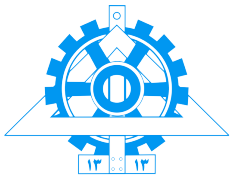


باسمه تعالی

دانشگاه تهران - دانشکده مهندسی برق و کامپیوتر

# ساختمان‌داده‌ها و الگوریتم‌ها

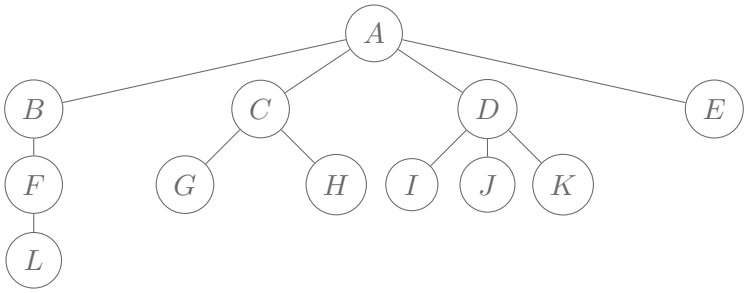
## پاسخ تمرین سوم - درخت

مجید فریدفر، ماردین نیچی

تاریخ تحویل: ۱۴۰۳/۲/۹

### ۱. درخت دودویی معادل — ۱۰ نمره

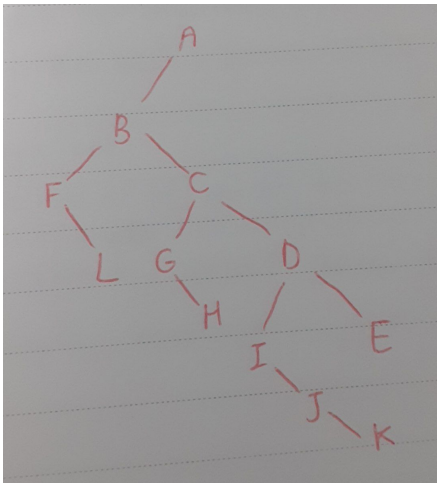
الف) درخت دودویی معادل درخت زیر را رسم کنید:




ب) الگوریتمی ارائه دهید که پدر راس n را در درخت G به کمک درخت دودویی معادل آن پیدا کند. (فرض کنید در درخت دودویی معادلی که داریم، برای هیچ نودی، لینکی به پدر واقعیش در G وجود ندارد).

**پاسخ:**

الف) برای ساخت این درخت از ریشه شروع میکنیم. چپ ترین فرزند آن را در نظر بگیرید. این راس، فرزند چپ ریشه در درخت دودویی معادل خواهد بود و sibling های آن به ترتیب، در سمت راست، نوادگان آن خواهند بود (فرزند، نوه و...). با ادامه ی این روند برای باقی نودها، به درخت زیر میرسیم:



ب) الگوریتم به این صورت است که از نود n به بالا حرکت می‌کنیم (فرض کنید نام iterator ای که استفاده می‌کنیم، p است) این کار را تا جایی که p فرزند راست است ادامه می‌دهیم. اگر به جایی رسیدیم که p فرزند چپ بود، p.parent را به عنوان جواب برمی‌گردانیم.

## ۲. پیمایش درخت ۱۵ نمره

الف) پیمایش postorder یک درخت جست و جوی دودویی (BST) داده شده است:

۵, ۶, ۱۵, ۱۰, ۲۳, ۲۴, ۲۲, ۲۶, ۲۰

پیمایش preorder آن را به دست بیاورید.

ب) با استفاده از پیمایش‌های زیر، نمایش درخت دودویی معادل را به دست بیاورید.

$$Preorder: A, B, D, E, F, C, G, H, J, L, K$$
$$Inorder: D, B, F, E, A, G, C, L, J, H, K$$

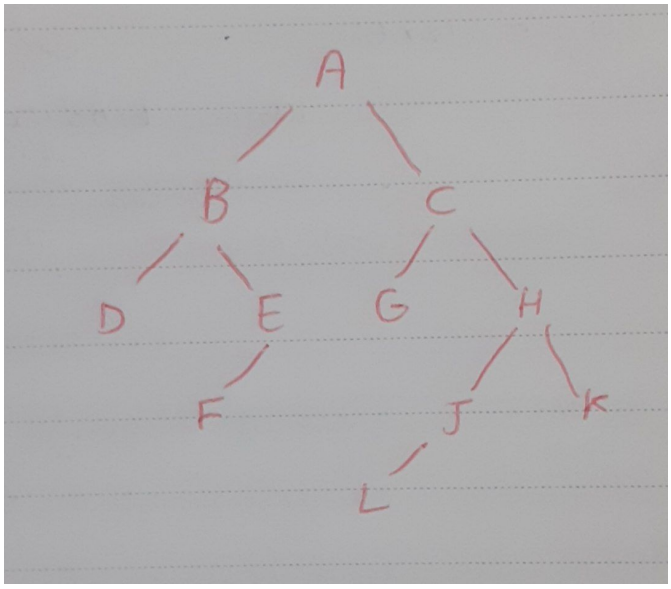
ج) آیا می‌توان نمایش یک درخت را از روی پیمایش‌های Preorder و Postorder آن به دست آورد؟ چرا؟

### پاسخ:

الف) در قدم اول متوجه میشویم که ریشه‌ی درخت راس ۲۰ است (آخرین نود در نمایش postorder)، پس ۲۰ اولین نود در نمایش pre-order است. همچنین راس های کوچک تر از ۲۰ در زیردرخت چپ (۵, ۶, ۱۵, ۱۰) و راس های بزرگ تر در زیردرخت راست (۲۳, ۲۴, ۲۲, ۲۶) قرار دارد. با ادامه ی این روند برای زیردرخت ها، به نمایش زیر می‌رسیم.

۲۰, ۱۰, ۶, ۵, ۱۵, ۲۶, ۲۲, ۲۴, ۲۳

ب) از روی نمایش preorder می‌فهمیم که ریشه نود A است و از روی نمایش inorder می‌فهمیم که نودهای D و B و F و E در سمت چپ و بقیه ی نودها در زیردرخت راست ریشه‌اند. حالا درمی‌یابیم نمایش preorder زیردرخت راست، CGHJLK و نمایش inorder آن، GCLJHK است. همچنن این نمایش ها برای زیردرخت چپ به ترتیب برابرست با: BDEF و DBFE. در نتیجه ریشه ی زیردرخت راست، C و ریشه ی زیردرخت چپ، B است (فرزندان ریشه). این روند را برای نمایش های کوچک تری که دست آوردیم ادامه می‌دهیم تا در نهایت به درخت زیر برسیم:



ج)

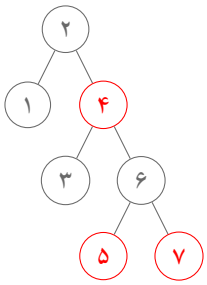
خیر نمیتوان دقیقا مشخص کرد شکل درخت ما چگونه است. مثال زیر را ببینید:

**Postorder = 3,2,1**

**Preorder = 1,2,3**


## ۳. درخت قرمز-سیاه* ۱۵ نمره

الف) راس ۸ را به درخت قرمز-سیاه زیر اضافه کنید. مراحل درج را به صورت کامل بنویسید.

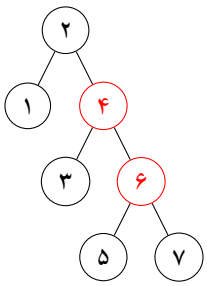



ب) اگر گره‌ای دلخواه به یک درخت قرمز-سیاه اضافه کنیم و بلافاصله پس از درج آن را حذف کنیم، آیا درخت حاصل با درخت اولیه یکسان است؟ درستی ادعای خود را اثبات کنید.

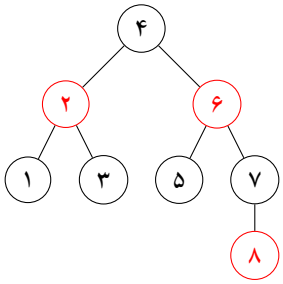
### پاسخ:

الف)

از آنجا که راس ۶، دو فرزند قرمز دارد، ۶ را قرمز کرده و فرزندانش را سیاه میکنیم. این کار باعث میشود دو راس قرمز ۴ و ۶ پشت سر هم بیفتند.

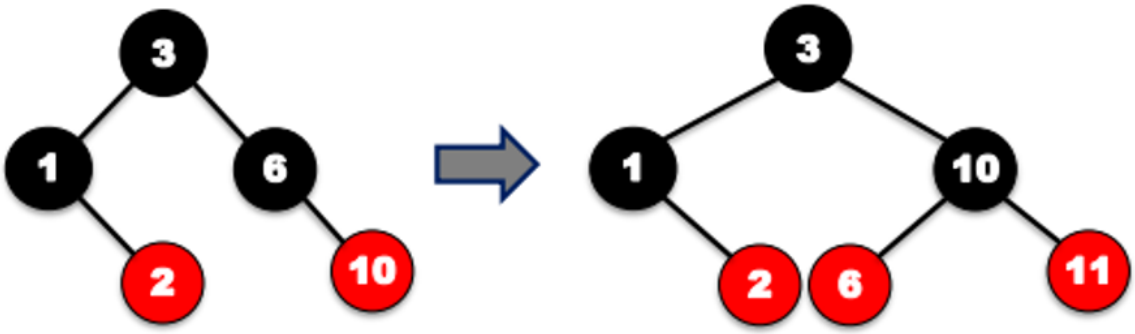


در قدم بعدی با استفاده از چرخش ها، درخت را مرتب کرده و راس ۸ را به آن اضافه میکنیم.

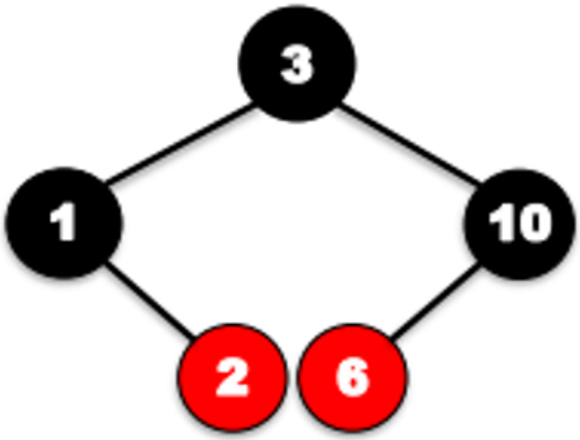



ب)

خیر میتوانند یکسان نباشند. مثلا به درخت پایین توجه کنید که به ترتیب اعداد ۲ ,۱۰ ,۶ ,۳ ,۱ به آن اضافه شده اند. حالا به آن گره ی ۱۱ را اضافه میکنیم:



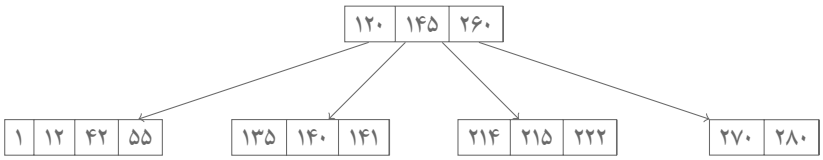

حالا بلافاصله گره ۱۱ را از آن حذف میکنیم.




همان طور که مشاهده میکنید، درخت حاصل با درخت اولیه متفاوت است.

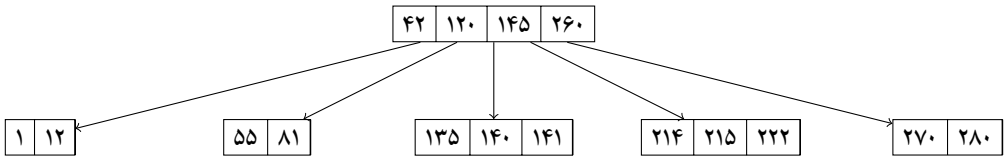
## ۴. درخت B * ۱۵ نمره

در B-Tree زیر بیشترین تعداد کلید ممکن برای هر گره را ۴ در نظر بگیرید. به ترتیب اعداد ۸۱، ۱۳۷ و ۱۲۵ را به آن اضافه کرده و شکل درخت بدست آمده در هر مرحله را رسم کنید



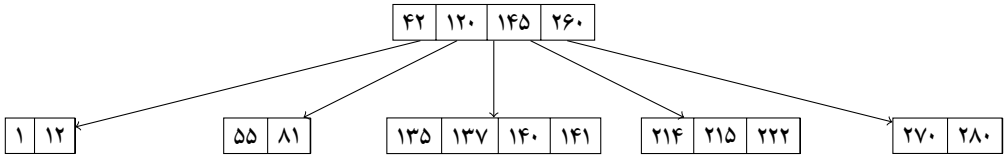

**پاسخ :**

۱۲۰ > ۸۱ پس ۸۱ باید در اولین برگ قرار بگیرد اما چون تعداد اعضای این گره از ۴ بیشتر میشود برگ میشکند و میانه آن ها یعنی ۴۲ به ریشه انتقال میابد شکل درخت بعد از اضافه کردن ۸۱:



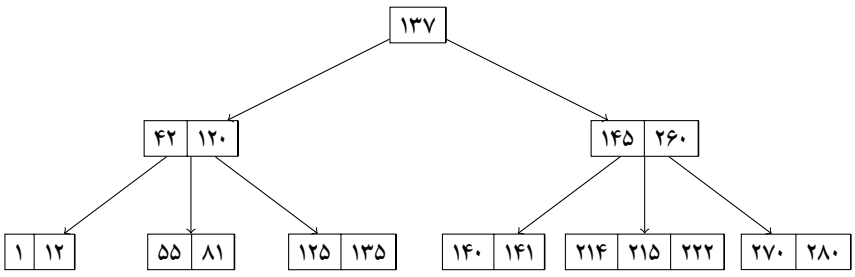

اکنون ۱۳۷ را اضافه میکنیم ۱۴۵ > ۱۳۷ > ۱۲۰ پس باید در سومین برگ قرار بگیرد

شکل درخت بعد از اضافه کردن ۱۳۷:




و حال برای اضافه کردن ۱۲۵ به درخت باید آن را در دومین برگ قرار بدهیم. ولی چون تعداد اعضای این برگ از ۵ بیشتر میشود پس این برگ به دو برگ دوتایی شکسته شده و ۱۳۷ به پدرش انتقال میابد. پس از انتقال نود ریشه دارای ۵ عضو خواهد شد پس ربشه را نیز باید شکسته و ۱۲۵ به ریشه جدید تبدیل میشود و ۸۱ و ۱۲۵ بچه چپ و ۱۴۵ و ۲۶۰ بچه راست آن خواهد شد.

شکل درخت بعد از اضافه کردن ۱۳۷:



## ۵. تصمیم حیاتی ۲۰ نمره

امروز ۱۱ سپتامبر ۲۰۰۱ ساعت ۸:۳۰ صبح است. محمد عطا، بعد از ربودن پرواز ۱۱ ام American Airlines ، در حال هدایت آن به سمت ساختمان شمالی مرکز تجارت جهانی واقع در منتهن می‌باشد. ماموریت او برخورد با برج و منفجر کردن تمام طبقات a ام تا b ام است.

فرض کنید این برج N طبقه دارد و دارای k آسانسور می‌باشد که هر کدام از آن‌ها در $n \leqslant N$ طبقه‌ی مختلف، درِ ورودی دارند (لزوما همه طبقات قابلیت دسترسی به آسانسور ندارند و ممکن است طبقه‌ای باشد که قابلیت دسترسی به چند آسانسور مختلف را داشته باشد). فرض کنید k لیست پیوندی n تایی از طبقات قابل دسترسی هر آسانسور به صورت صعودی، به عبدالعزیز العمری (دستیار محمد عطا) داده شده است. طبق دستور اسامه بن‌لادن، العمری باید کوچک‌ترین بازه‌ای از طبقات این برج را به دست بیاورد، به طوری که انفجارهای ایجاد شده، منجر به از کار افتادن تمام k آسانسور این ساختمان شود. طبق نتایج به دست آمده از تحقیقات القاعده، تنها در صورتی یک آسانسور از کار می‌افتد که در حداقل یکی از طبقاتی که به آن دسترسی دارد (به معنای وجود داشتن درِ ورودی)، انفجار مهیبی رخ دهد (با توجه به نامرغوب بودن جنس درِ آسانسور، انفجار موجب از بین رفتنِ در و پاره شدن کابل و در نتیجه از کار افتادن آن می‌شود).

الگوریتمی از مرتبه‌ی زمانی $O(nk \log k)$ ارائه دهید که با توجه به اطلاعات داده شده، العمری a و b مناسب را به محمد عطا بگوید.

### پاسخ :

ایده کلی این است که یک پوینتر بر روی هر یک از لیست های آسانسورها نگه داریم که این پوینترها عناصر درون بازه را تشکیل میدهند، با گرفتن کمینه و بیشینه این $k$ عنصر می‌توان بازه را تشکیل داد. در ابتدا، تمام پوینترها به ابتدای تمامی لینکدلیست‌ها اشاره خواهند کرد. اگر بخوایم بازه را کمینه کنیم، باید یا مقدار کمینه را افزایش دهیم یا مقدار بیشینه را کاهش دهیم. برای کاهش مقدار بیشینه، باید پوینتر بیشینه فعلی را به چپ ببریم و از آنجا که در حال حاضر در اندیس ۰ هر لیست هستیم، نمی‌توانیم پوینتر را به چپ ببریم، بنابراین نمی‌توانیم بیشینه فعلی را کاهش دهیم. بنابراین، تنها گزینه ممکن برای به دست آوردن بازه بهتر افزایش کمینه فعلی است. برای ادامه افزایش مقدار کمینه، پوینتر لیست حاوی مقدار کمینه را افزایش داده و بازه را تا زمانی که یکی از لیست‌ها تمام شود به‌روز کنید. پس بطور دقیقتر ابتدا همه $k$ عنصر اولی لیست را در یک min-heap ذخیره میکنیم در و متغیر max را برای ذخیره کردن مقدار ماکسیموم $k$ عنصر تعریف میکنیم.

در هر مرحله مقدار max - min-heap.top با مقدار جواب فعلی مقایسه و آپدیت میکنیم. سپس پوینتری که به کوچک‌ترین عنصر اشاره میکند را یکی اضافه میکنیم. تاپ هیپ را حذف میکنیم و مقدار جدید پورینتر اضافه شده را به هیپ اضافه کرده و max را نیز آپدیت میکنیم. همین کار را تا جایی ادامه میدهیم که یکی از لیست ها به اتمام برسد.

listings

## ۶. Suber ۲۵ نمره

صابر یک سرویس تاکسی اینترنتی به نام Suber تاسیس کرده است. روشی که او برای تخصیص مسافر به راننده‌ها استفاده می‌کند به این صورت است که هر درخواستی که در اپلیکیشن ثبت می‌شود، وارد صف درخواست‌ها شده، و سیستم در هر لحظه بین راننده‌هایی که در حال رانندگی نیستند بررسی می‌کند که کدام یک از آن‌ها ماشین سریع‌تری دارد. سپس اولین درخواستی که سر صف قرار دارد را به او می‌دهد. هم‌چنین اگر چند راننده صاحب سریع‌ترین ماشین بودند، درخواست سر صف به راننده‌ای می‌رسد که زودتر به Suber پیوسته باشد.

صابر برای مصاحبه‌های استخدامی، سوالی طرح کرده که هر کس بتواند پاسخ صحیحی به آن بدهد، به استخدام Suber درمی‌آید. سوال به این صورت است که دو آرایه به نام‌های drivers و commuters داریم که drivers[i] نشان‌دهنده‌ی سرعت ماشین راننده‌ی i ام (راننده‌ها به ترتیبی که به Suber پیوسته‌اند در این آرایه قرار گرفته‌اند) است و commuter[i] نشان‌دهنده‌ی مدت زمان بین مبدا تا مقصد مسافری است که درخواستش را در ثانیه‌ی i ام ثبت می‌کند (فرض کنید در هر ثانیه، یک درخواست ثبت می‌شود). مشخص کنید طبق روش مورد استفاده در Suber در نهایت هر درخواست توسط چه راننده‌ای انجام خواهد شد. نتیجه را در آرایه‌ی answers ذخیره کنید به طوری که answers[i] حاوی ایندکس راننده‌ای است که درخواست ثانیه‌ی i ام به او رسیده. هم‌چنین مرتبه‌ی زمانی راه حل خود را ارزیابی کنید.

برای سادگی، از مدت زمان رسیدن راننده تا مبدا مصافر صرف نظر کنید.

### پاسخ :

ایده به این صورت که دو تا heap در نظر می‌گیریم. یکی برای راننده‌هایی که در حال کارند (driving)، یکی برای راننده‌هایی که منتظر مسافرند (waiting). راننده‌ها در هیپ driving بر اساس زمان آزاد شدنشان مرتب می‌شوند (مجموع لحظه‌ای که مسافر سوار کرده‌اند و مدت

زمان رساندن او) و در هیپ waiting بر اساس ترکیب معکوس سرعت ماشین (برای معکوس کردن اثر آن) و ایندکسشان در آرایه‌ی drivers. هم‌چنین در نظر بگیرید که این دو هیپ min-heap هستند. در ابتدا تمام رانندگان در هیپ waiting قرار دارند.

حالا روی لیست commuters از چپ به راست حرکت میکنیم (با توجه به فرض مطرح شده در سوال، وقتی در حال بررسی مسافر i ام هستیم، در واقع در ثانیه‌ی i ام هم قرار داریم). در ابتدا، تمام رانندگانی که در هیپ driving قرار دارند، و زمان بیکار شدنشان فرا رسیده را از هیپ خارج، و وارد هیپ waiting می‌کنیم. در قدم بعدی، ابتدا به waiting نگاه میکنیم. اگر عضوی در این هیپ وجود داشت، راننده‌ی top در آن را به commuter کنونی تخصیص داده و آن را به هیپ driving منتقل می‌کنیم. در غیر این صورت، راننده‌ی top هیپ driving را به commuter کنونی تخصیص داده و زمان بیکار شدنش را به اندازه‌ی مدت زمان سفر مسافر کنونی اضافه می‌کنیم (دقت کنید در این مرحله لازم است این عنصر از هیپ حذف و دوباره درج شود. به این علت که ممکن است این افزایش سبب شود عنصر top هیپ، دیگر top نباشد).

شبه کد راه حل بالا را در تصویر زیر می‌توانید مشاهده کنید که به زبان java نوشته شده است:

```java
//O((M + N) * logN)
public int[] assignCommuters(int[] drivers, int[] commuters) {
    //since if there are multiple commuters that need to be assigned, we need to assign in  the order of index
    //so we can go through commuters from left to right
    //and ask what could be the driver for this task
    //For which driver it can be assigned, it is determined by which drivers are avaiable
    //we just need to choose from the drivers that are available with smallest speed
    //Like in priorityQueue
    //So if we have a group of drivers with their own next available time
    //if next avaibable time <= the execution time of the task, we need to choose the driver with smallest speed
    //can we have 2 PQ, one is sorted by speed and index, this is currently avaiable pq
    //one is sorted by available time, this is currently used driver pq
    //[speed, index, avaialbe_time]
    PriorityQueue<int[]> waiting = new PriorityQueue<>((a, b) -> (a[0] != b[0]) ? (a[0] - b[0]) : (a[1] - b[1]));
    PriorityQueue<int[]> driving = new PriorityQueue<>((a, b)->(a[2] != b[2]) ? (a[2] - b[2]) : ((a[0] != b[0]) ? (a[0] - b[0]) : (a[1] - b[1])));
    int n = drivers.length;
    int m = commuters.length;
    //O(nLogn)
    for (int i = 0; i < n; i++) {
        waiting.add(new int[] {drivers[i], i, 0});
    }
    int[] res = new int[m];
    //O(m * Logn)
    for (int i = 0; i < m; i++) {
        int t = commuters[i];
        while (!driving.isEmpty() && driving.peek()[2] <= i) {
            waiting.add(driving.poll());
        }
        //If there is no free drivers now, we can find the used driver with smallest available time
        if (waiting.isEmpty()) {
            int[] cur = driving.poll();
            res[i] = cur[1];
            cur[2] += t;
            driving.add(cur);
        } else {
            int[] cur = waiting.poll();
            res[i] = cur[1];
            cur[2] = i + t;
            driving.add(cur);
        }
    }
    return res;
}
```