ساختمان داده
دانشکده ریاضی. دانشگاه صنعتی خواجه نصیرالدین طوسی. پاییز ۱۴۰۲

۱ ساختمان داده انتزاعی مجموعه‌های مجزا

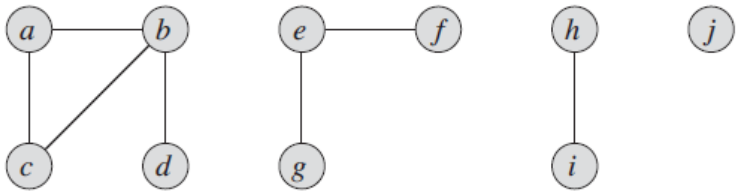
می‌خواهیم ساختمان داده‌ای داشته باشیم که اعمال زیر را پشتیبانی کند.

- Make-Set(x)
  یک مجموعه تک عضوی شامل عنصر جدید x ایجاد کند
- Union(x, y)
  مجموعه‌ای که شامل عنصر x است را در مجموعه‌ای که شامل عنصر y است ادغام کند
- Find-Set(x)
  مجموعه‌ای که شامل عنصر x است را برمی‌گرداند. دقت کنید هر مجموعه یک برچسب منحصر بفرد دارد. این تابع برچسب مجموعه را برمی‌گرداند. چون مجموعه‌های ایجاد شده اشتراکی با هم ندارند، در بعضی از پیاده سازی ها، یکی از اعضای مجموعه به عنوان برچسب مجموعه در نظر گرفته می‌شود.

۲ کاربردهای ساختمان داده مجموعه‌های مجزا

به دو کاربرد اشاره می‌کنیم.

- **محاسبه مولفه‌های همبندی گراف.** می‌خواهیم با داشتن دنباله یالهای یک گراف، مولفه‌های همبندی گراف را محاسبه کنیم. یک مولفه همبندی در گراف $G(V, E)$ یک زیرمجموعه $S \subseteq V$ از رئوس است بطوریکه هیچ یالی از S به خارج آن نباشد و خود S همبند باشد. شکل زیر یک گراف با 4 مولفه همبندی را نشان می‌دهد.



برای محاسبه های مولفه همبندی می‌توان از امکانات ساختمان داده مجموعه‌های مجزا استفاده کرد. در شروع برای هر راس $x \in V$ تابع Make-Set(x) را فراخوانی می‌کنیم. با این کار n مجموعه مجزای تک عضوی ایجاد می‌شود. انگار در شروع کار هر راس یک مولفه است. سپس دنباله یالها را پردازش می‌کنیم. با مشاهده یال (u, v) تابع Union(u, v) را فراخوانی می‌کنیم. اگر دو راس u و v قبلا در یک مولفه باشند، این کار تغییری در مولفه‌های گراف ایجاد نمی‌کند. در غیر این صورت دو مولفه مجزا شامل u و v را در هم ادغام می‌شوند. در انتهای کار، زیرمجموعه‌های بدست آمده همان مولفه‌های همبندی گراف ورودی هستند.

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,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

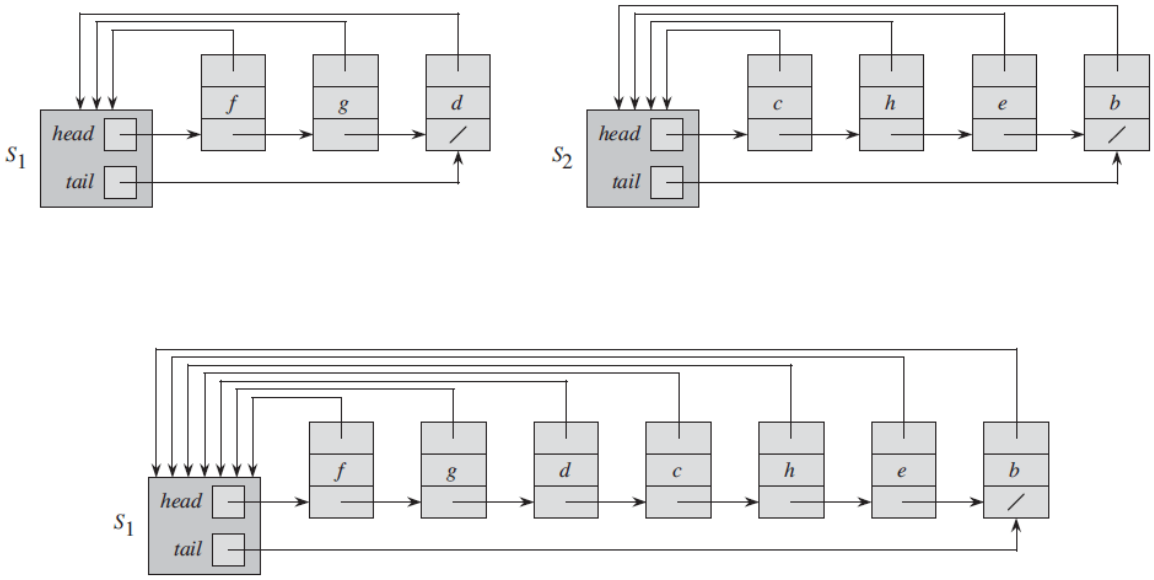
- **پیاده سازی الگوریتم کراسکال** الگوریتم کراسکال، مانند الگوریتم پریم، درخت فراگیر کمینه در یک گراف وزن دار را پیدا می‌کند. هر یال گراف وزن دارد. می‌خواهیم درختی فراگیر در گراف ورودی پیدا کنیم که مجموع وزن یالهایش کمینه باشند. برای این منظور، الگوریتم کراسکال ابتدا یالهای را گراف را بر حسب وزن از کوچک به بزرگ مرتب می‌کند. فرض کنید دنباله زیر یالهای گراف بعد از مرتب سازی باشند.

$$e_1, e_2, \cdots, e_m$$

در شروع کار، هر راس یک مولفه همبندی است. یالها را به ترتیب پردازش می‌کنیم. با مشاهده یال $e_i = (x, y)$ اگر $\text{Find-Set}(x) = \text{Find-Set}(y)$ یعنی دو راس x و y جزو یک مولفه هستند (قبلا مسیری بین آنها وجود دارد.) لذا یال e_i در این حالت دور انداخته می‌شود در غیر اینصورت به عنوان یکی از یالهای درخت فراگیر کمینه انتخاب می‌شود و سپس تابع $\text{Union}(x, y)$ فراخوانی می‌شود تا مولفه‌های شامل x و y در هم ادغام شوند.

۳ یک پیاده سازی با استفاده از لیستهای پیوندی

مجموعه‌های مجزا را می‌توان با استفاده از ساختار لیست پیوندی یک سویه پیاده‌سازی کرد. هر زیرمجموعه بصورت یک لیست پیوندی که شامل اعضای زیر مجموعه است پیاده‌سازی می‌شود. برای سرعت بخشیدن به عمل $\text{Find-Set}(x)$ از هر عنصر لیست یک پیوند به سر لیست قرار می‌دهیم. قسمت بالای شکل زیر لیست پیوندی مربوط به دو زیر مجموعه $S_1 = \{f, g, d\}$ و $S_2 = \{c, h, e, b\}$ را نشان می‌دهد. قسمت پایین شکل زیر مجموعه S_2 در مجموعه S_1 ادغام شده است.



فرض کنید S_x زیرمجموعه شامل x و S_y زیرمجموعه شامل y باشد. برای پیاده سازی $\text{Union}(x, y)$ در صورتیکه $S_y \neq S_x$ می‌توانیم پیوندهای عناصر S_y را بروزرسانی کنیم و همه آنها را به سر لیست مربوط به S_x جهت دهی کنیم. قاعدتا اگر S_y بزرگتر از S_x باشد، بهتر است پیوندهای لیست S_x را به جای لیست S_y بروزرسانی کنیم. اگر ادغام را بدون توجه به اندازه مجموعه‌ها انجام دهیم، فراخوانی $n-1$ بار تابع Union ممکن است به تعداد $O(n^2)$ بروزرسانی نیاز داشته باشد. به مثال زیر توجه کنید.

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

اینجا ابتدا n فراخوانی Make-Set داریم که مجموعه‌های تک عضوی $\{x_1\}, \{x_2\}, \cdots, \{x_n\}$ را ایجاد میکند. سپس با ترتیب داده شده زیرمجموعه‌ها در هم ادغام می‌شوند تا اینکه یک مجموعه n عضوی که شامل همه اعضا باشد ایجاد شود. دقت کنید هر بار زیرمجموعه تک عضوی $\{x_i\}$ و مجموعه $\{x_1, \cdots, x_{i-1}\}$ ادغام می‌شوند. اگر پیوندهای لیست مربوط به مجموعه بزرگتر یعنی $\{x_1, \cdots, x_{i-1}\}$ بروزرسانی شوند، تعداد بروزرسانی ها در هر ادغام برابر با $i-1$ خواهد بود و لذا جمع بروزرسانی ها برابر با

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

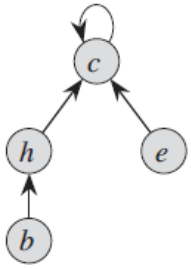
خواهد شد. اما اگر ادغام ها طوری انجام شود که مجموعه کوچکتر در مجموعه بزرگتر ادغام شود، می‌توان نشان داد که انجام $n-1$ عمل Union زمان اجرایش حداکثر $O(n \log n)$ خواهد بود.

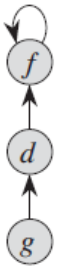
لم. اگر موقع ادغام دو مجموعه S_1 و S_2 پیوندهای لیست کوچکتر بروزرسانی شوند، زمان $n-1$ عمل Union حداکثر $O(n \log n)$ خواهد بود.

اثبات. کافی است حداکثر تعداد دفعاتی که پیوند یک عنصر تغییر می‌کند را بشماریم. دقت کنید چون پیوندهای لیست کوچکتر بروزرسانی می‌شوند اندازه مجموعه حاصل بعد از ادغام حداقل دو برابر اندازه مجموعه کوچکتر است. عنصر x را در نظر بگیرید. فرض کنید در طول محاسبات S_x مجموعه شامل x باشد. بعد از اولین بروزرسانی پیوند x داریم $|S_x| \geq 2$. بعد از دومین بروزرسانی اداریم $|S_x| \geq 4$. بعد از k امین بروزرسانی داریم $|S_x| \geq 2^k$. چون اندازه هر مجموعه حداکثر n است، لذا حداکثر $\lceil \log n \rceil$ بروزرسانی روی پیوند x انجام می‌شود. پس در مجموعه $n\lceil \log n \rceil = O(n \log n)$ بروزرسانی انجام می‌شود. □

۴ پیاده سازی با استفاده از درختهای دودویی

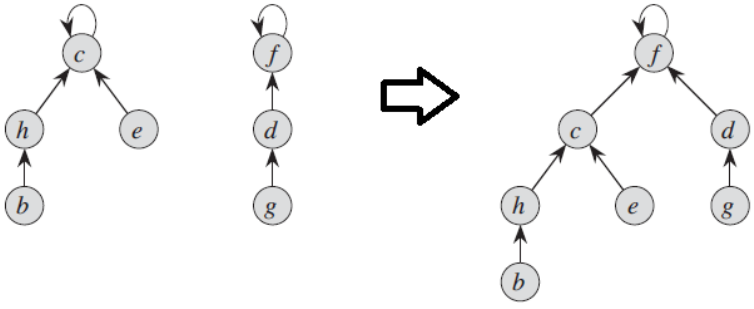
می‌توانیم از یک درخت دودویی برای نمایش یک مجموعه استفاده کنیم. عنصری که در ریشه درخت ذخیره می‌شود همان برچسب درخت است. در واقع عنصری که در ریشه درخت ذخیره می‌شود یکی از عناصر مجموعه است که به عنوان برچسب مجموعه از آن استفاده می‌کنیم. هر راس درخت به پدرش اشاره می‌کند. برای پیدا کردن برچسب یک مجموعه از راس مورد سوال به سمت ریشه حرکت می‌کنیم. توجه کنید که ریشه به خودش اشاره می‌کند چون پدری ندارد. در شکل زیر دو مجموعه $S1$ و $S2$ داریم که هر کدام با یک درخت دودویی نمایش داده شده است.





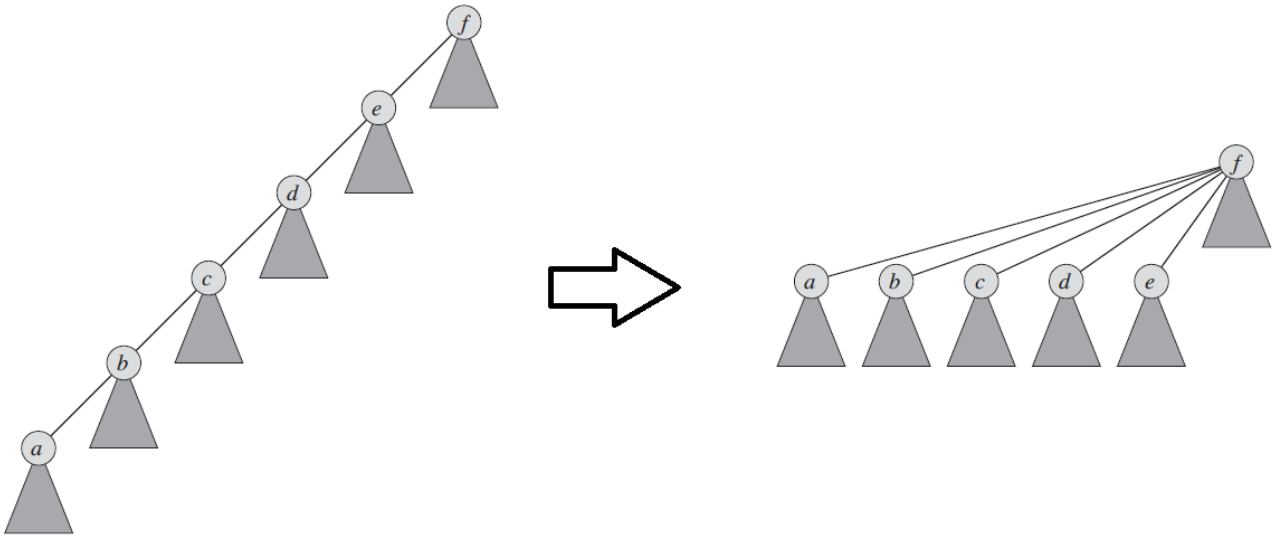
S1 = {b, h, c, e} **S2={f, d, g}**

حال برای ادغام دو مجموعه $S1$ و $S2$ می‌توان ریشه درخت یکی از این مجموعه‌ها را به ریشه درخت دیگری وصل کرد.



همانطور که می‌توانید حدس بزنید، با این پیاده‌سازی، عمل Union خیلی سریع انجام می‌شود اما عمل FindSet کند می‌شود چون برای پیدا کردن برچسب مجموعه حاوی x باید از عنصر x شروع کنیم و به سمت ریشه حرکت کنیم. اگر ادغامها بدون نظام خاصی انجام شود، عمل FindSet می‌تواند $O(n)$ زمان ببرد.
برای تسریع عمل FindSet پیرو ساختاری که Robert Tarjan پیشنهاد داده است، از دو ایده استفاده می‌کنیم.

- **Union by rank** بروزرسانی ریشه درختی که رتبه کمتری دارد. اینجا رتبه در واقع یک کران بالا برای ارتفاع درخت است. اگر نخواهیم خیلی دقیق بگوییم، درختی که رتبه (ارتفاع) کمتری دارد ریشه‌اش به درخت با رتبه (ارتفاع) بیشتر اشاره می‌کند.
- **Path compression** موقع عمل FindSet(x) همه رئوس واقع در مسیر x به ریشه، پدرشان به ریشه درخت تغییر می‌کند. مانند شکل زیر که بعد از انجام عمل FindSet(a) ساختار درخت تغییر کرده است.



شبه کد اعمال مختلف ساختار داده مجموعه‌های مجزا بر اساس پیاده‌سازی تارجان در زیر آمده است (شکلها و شبه کد برگرفته از کتاب CLRS هستند.)

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

```
UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1
```

```
FIND-SET(x)
1  if x ≠ x.p
2       x.p = FIND-SET(x.p)
3  return x.p
```

۱.۴ زمان اجرای اعمال ساختار داده مجموعه‌های مجزا

می‌توان نشان داد که زمان اجرای m عمل پی در پی (متشکل از ترکیبی از MakeSet و Union و FindSet) با استفاده از ایده‌های تارجان، حداکثر $O(m\alpha(n))$ است وقتی که n تعداد دفعاتی که است که MakeSet فراخوانی می‌شود. اینجا $\alpha(n)$ معکوس تابع آکرمن Ackermann's function است که رشد فوق العاده کمی دارد. در واقع مقدار $\alpha(n)$ برای $n = 2^{2048}$ کمتر از 5 است! پس بطور عملی می‌توانیم فرض کنیم که هر بروزرسانی این ساختار داده بطور متوسط زمانش $O(1)$ است. برای جزئیات بیشتر در مورد این تابع و تحلیل سرشکنی ساختار داده تارجان کتاب مرجع را ببینید.