V do
 | if not visited[v] then
 |  | Explore(v)
 | end
end
```

Algorithm 47: DFS(V)

زیر است:

جست‌جوی اول عمق یال‌هایی که تشکیل دور می‌دهند را نمی‌رود در نتیجه اگر یال‌های رفته شده را کنار هم بگذاریم، تشکیل یک درخت ریشه دار می‌دهند که به آن درخت DFS می‌گویند.

۳.۲.۲۷ Previsit and Postvisit Orders

زمان ورود و خروج راس نیز ویژگی‌های منحصر به فردی دارد که این‌گونه تفریف میگردند :

- Previsit Order : زمانی که برای اولین بار وارد یک راس می‌شویم و آن را علامت‌گذاری میکنیم.
- Postvisit Order : زمانی که برای آخرین بار راس را می‌بینیم و از آن خارج می‌شویم و تمام همسایه‌هایش دیده شده است و در حال بازگشت به راس پدر هستیم.

اگر در شکل ۲۷.۱ بخواهیم با رنگ‌ها این دو زمان را معادل کنیم، زمان خاکسنری شدن برابر زمان ورود و زمان سیاه شدن برابر زمان خروج است. خاصیت خوبی که این تعاریف می‌دهد این است که ما می‌توانیم فرض کنیم هنگامی که از یک راس خارج می‌شویم، کار تمام زیردرخت آن تمام شده است.

۳.۲۷ مولفه های همبند

می خواهیم ببینیم کدام گره ها در گراف G از بقیه گره ها قابل دسترسی است.
زمانی دو گره از یکدیگر قابل دسترسی اند اگر و تنها اگر هر دو در یک مولفه همبند باشند. قابل دسترسی بودن یک رابطه هم ارزی است :

۱. هر گره به خودش دسترسی دارد

۲. اگر گره u به گرهv قابل دسترسی باشد ، پس گره v هم به گره u دسترسی دارد.

```
Result: DFS(G)
for (v) ∈ Vmarkvunvisited do
    cc ← 1
    for (v) ∈ V do
        if not visited(v) then
            Explore(W)
            cc ← cc + 1
        end
    end
end
```

۴.۲۷ گراف جهتدار

گراف جهت دار در این گراف هر یال جهت دارد و دارای راس شروع و پایان می باشد. پیمایش DFS در گراف جهت دار : برای این کار فقط اجازه حرکت در جهت یال ها را داریم ، سپس با صدا زدن تابع Explore(v) ، تمام راس های قابل دسترس از راس v را می یابیم .
چرخه در گراف جهت دار اگر از یکی از راس های گراف شروع کنیم و به ترتیب یال ها را بپیماییم به طوریکه راس شروع یال بعدی راس پایانی یال فعلی باشد و به همان راس ابتدایی برسیم میگوییم که در گراف چرخه وجود دارد و هیچ ترتیب خطی از آن وجود ندارد.
dag (directed acyclic graph) : به گراف جهت داری میگوییند که چرخه در آن وجود ندارد و میتوان ترتیب خطی از آن به دست آورد.

جلسه ۲۹

Topological Sort - SCCs

سید مصطفی مسعودی - یاسین عسکریان - ۱۳۹۸/۱۰/۰۷

۱.۲۹ مرتب سازی موضعی (Topological Sort)

مرتب سازی توپولوژیک، مرتب سازی رئوس یک گراف جهت دار بدون طوقه [۱] ý Insert

[۱] طوقه : گره ای که یالی از خود از خودش به خودش وجود داشته باشد

و بدون دور (DAG) است به طوری که هر راس قبل از رئوسی میاید که به آنها یال خروجی داده است.

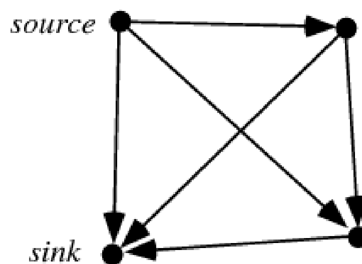
کاربرد اصلی مرتب‌سازی موضعی در زمانبندی یک سلسله‌ای ازکارها یا وظایف است. برای مثال در زمان شستن لباس‌ها، قبل از شروع خشک شدن لباس‌ها، کار شستن باید تمام شود

۱.۱.۲۹ Source

گره ای است که ، هیچ یالی به آن وارد نشده است .

۲.۱.۲۹ Sink

گره ای است که ، هیچ یالی از آن خارج نشده است .



شکل ۱.۲۹: Source and Sink

۳.۱.۲۹ الگوریتم پایه

- بر روی گراف DFS زده تا گره Sink پیدا شود
- آن را به انتهای لیست اضافه کرده
- آن را از گراف حذف کنید
- دوباره این مراحل را تا تمام شدن گره ها تکرار کنید

```
Data: Graph g
Result: void
while g is non-empty do
    Follow a path until cannot extend
    Find sink v
    put v at end of order
    Remove v form g
end
```

Algorithm 48: LinearOrder

اردر الگوریتم پایه

- هر بار اجرای الگوریتم DFS با اردر تعداد گره ها طول میکشد O(V)
- و برای هر گره ،مرحله ی قبل را باید اجرا کرد،.یعنی به تعداد V بار تکرار می کنیم
- پس اردر کلی الگوریتم O(V*V) می شود

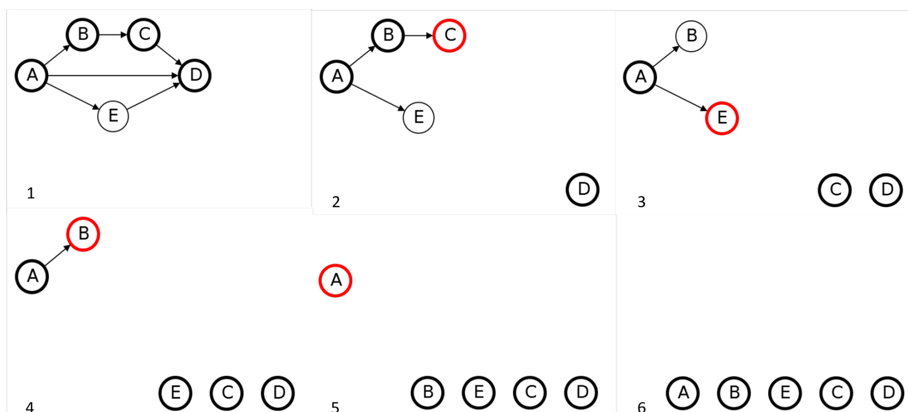
۴.۱.۲۹ الگوریتم سریعتر

در این حالت ، تنها یک بار DFS را اجرا کنید و بر اساس شماره ی PostVisit به ترتیب از کوچک به بزرگ به انتهای لیست اضافه کنید. در این حالت فقط یک بار الگوریتم پیمایش عمق اول اجرا خواهد شد و اردر برنامه بهتر خواهد شد.

```
Data: Graph g
Result: void
DFS(g)
sort vertices by reverse post-order
```

Algorithm 49: TopologicalSort

مراحل اجرای الگوریتم مرتب سازی موضعی،به صورت زیر است :



شکل ۲.۲۹: مرتب سازی موضعی

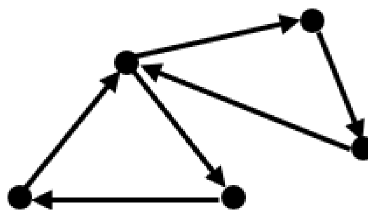
۲.۲۹ گراف قویا همبند

۱.۲.۲۹ جفت راس قویا همبند

در گراف جهت‌دار دو راس u و v قویا همبند هستند، اگر مسیری از u به v و مسیری از v به u وجود داشته باشد.

۲.۲.۲۹ گراف قویا همبند

یک گراف جهت دار قویا همبند است اگر هر دو راس آن قویا همبند باشند. در تصویر می‌توانید یک گراف قویا همبند را مشاهده کنید.



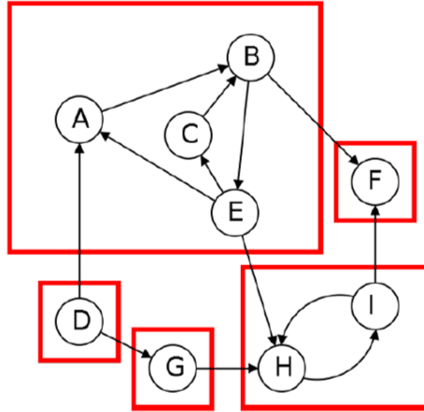
شکل ۳.۲۹: گراف قویا همبند

۳.۲.۲۹ مؤلفه‌ی قویا همبند Insert[۲] ý

Strongly Connected Component (SCC)[۲]

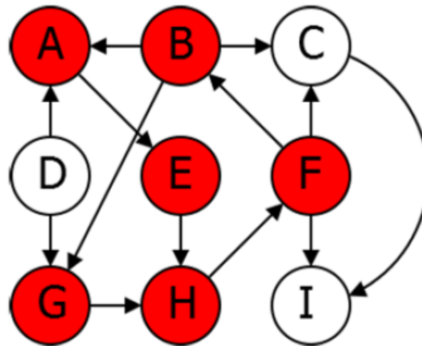
اگر یک زیر مجموعه‌ای از رئوس گراف جهت‌دار G همراه با تمام یال‌های بین آنها(یک زیر گراف) که خاصیت قویا همبندی بین هر دو راس آن وجود دارد، قابل گسترش نباشد، یک مؤلفه‌ی قویا همبند در گراف جهت‌دار G است. قابل گسترش نبودن به این معنی که نتوان هیچ راسی به این زیرمجموعه اضافه کرد که همچنان این زیرمجموعه خاصیت قویا همبندی خود را حفظ کند. رئوس هر گراف جهت‌داری را می‌توان به تعدادی مؤلفه‌ی

قویا همبند افراز کرد.



شکل ۴.۲۹: افراز یک گراف به زیر گراف های قویا همبند

در شکل زیر،گره هایی که با گره A در یک زیر گراف قویا همبند هستند ،با رنگ قرمز نشان داده شده است.



شکل ۵.۲۹: یک زیر گراف قویا همبند

۴.۲.۲۹ گراف معکوس

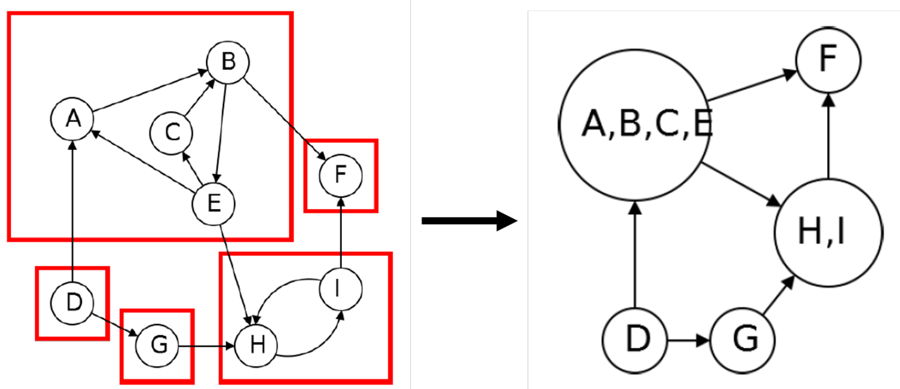
گراف معکوس، گرافی است که برای گراف‌های جهت‌دار تعریف می‌شود.به عبارت ساده‌تر گراف معکوس G ،همان گراف G است که جهت یال‌هایش عکس شده.

ویژگی قویا همبندی بین هر دو جفت راس بعد از معکوس کردن حفظ می‌شود. پس ویژگی قویاهمبندی گراف در گراف معکوس نیز حفظ می‌شود.

به صورت کلی‌تر مولفه‌های قویا همبند گراف و گراف معکوس یکیست.

۵.۲.۲۹ Metagraph

برای ساختن Metagraph ،باید مؤلفه های قویا همبند یک را پیدا کرد و روابط بین آن ها را درنظر گرفت. به شکل زیر دقت کنید.



شکل ۶.۲۹: Metagraph

و می توان گفت،همیشه Metagraph مربوط به یک گراف ، یک گراف جهت دار بدون دور (DAG) است.

۶.۲.۲۹ الگوریتم ساده

```
Data: Graph g
Result: SCCs
for each vertex v do
 | run explore(v) to determine vertices reachable from v
end
for each vertex v do
 | find the v reachable from v that can also reach v
end
return SCCs
```
Algorithm 50: EasySCC

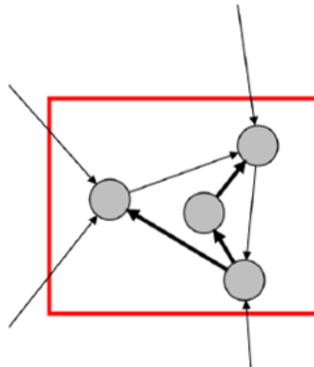
به دلیل بالا بودن پیچیدگی زمانی این الگوریتم به یک الگوریتم سریعتر نیاز خواهیم داشت.

Sink Component

برای پیدا کردن مؤلفه های قویا همبند در گراف، می توان هر بار مؤلفه ی Sink را پیدا کرد، و آن را از گراف جدا کرده و مجددا به دنبال مؤلفه ی Sink بعدی گشته و این کار را تکرار می کنیم تا تمام مؤلفه ها بدست آیند.

۷.۲.۲۹ الگوریتم پایه

الگوریتم DFS را روی گراف اجرا کرده و PreVisit و PostVisit را برای گره ها می نویسیم.آن گره ای که بزرگترین شماره یPostvisit را دارد، قطعا در مؤلفه ی Source قرار دارد. پس برای بدست آوردن مؤلفه



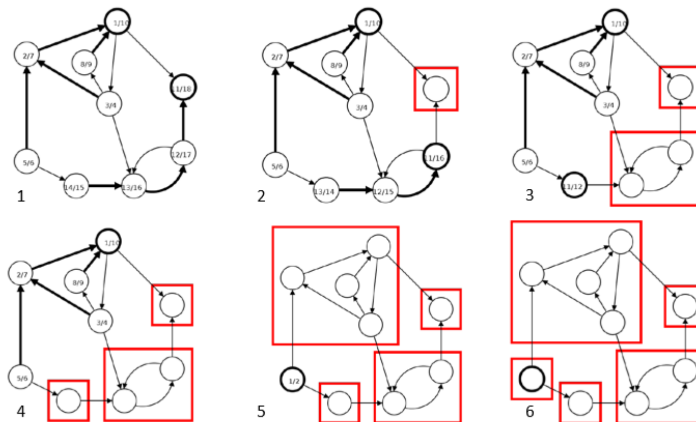
شکل ۷.۲۹: Sink Component

های Sink ، باید مؤلفه های Source در گراف معکوس را بدست آورد.

Data: Graph g
Result: SCCs
run DFS(g^R)
let v have largest post number
vertices found are first SCC
Remove from g and repeat

Algorithm 51: EasySCC

می توانید پیمایش یک گراف برای پیدا کردن مؤلفه های قویا همبند از طریق الگوریتم ساده را در شکل زیر مشاهده کنید



شکل ۸.۲۹: پیمایش گراف از طریق الگوریتم پایه

۸.۲.۲۹ الگوریتم سریع

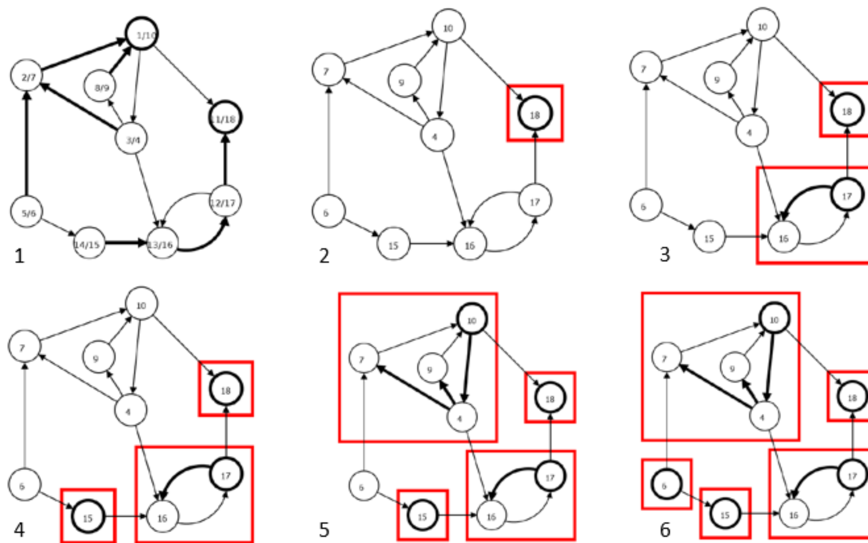
همانطور که مشاهده کردید در الگوریتم پایه پس از هر بار اجرای DFS و پیدا کردن مؤلفه های همبند،گره های آن مؤلفه حذف شده و دوباره برای پیدا کردن بزرگترین شماره ی PostVisit گراف جدید تمام آن مراحل از اول اجرا شده است اما در الگریتم سریع تنها با یک بار اجرای DFS و یادداشت کردن کردن شماره ی PostVisit های گراف معکوس،جستجو را برای پیدا کردن مؤلفه های قویا همبند از بزرگ ترین شماره ی PostVisit، از گراف معکوس،در گراف اصلی شروع کرده و پس از پیدا کردن هر مؤلفه قویا همبند گره های مشاهده شده را یادداشت کرده و دوباره از بزرگ ترین شماره ی PostVisit مشاهده نشده جستجو را آغاز کنید

```
Data: Graph g
Result: SCC
Run DFS(g^R)
for v in reverse postorder do
    if not visited(v) then
        Explore(v)
        masrk visited vertices as new SCC
    end
end
```

Algorithm 52: SCCs

می توانید پیمایش یک گراف برای پیدا کردن مؤلفه های قویا همبند از طریق الگوریتم سریع را در شکل زیر مشاهده کنید

زمان اجرای الگوریتم سریع O(|V|+|E|) می باشد



شکل ۹.۲۹: پیمایش گراف از طریق الگوریتم سریع

جلسه ۳۰

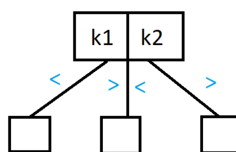
درخت ۲-۳ و درخت قرمز-سیاه

زهرا حسینی - ۱۳۹۸/۱۰/۹

جزوه جلسه ۳۰ام مورخ ۱۳۹۸/۱۰/۹ درس ساختمان‌های داده تهیه شده توسط زهرا حسینی. در این جلسه به بررسی درخت های قرمز-سیاه و ۲-۳ پرداختیم، ابتدا از درخت ۲-۳ آغاز میکنیم.

۱.۳۰ درخت ۲-۳

میدانیم در درخت باینری هر گره دو فرزند دارد،حال در درخت ۲-۳ همانطور که از اسمش هم میتوان حدس زد هر گره دو یا سه فرزند دارد.باید توجه داشت که این فرزند ها با ترتیب خاصی قرار میگرند اگر در هر گره پدر دو key قرار بگیرد ترتیب قرارگرفتن فرزند ها مانند شکل زیر است:



شکل ۱.۳۰: درخت ۲- ۳

هدف از این ساختمان داده بهبود worst-case های BST است که در جدول زیر مشاهده میکنید از مقدار N به logN است. ۲.۳۰

درخت ۲-۳ سه نوع گره دارد:

- گره برگ که فقط یک مقدار دارد.
- گره۲
- گره۳

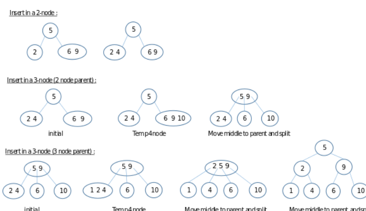
implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	compareTo()
goal	log N	log N	log N	log N	log N	log N	yes	compareTo()

شکل ۲.۳۰: پیچیدگی زمانی BST

ایده کلی به اینصورت است که هر گره ایی که یک مقدار دارد میتواند تا ۲ فرزند داشته باشد و همچنین گره ایی که ۲ مقدار داشته باشد میتواند تا۳ فرزند داشته باشد.اگر این گره ها در پایین ترین جایگاه درخت بودند به همان تغداد گفته شده دارای فرزند های null هستند. از نکاتی که باید به آن توجه کرد این است که درخت ۲-۳ همواره متوازن است و تعداد یال ها برای رسیدن به برگ یکسان است. حال به بررسی عملیات این نوع درخت میپردازیم

۱.۱.۳۰ Insert

مقدار مورد نظر را به گره سه تایی می افزاییم سپس مقدار وسطی را به پدر کرده و مقادیر دیگر را فرزند آن پدر میکنیم به اشکال زیر دقت کنید
۴۶ استفاده کنید :



شکل ۳.۳۰: Insertion

۲.۱.۳۰ Search

در این بخش در هر گره با توجه به حالاتی مه ممکن پیش بیایید و در شکل ۱.۳۰ نشان داده شد پیش میرویم تا به مقدار کلید داده شده برسیم. بعد از مطالبی که بالا گقته شد میتوان جدول زیر را تشکیل داد

۲.۳۰ Red-Black Tree

این ساختمان داده بر پایه درخت قسمت قبلی است که مورد بحث قرار گرفت،یک نوع درخت جستجوی دودویی خود-متوازن است .. این ساختمان داده پیچیده است با این حال اعمال مربوط به آن حتی در بدترین حالت نیز

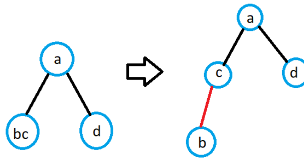
implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	compareTo()
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	compareTo()

constants depend upon implementation

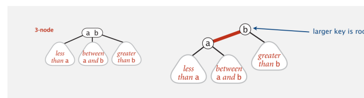
شکل ۴.۳۰: پیچیدگی زمانی

زمان اجرای خوبی دارند، در واقع زمان جستجو، حذف، و درج برای این درخت مانند درخت AVL لگاریتمی است $O(\log(n))$ که n تعداد گره‌های موجود در درخت است). مزیت عمده این ساختمان داده نسبت به درخت AVL این است که اعمال درج و حذف با تنها یک بار پیمایش درخت از بالا به پایین و تغییر رنگ گره‌ها انجام می‌شوند و در نتیجه پیاده‌سازی این درخت از درخت AVL ساده‌تر است.

این نوع درخت برای رهایی از مشکلات و پیچیدگی های درها ۲-۳ استفاده میشود تبدیل این دو درخت با توجه به شکل زیر قابل انجام است(الف) ۵.۳۰ برای درک بهتر شکل زیر را مشاهده کنید: (ب) ۶.۳۰



شکل ۵.۳۰: (الف)تبدیل درخت ۲-۳ به درخت قرمز-سیاه



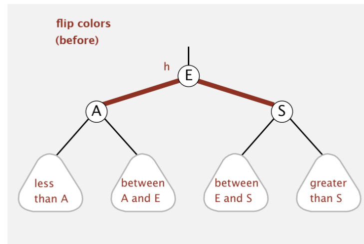
شکل ۶.۳۰: (ب)تبدیل درخت ۲-۳ به درخت قرمز-سیاه

حال به بررسی قواعد کلی حاکم بر این درخت و درخت Left-leaning red-black یا به اختصار LLRB میپردازیم:

- هیچ گره ایی دو یال قرمز ندارد
- تعداد یال های سیاه بین ریشه و برگ یک اندازه هستند
- یال های قرمز در سمت چپ قرار دارند
-

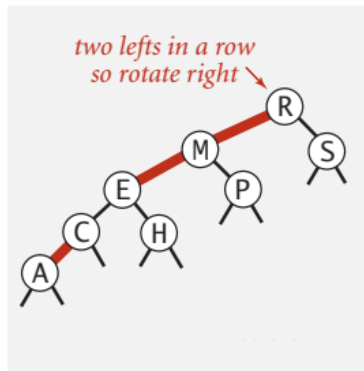
اگر هر یک از موارد بالا نقض شود باید به رفع آن بپردازیم که غالبا با عملیات rotation این مشکلات حل میشوند
در رابطه با مورد اول دو حالت پیش می آید:

- هر دو فرزند یک گره قرمز باشند.



شکل ۷.۳۰: دو فرزند قرمز

- دو یال قرمز در یک گره به طور متوالی مشترک باشند .

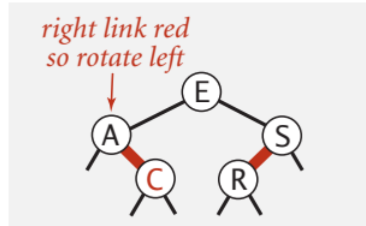


شکل ۸.۳۰: دو یال قرمز متوالی

در مورد اول همانطور که در شکل نیز گفته شده است رنگ یال ها را عوض میکنیم. و دوباره قواعد گفته شده را برای درخت چک میکنیم. در مورد دوم نیز با توجه به اینکه قانون سوم رعایت شده باشد به سمت راست میچرخانیم و سپس مجدد قواعد را بررسی میکنیم.
قانون سومی که مطرح شد سمت چپ بودت تمام یال های قرمز است یعنی اگر یالی مانند شکل زیر داشتیم باید با چرخش به چپ این مشکل را حل کنیم و مجدد صحت قواعد را بررسی کنیم ۹.۳۰
در نهایت حدول بدسا آمده به این شکل تکمیل میشود: ۱۰.۳۰ از تکاتی مه باید به آن توجه داشت هر گره ایی که اضاقه میشود به این نوع درخت به رنگ قرمز است و قواعد را برای آن بررسی میکنیم.

۳.۳۰ منابع بیش تر

- Red–black tree https://en.wikipedia.org/wiki/Red%E2%80%93black_tree



شکل ۹.۳۰: یال قرمز سمت راست

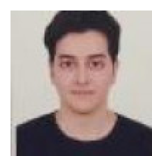
implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	compareTo()
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	compareTo()
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N *	1.00 lg N *	1.00 lg N *	yes	compareTo()

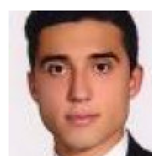
* exact value of coefficient unknown but extremely close to 1

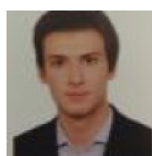
شکل ۱۰.۳۰: پیچیدگی زمانی

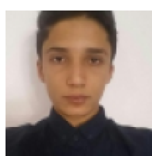
- princeton course pr https://algs4.cs.princeton.edu/33balanced/

۳۱
فهرست دانشجویان


احمد بهمني

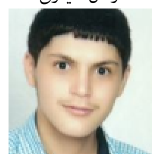

ارمان حيدری


ارمين غلام پور


اميد ميرزاجانی

امير حسين احمدی

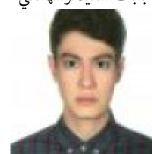
بابک سفيدگرشهانقي


حسن صبور

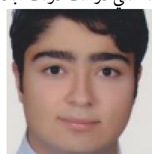
رضا علي دوست دولت ابادي

زهرا حسيني

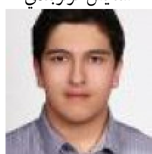
ستايش کولوبندي


سهند نظرزاده دينار

سيدمصطفي مسعودي


سپهر باباپور

شقايق مبشر

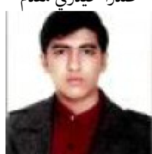

صدرا حيدري مقدم

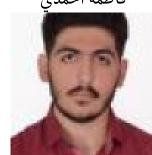
فاطمه احمدي

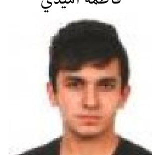
فاطمه اميدي

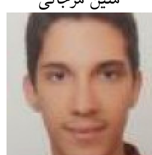
متين مرجانی

مجتبي نافذ


محمد رحمانی

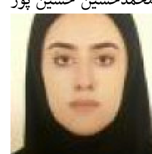

محمدحسين حسين پور


محمدحسين کريميان


محمدصدرا خاموشی فر

محمدعلی فراهت

محمدمصطفی رستم خاني

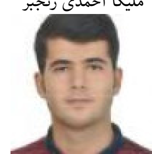

مليکا احمدي رنجبر

ميلاد اسفندياري فر

نگار زين العابدين

نگين درخشان

هادی شيخی محمدابادی


هژار ازيز

پوريا پرهيزکارغازانی

پيام صفائي پور

کيوان بوشهري

ياسمن لطف اللهی ميراشرف