در جلسه‌ی آخر، در مورد $set\ disjoint$ صحبت کردیم. داده‌ساختار $union - find$ را معرفی و پیاده‌سازی‌های مختلف آن را بررسی کردیم. سپس پیاده‌سازی‌های آن را بهبود بخشیدیم.

union-find: داده ساختاری است که سه عمل زیر را انجام می‌دهد:

- make-set(x) : یک مجموعه شامل عنصر x را ایجاد می‌کند.
- find(x) : مجموعه‌ای که شامل عنصر x است را برمی‌گرداند. (در حقیقت نماینده‌ی آن بازگردانده می‌شود)
- union(x،y) : مجموعه‌هایی که شامل x و y هستند را با هم ادغام می‌کند.

پیاده‌سازی‌های مختلف union-find:

- **quick find:** فرض کنید عناصر ما از ۱ تا n هستند. در این پیاده‌سازی، نیازمند به یک آرایه set هستیم که در ابتدا $set[i] = i$ باشد. شبه کد مربوط به عملیات $find(i)$ و $union(i,j)$ به صورت زیر می‌باشد:

```
int find(i) {
        return(set[i])
}
```

```
union(i,j) {
        x = set[i]
        y = set[j]
        for(k:1→n)
                if(set[k] == x)
                        set[k] = y
}
```

عمل $find(i)$ در $O(1)$ و عمل $union(i,j)$ در $O(n)$ انجام می‌شود.

Set:

1	2	3	4	5	...	n
{1}	{2}	{3}	{4}	{5}	...	{n}

Find(1) = 1

Union(4,6) →

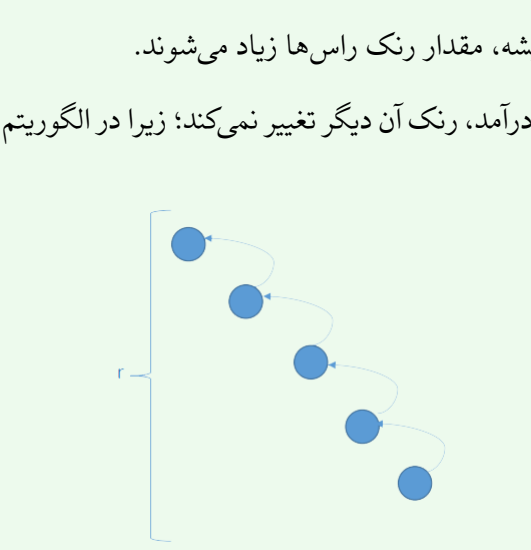
1	2	3	4	5	4	...	n

Union(3,4)→

1	2	4	4	5	4	...	n

- **quick union:** ایده این پیاده‌سازی به کمک اشاره‌گرها می‌باشد؛ بدین صورت که در ابتدا، به ازای هر راس، یک $node$ ساخته می‌شود. این $node$ ها، یک اشاره‌گر دارند که در ابتدا به صفر اشاره می‌کند.(هنگامی که یک $node$ به صفر اشاره کند یعنی ریشه است.) در این پیاده‌سازی، وقتی $find(i)$ فراخوانی می‌شود، نماینده‌ی مجموعه‌ای که i در آن قرار دارد، برگردانده می‌شود. نماینده‌ی مجموعه‌ها را راس ریشه در نظر می‌گیریم که اشاره‌گر آن به صفر اشاره می‌کند. در عمل $union(i,j)$، نماینده مجموعه‌هایی که i و j در آنها قرار دارد را پیدا کرده و بعد نماینده‌ها را یکی می‌کنیم.

در این پیاده‌سازی، $find(i)$ و $union(i,j)$ در $O(n)$ انجام می‌شوند؛ اما اگر عملیات $find(i)$ و $find(j)$ خارج از تابع صورت گیرد و به جای $union(i,j)$، عمل $union(x,y)$ را فراخوانی کنیم؛ عملیات $union$ در $O(1)$ انجام می‌شود.



قطعه کد زیر عملیات $find(i)$ و $union(i,j)$ را نشان می‌دهد:

```
int find(i) {
        if(parent[i] == 0)
                return(i)
        return find(parent[i])
}
```

```
union(i,j) {
        x = find(i)
        y = find(j)
        parent[x] = y
}
```

حال می‌خواهیم پیاده‌سازی ارائه شده را بهبود دهیم.

بهبود ۱:

هر بار ریشه درخت با ارتفاع کمتر را فرزند ریشه درخت با ارتفاع بیشتر قرار می‌دهیم. اگر مقدار رنک(ارتفاع) یک راس برابر k باشد، زیر درخت شامل آن راس حداقل ۲^k راس دارد. در نتیجه:

- تعداد راس‌های با رنک k حداکثر $\frac{n}{۲^k}$ است.
- ارتفاع درخت‌ها حداکثر $log(n)$ است پس عملیات $find(i)$ از مرتبه $O(log(n))$ خواهد شد.

رنک هیچ راسی نمی‌تواند بیشتر از $log(n)$ باشد زیرا در این صورت تعداد راس‌ها باید بیشتر از $۲^{log(n)} = n$ باشد.

بهبود ۲:

زمانی که $find$ یکی از راس‌ها اجرا می‌شود، پدر آن راس را بررسی می‌کنیم؛ اگر ریشه نبود، پدر پدر آن را بررسی می‌شود و تا رسیدن به ریشه این کار را ادامه می‌دهیم. در مسیر، پدر همه‌ی راس‌های بررسی شده را برابر راس ریشه قرار می‌دهیم. این بهبود باعث می‌شود که عملیات $find$ در زمان $O(log^*(n))$ انجام شود. در این بهبود:

- با حرکت از هر راسی به سمت ریشه، مقدار رنک راس‌ها زیاد می‌شوند.
- اگر زمانی یک راس از ریشه بودن درآمد، رنک آن دیگر تغییر نمی‌کند؛ زیرا در الگوریتم تنها رنک راس‌های ریشه تغییر می‌کند.

حال راس‌ها را بر اساس رنک در log^* در گروه مختلف قرار می‌دهیم:

[۰ ۱] [۲ ۲] [۳ ۴] [۵ ۱۶] [۱۷ ۶۵۵۳۶] [۶۵۵۳۷ $۲^{۶۵۵۳۶}$]

هزینه عملیات $find$ را به دو قسمت تقسیم می‌کنیم:

- **HB (هزینه بین‌گروهی)**: اگر پدر راس v ریشه باشد یا $parent[v]$ در گروه دیگری نسبت به v قرار داشته باشد، یک واحد به HB اضافه می‌کنیم.
- **HD (هزینه‌ی درون گروهی)**: اگر راس v و پدرش در یک گروه باشند، یک واحد به HD اضافه می‌کنیم.

بعد از n عملیات:

- به ازای هر $find$ مقدار HB حداکثر log^* است بنابراین بعد از n عملیات، HB حداکثر $nlog^*n$ خواهد بود.
- حداکثر ۲^k بار می‌تواند به HD اضافه کند. پس از n عملیات، HD حداکثر $nlog^*n$ خواهد بود.

$$HB + HD = O(nlog^*(n))$$