

به نام خدا

ساختمان داده

آرش شفيعی



## داده‌ساختارهای پیشرفته

- در برخی از الگوریتم‌ها نیاز داریم  $n$  عنصر متمایز را در تعدادی مجموعهٔ مجزا افراز کنیم به طوری که هیچ دو مجموعه‌ای عنصر مشترک نداشته باشند. چنین الگوریتم‌هایی معمولاً به دو عملگر کلیدی نیاز دارند : پیدا کردن مجموعه‌ای که یک عنصر داده شده در آن قرار دارد و محاسبه اجتماع دو مجموعه.
- در این بخش با روش‌هایی برای نگهداری داده ساختاری آشنا می‌شویم که چنین عملگرهایی را پشتیبانی می‌کند.

- یک داده‌ساختار مجموعه مجزا<sup>1</sup> یک خانواده<sup>2</sup> از مجموعه‌های پویای مجزا به صورت  $S = \{S_1, S_2, \dots, S_k\}$  است. برای مشخص کردن یک مجموعه، معمولاً یک نماینده<sup>3</sup> از هر مجموعه انتخاب می‌کنیم که عضوی از آن مجموعه است. فرقی نمی‌کند کدام یک از اعضا نماینده باشند. تنها نکته مورد اهمیت این است که اگر نماینده را دوبار برای تعیین مجموعه‌ای که متعلق به آن است فراخوانی کنیم، هر دو بار جواب یکسانی دریافت کنیم.

---

<sup>1</sup> disjoint-set data structure

<sup>2</sup> collection/family

<sup>3</sup> representative

## مجموعه‌های مجزا

مانند همه داده‌ساختارهایی که بررسی کردیم، هر عنصر در یک مجموعه مجزا، یک نمونه (متغیر یا شیء) از یک نوع (ساختمان یا کلاس) مشخص است. فرض کنید  $x$  یک متغیر باشد، در آن صورت عملگرهای زیر را در یک مجموعه مجزا خواهیم داشت.

- **Make-Set( $x$ )** : در صورتی که  $x$  متعلق به هیچ مجموعه‌ای نباشد، این تابع یک مجموعه جدید با یک عضو  $x$  به خانواده اضافه می‌کند. عنصر  $x$  نماینده مجموعه خود نیز خواهد بود.

- **Union( $x, y$ )** : دو مجموعه مجزای پویا که شامل  $x$  و  $y$  هستند، یعنی  $S_x$  و  $S_y$  را با یکدیگر الحاق می‌کند، بدین معنی که عناصر یکی از آن دو مجموعه را به دیگری انتقال می‌دهد و دو مجموعه را به یکدیگر پیوند داده و مجموعه جدیدی ایجاد می‌کند که اجتماع آن دو مجموعه است. نماینده مجموعه به وجود آمده، هر عنصری از  $S_x \cup S_y$  می‌تواند باشد، اما معمولاً یکی از نماینده‌های  $S_x$  یا  $S_y$  به عنوان نماینده انتخاب می‌شود، بنابراین این تابع  $S_x \cup S_y$  را محاسبه و به خانواده اضافه می‌کند و  $S_x$  و  $S_y$  را حذف می‌کند. البته در عمل در هنگام پیاده‌سازی عناصر یک مجموعه به دیگری منتقل می‌شوند.

- **Find-Set( $x$ )** : یک اشاره‌گر به نماینده مجموعه‌ای که شامل  $x$  است بر می‌گرداند.

زمان اجرای عملیات مجموعه‌های مجزا را بر اساس دو پارامتر تحلیل می‌کنیم :

-  $n$  تعداد عملیات Make-Set است.

-  $m$  تعداد کل عملیات Union ، Make-Set و Find-Set است.

از آنجایی که تعداد کل عملیات، عملیات Make-Set را نیز بر می‌گیرد، بنابراین  $m \geq n$  است. تعداد  $n$  عملیات اول همیشه Make-Set هستند، بنابراین بعد از  $n$  عملیات اول، خانواده از  $n$  مجموعه تشکیل شده است. هر اعمال تابع Union تعداد مجموعه‌ها را یک واحد کاهش می‌دهد. بنابراین بعد از  $n - 1$  عملیات Union تنها یک مجموعه باقی می‌ماند، پس فقط  $n - 1$  عملیات Union می‌تواند انجام شود.

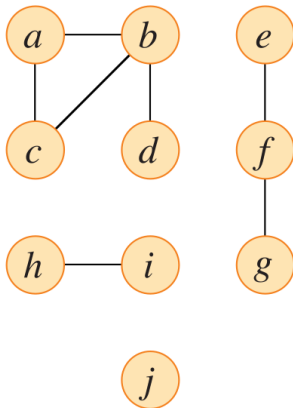
– یکی از کاربردهای داده‌ساختار مجموعه‌های مجزا پیدا کردن اجزای همبند<sup>1</sup> یک گراف بدون جهت است.

---

<sup>1</sup> connected component

# یک کاربرد مجموعه‌های مجزا

- شکل زیر یک گراف با چهار جزء همبند را نشان می‌دهد.





- الگوریتم‌های زیر از عملیات داده‌ساختار مجموعه‌های مجزا برای محاسبه اجزای همبند یک گراف استفاده می‌کند.

---

## Algorithm Connected-Components

---

```
function CONNECTED-COMPONENTS(G)
1: for each vertex  $x \in G.V$  do
2:   Make-Set( $v$ )
3: for each edge  $(u,v) \in G.E$  do
4:   if Find-Set( $u$ )  $\neq$  Find-Set( $v$ ) then
5:     Union( $u,v$ )
```

---

---

### Algorithm Same-Component

---

—     **function** SAME-COMPONENT( $u,v$ )  
1: **if** Find-Set( $u$ ) == Find-Set( $v$ ) **then**  
2:     **return** True  
3: **else**  
4:     **return** False

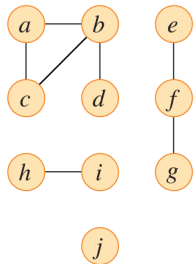
---

## یک کاربرد مجموعه‌های مجزا

- وقتی که تابع  $\text{Connected-Components}$  گراف را پردازش کرد، تابع  $\text{Same-Component}$  بررسی می‌کند که آیا دو رأس به یک جزء همبند متعلق هستند یا خیر. توجه کنید که  $G.V$  مجموعه رئوس گراف و  $G.E$  مجموعه یال‌های گراف است. تابع بررسی همبند بودن گراف، در ابتدا هر رأس  $v$  را در یک مجموعه جدید قرار می‌دهد. سپس به ازای هر یال  $(u, v)$  مجموعه‌هایی که  $u$  و  $v$  به آنها تعلق دارند را پیوند می‌دهد. پس از اینکه همه یال‌ها بررسی شدند، دو رأس به یک جزء همبند تعلق دارند اگر و تنها اگر متعلق به یک مجموعه در خانواده مجموعه‌های مجزا باشند.

# یک کاربرد مجموعه‌های مجزا

- شکل زیر نشان می‌دهد اجزای همبند توسط مجموعه‌های مجزا چگونه محاسبه می‌شوند.



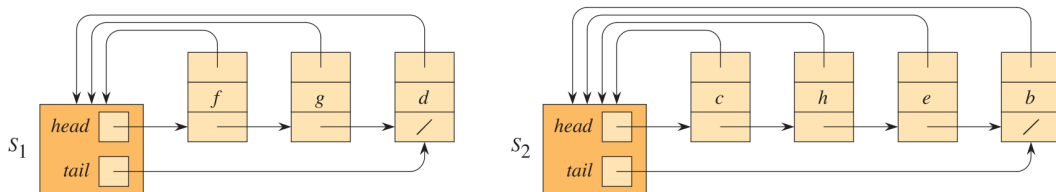
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
$(b, d)$	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
$(e, f)$	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
$(a, c)$	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
$(h, i)$	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
$(a, b)$	$\{a, b, c, d\}$				$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
$(f, g)$	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
$(b, c)$	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

## یک کاربرد مجموعه‌های مجزا

– اگر گراف ثابت باشد و در طول زمان تغییر نکند، جستجوی عمق اول اجزای همبند را سریع‌تر محاسبه می‌کند اما اگر گراف پویا باشد و یال‌ها به تدریج اضافه شوند، پیاده‌سازی آن توسط مجموعه‌های مجزا بهینه‌تر است.

# مجموعه‌های مجزا توسط لیست پیوندی

– شکل زیر یک روش ساده پیاده‌سازی داده‌ساختار مجزا را نشان می‌دهد. در این پیاده‌سازی هر مجموعه توسط یک لیست پیوندی ذخیره می‌شود.

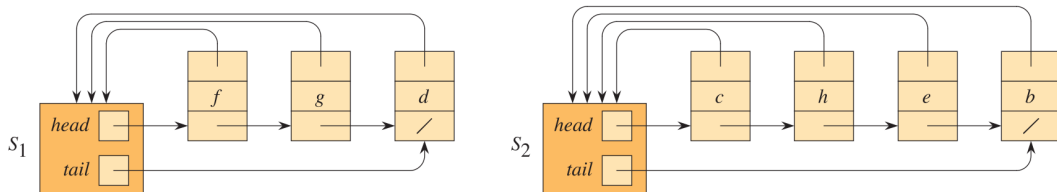


## مجموعه‌های مجزا توسط لیست پیوندی

- هر مجموعه یک ابتدا (head) دارد که اولین عضو لیست اشاره می‌کند و یک انتها (tail) دارد که به آخرین عضو لیست اشاره می‌کند. هر عنصر در لیست یک ویژگی برای نگهداری نام خود دارد و همچنین به عنصر بعد از خود اشاره می‌کند. یک عضو اشاره‌گری نیز به مجموعه خود دارد. عناصر در یک مجموعه می‌توانند هر ترتیبی داشته باشند. نماینده یک مجموعه اولین عنصر لیست است.

## مجموعه‌های مجزا توسط لیست پیوندی

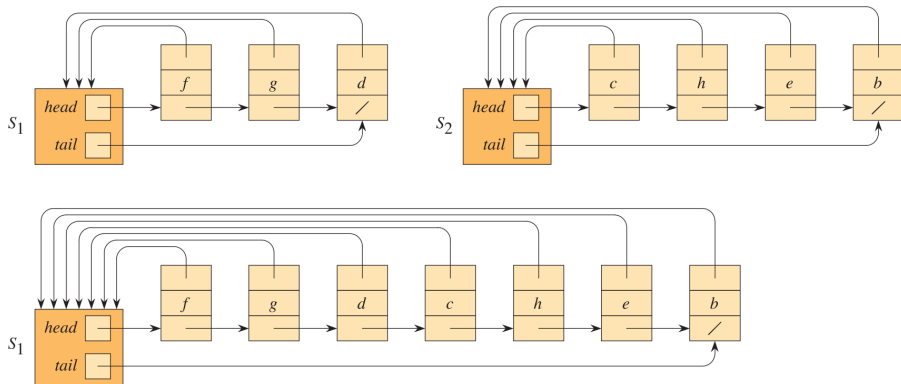
- با استفاده از این پیاده‌سازی توسط لیست پیوندی، هر دو تابع  $\text{Find-Set}$  و  $\text{Make-Set}$  در زمان  $O(1)$  انجام می‌شوند. برای پیاده‌سازی  $\text{Make-Set}(x)$  کافی است یک لیست پیوندی جدید بسازیم که تنها  $x$  را شامل شود. برای  $\text{Find-Set}(x)$  اشاره‌گر  $x$  که به مجموعه آن اشاره می‌کند را دنبال می‌کنیم و سپس اولین عنصر لیست که نماینده آن مجموعه است بر می‌گردانیم. برای مثال در شکل زیر  $\text{Find-Set}(g)$  مقدار  $f$  را باز می‌گرداند.





## مجموعه‌های مجزا توسط لیست پیوندی

- ساده‌ترین پیاده‌سازی عملیات Union توسط لیست پیوندی زمان بیشتری نسبت به عملیات Make-Set و Find-Set نیاز دارد. شکل زیر عملیات  $\text{Union}(x, y)$  برای پیوند دادن لیست  $y$  به انتهای لیست  $x$  را نشان می‌دهد.

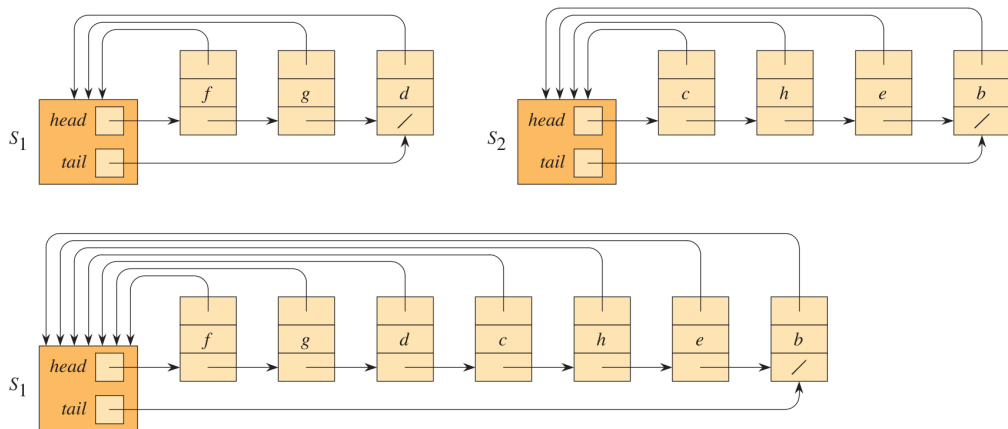


## مجموعه‌های مجزا توسط لیست پیوندی

- در نتیجه نماینده لیست  $x$  نماینده مجموعه حاصل شده می‌شود. برای اینکه سریعاً بتوانیم مکانی را که لیست  $y$  باید به آن اضافه شود محاسبه کنیم، از اشاره‌گر  $tail$  از لیست  $x$  استفاده می‌کنیم. از آنجایی که همهٔ اعضای لیست  $y$  به لیست  $x$  می‌پیوندند، تابع  $Union$  مجموعه لیست  $y$  را تخریب می‌کند. تابع  $Union$  همچنین باید اشاره‌گر به مجموعه‌ای که اعضای لیست  $y$  متعلق به آن هستند را نیز به روز رسانی کند. این به روز رسانی در زمان خطی نسبت به طول لیست  $y$  انجام می‌شود.

# مجموعه‌های مجزا توسط لیست پیوندی

- در شکل زیر عملیات  $\text{Union}(g, e)$  همه اشاره‌گرهای عناصر  $c$  و  $h$  و  $e$  و  $b$  را به روز رسانی می‌کند.



## مجموعه‌های مجزا توسط لیست پیوندی

- می‌توانیم دنباله‌ای از  $m$  عملیات بر روی  $n$  عنصر را به گونه‌ای بسازیم که به زمان  $\Theta(n^2)$  نیاز داشته باشد. با عناصر  $x_1, x_2, \dots, x_n$  آغاز کنیم و تعداد  $n$  عملیات Make-Set فراخوانی می‌کنیم و به دنبال آن  $n - 1$  عملیات Union انجام می‌دهیم به گونه‌ای که  $m = 2n - 1$  باشد.

Operation	Number of objects updated
MAKE-SET( $x_1$ )	1
MAKE-SET( $x_2$ )	1
$\vdots$	$\vdots$
MAKE-SET( $x_n$ )	1
UNION( $x_2, x_1$ )	1
UNION( $x_3, x_2$ )	2
UNION( $x_4, x_3$ )	3
$\vdots$	$\vdots$
UNION( $x_n, x_{n-1}$ )	$n - 1$

## مجموعه‌های مجزا توسط لیست پیوندی

- بنابراین تعداد  $n$  عملیات Make-Set در زمان  $\Theta(n)$  انجام می‌شود. از آنجایی که  $i$  امین عملیات Union تعداد  $i$  عنصر را به روز رسانی می‌کند، تعداد کل عناصر به روز رسانی شده توسط  $n - 1$  عملیات Union یک سری حسابی به صورت زیر می‌سازد :

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

تعداد کل عملیات  $2n - 1$  است و بنابراین عملیات به طور میانگین در زمان  $\Theta(n)$  به ازای هر یک از عملیات انجام می‌شود.

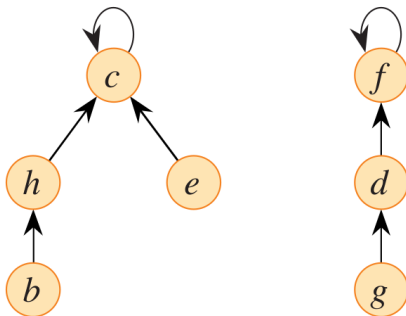
- یک پیاده‌سازی سریع‌تر مجموعه‌های مجزا توسط درخت انجام می‌شود، به طوری که هر مجموعه توسط ریشه یک درخت نشان داده می‌شود و هر عنصر در یک مجموعه یک رأس از یکی از درخت‌هاست. در نتیجه خانواده مجموعه‌های مجزا یک جنگل<sup>1</sup> را تشکیل می‌دهد که خانواده‌ای از درخت‌هاست.

---

<sup>1</sup> disjoint-set forest

## مجموعه‌های مجزا توسط درخت

– در شکل زیر پیاده‌سازی مجموعه‌های مجزا توسط درخت‌ها نمایش داده شده است. ریشه هر درخت نماینده یک مجموعه است.



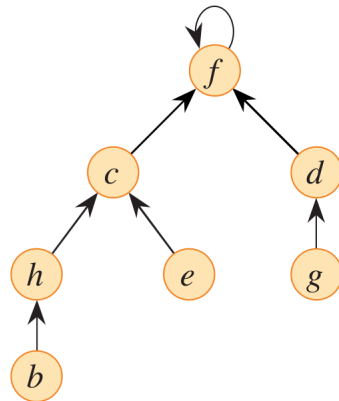
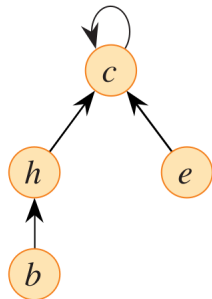
## مجموعه‌های مجزا توسط درخت

- سه عملیات مجموعه‌های مجزا پیاده‌سازی ساده‌ای دارند. عملیات Make-Set یک درخت با یک رأس می‌سازد. عملیات Find-Set اشاره‌گرهای پدر را دنبال می‌کند تا ریشه یک درخت را پیدا کند.



## مجموعه‌های مجزا توسط درخت

- عملیات Union باعث می‌شود ریشه یکی از درخت‌ها فرزند ریشه دیگر قرار بگیرد.



- با چنین پیاده‌سازی، دنباله‌ای از  $n - 1$  عملیات Union می‌تواند درختی بسازد که زنجیره‌ای از  $n$  رأس است.
- برای جلوگیری از ایجاد چنین شرایطی، از روشی استفاده می‌کنیم که زمان اجرا را کاهش دهد. روشی که استفاده می‌کنیم بدین صورت است که ریشه درختی که تعداد کمتری رأس دارد به ریشه درختی اشاره می‌کند که تعداد بیشتری رأس دارد.
- بنابراین برای هر رأس مرتبه <sup>1</sup> آن را نیز ذخیره می‌کنیم. مرتبه یک رأس در واقع ارتفاع آن رأس است. بنابراین در روش اجتماع توسط مرتبه <sup>2</sup> ، در عملیات Union ریشه درخت با مرتبه کمتر به ریشه درخت با مرتبه بیشتر اشاره می‌کند.

---

<sup>1</sup> rank

<sup>2</sup> union by rank

## مجموعه‌های مجزا توسط درخت

- در پیاده‌سازی این روش هر رأس  $x$  یک عدد صحیح  $x.rank$  را نگهداری می‌کند که ارتفاع  $x$  است. ارتفاع رأس  $x$  تعداد یال‌هایی است که بر روی بلندترین مسیر ساده از یکی از برگ‌ها به  $x$  قرار می‌گیرد.
- وقتی تابع  $Make-Set$  مجموعه‌ای با یک عضو می‌سازد، تنها رأس مجموعه که ریشه درخت است دارای مرتبه صفر است.
- تابع  $Find-Set$  مرتبه عناصر را تغییر نمی‌دهد.

## مجموعه‌های مجزا توسط درخت

- تابع Union دو حالت دارد، بسته به این که ریشه درخت‌ها دارای مرتبه یکسان باشند یا خیر.
- اگر دو ریشه دارای مرتبه نابرابر باشند، ریشه درخت با مرتبه بالاتر پدر ریشه درخت با مرتبه پایین‌تر قرار می‌گیرد، اما مرتبه‌ها تغییری نمی‌کنند. اگر ریشه‌ها مرتبه یکسان داشته باشند، به طور دلخواه یکی از ریشه‌ها پدر قرار می‌گیرد و مرتبه آن یک واحد افزایش پیدا می‌کند.

# مجموعه‌های مجزا توسط درخت

- شبه کدهای زیر این روش پیاده‌سازی را نشان می‌دهند.

---

## Algorithm Find-Set

---

```
function FIND-SET(x)
1: while  $x \neq x.p$  do ▷ not the root?
2:    $x = x.p$         ▷ x becomes the parent of x
3: return x          ▷ return the root
```

---

---

## Algorithm Make-Set

---

```
function MAKE-SET(x)
1:  $x.p = x$ 
2:  $x.rank = 0$ 
```

---

---

## Algorithm Union

---

function UNION(x,y)  
1: Link(Find-Set(x),Find-Set(y))

---

---

## Algorithm Link

---

function LINK(x,y)  
1: if x.rank > y.rank then  
2:     y.p = x  
3: else  
4:     x.p = y  
5:     if x.rank == y.rank then  
6:         y.rank = y.rank + 1

---

- تابع Link که توسط Union فراخوانی می‌شود اشاره‌گرهایی به دو ریشه به عنوان ورودی دریافت می‌کند.

## مجموعه‌های مجزا توسط درخت

- می‌توان اثبات کرد، در یک ساختار درختی، برای دنباله‌ای از  $m$  عملیات که تعداد  $n$  تای آنها Make-Set هستند، زمان اجرا برابر است با  $\Theta(m \lg n)$ .
- بنابراین هر یک از عملیات به طور میانگین در زمان  $\Theta(\lg n)$  انجام می‌شود.



- برای بهبود زمان اجرا، از روشی به نام فشرده‌سازی مسیر<sup>1</sup> استفاده می‌شود. در این روش تابع Find-Set به گونه‌ای تغییر داده می‌شود که در هنگام پیدا کردن نماینده یک مجموعه، در حین پیمایش درخت، همه رئوس درخت مقدار اشاره‌گر پدر خود را تغییر داده و ریشه درخت را پدر خود قرار می‌دهند.

---

<sup>1</sup> path compression

- تابع یافتن مجموعه مجزا (Find-Set) با استفاده از روش فشرده‌سازی مسیر به صورت زیر پیاده‌سازی می‌شود. این تابع به صورت بازگشتی اجرا می‌شود. با اجرای تابع به صورت بازگشتی، پس از پیدا شدن مسیر به سمت ریشه، در بازگشت پدر هر رأس برابر با ریشه قرار می‌گیرد.

---

### Algorithm Find-Set

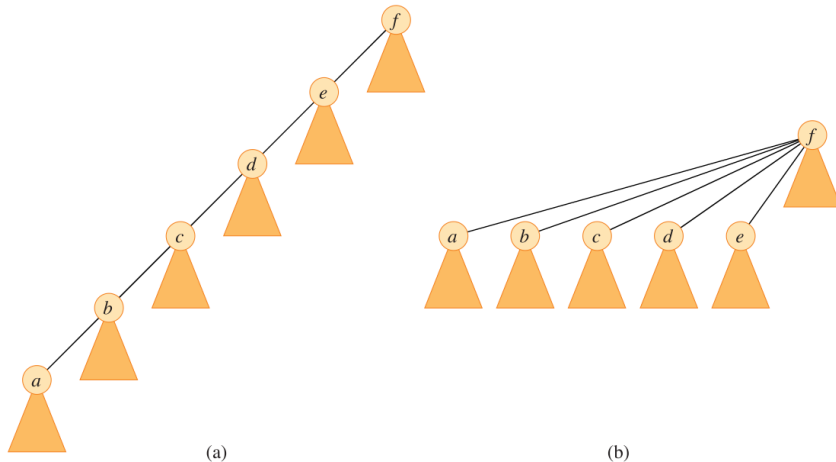
---

```
function FIND-SET(x)
1: if  $x \neq x.p$  then  $\triangleright$  not the root?
2:    $x.p = \text{Find-Set}(x.p)$   $\triangleright$  the root becomes the parent
3: return  $x.p$   $\triangleright$  return the root
```

---

## مجموعه‌های مجزا توسط درخت

- بنابراین وقتی تابع  $\text{Find-Set}(a)$  اجرا می‌شود، درخت سمت چپ به درخت سمت راست تبدیل می‌شود.



## مجموعه‌های مجزا توسط درخت

- توجه کنید که وقتی از روش فشرده‌سازی مسیر استفاده می‌کنیم، مرتبه رأس  $x$  یعنی  $x.rank$  یک کران بالا بر روی ارتفاع  $x$  خواهد بود، نه ارتفاع دقیق  $x$ .
- با استفاده از این روش، زمان اجرا به  $O(m\alpha(n))$  کاهش پیدا می‌کند. تابع  $\alpha(n)$  تابعی با کاهش بسیار آرام است.
- در همه کاربردهای مجموعه‌های مجزا می‌توان ثابت کرد که  $\alpha(n) \leq 4$ ، بنابراین زمان اجرای عملیات مجموعه‌های مجزا به مرتبه خطی نسبت به  $m$  تبدیل می‌شود.
- در نتیجه زمان اجرای هر یک از عملیات، با استفاده از پیاده‌سازی درختی و فشرده‌سازی مسیر، به طور میانگین برابر با  $O(\alpha(n))$  خواهد بود.