ناموفق خواهد بود، زیرا این عمل در محل «کلید تازه حذف‌شده» متوقف خواهد شد که اگر انجام عمل حذف ضرورت داشته باشد، باید برای حل این مشکل چاره‌ای بیندیشیم.

می‌توان اثر خوشه‌ی پرشده را با درهم‌سازی دوگانه کاهش داد. در این روش هنگام برخورد، تابع دومی (مانند $h_2(x)$) را برای درهم‌سازی به کار می‌بریم و به جای جست‌وجوی خطی،یعنی $i+1$، $i+2$، ... به ترتیب مکان‌های $i+h_2(x)$، $i+2h_2(x)$، ... را جست‌وجو می‌کنیم (باز هم به ترتیب چرخشی). اگر کلید دیگری مانند y به $i+h_2(x)$ نگاشته شود، محل بعدی $i+h_2(x)+h_2(y)$ خواهد بود، نه $i+2h_2(x)$. اگر $h_2(x)$ از $h_2(y)$ مستقل باشد، با پدیده‌ی خوشه‌ی پرشده روبه‌رو نخواهیم شد. باید در گزینش تابع دوم درهم‌ساز دقت کنیم تا دنباله‌ی $i+h_2(x)$، $i+2h_2(x)$، ...، $i+nh_2(x)$ تمام جدول را در بر گیرد (که اگر اعداد $h_2(x)$ و n نسبت به هم اول باشند، این‌گونه خواهد شد).

اشکال عمده‌ی درهم‌سازی دوگانه نیاز به محاسبات اضافی، برای محاسبه‌ی مقدار دوم، هنگام عمل جست‌وجوست. یک روش برای کم کردن محاسبات، برگزیدن تابع دوم درهم‌ساز به گونه‌ای است که از تابع نخست کاملاً مستقل نباشد، اما سبب کاهش خوشه‌ی پرشده گردد. یک نمونه از این روش برگزیدن تابع $h_2(x)$ به صورت

$$h_2(x)=\begin{cases}1 & ,h_1(x)=0\\ m-h_1(x) & ,h_1(x)\neq 0\end{cases}$$

است (با این فرض که m عددی اول است و $h_1(x)=x \bmod m$).

۴-۵ مسأله‌ی union-find

مسأله‌ی union-find (که آن را مسأله‌ی هم ارزی نیز می‌گویند) نمونه‌ای خوب از کاربرد ساختمان‌های داده‌ای عجیب و غریب برای بهبود کارایی الگوریتم‌هاست. مسأله چنین است: n عنصر x_1، x_2، ... و x_n موجودند که به گروه‌هایی دسته‌بندی شده‌اند. در آغاز، هر عنصر به تنهایی یک گروه تشکیل می‌دهد. دو نوع عمل روی عناصر و گروه‌ها به ترتیب دل‌خواه انجام می‌گردد:

find(i): نام گروهی را برمی‌گرداند که در برگیرنده‌ی x_i است.

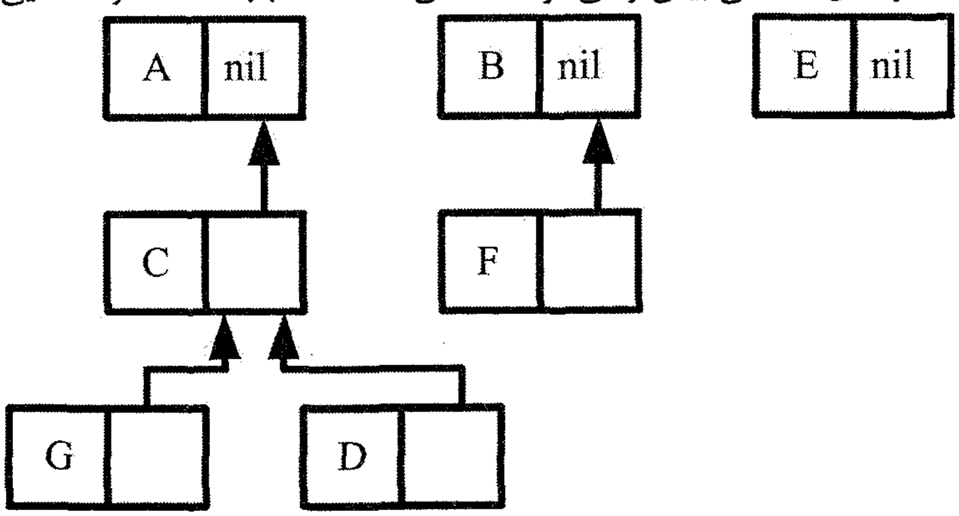
union(A,B): گروه A و B را با هم ترکیب می‌کند تا یک گروه تازه با نامی یکتا به وجود آورد (نام‌ها نباید تکراری باشند).

هدف، طراحی ساختاری است که هر دنباله‌ای از این دو عمل را به کارآمدترین شیوه‌ی ممکن پشتیبانی کند.

از آنجا که همه‌ی عناصر، پیشاپیش شناخته‌شده (و از ۱ تا n اندیس‌گذاری‌شده) هستند، می‌توان آرایه‌ی X[1..n] را به آن‌ها تخصیص داد. راه سرراست حل مسأله، ذخیره‌ی نام گروه دربرگیرنده‌ی عنصر iام در X[i] است. روشن است که عمل find به سادگی با نظر به آرایه انجام می‌شود، اما عمل

union نیاز به زمان بیش‌تری دارد. فرض کنیم نتیجه‌ی union(A,B) گروهی ترکیبی با نام A شود؛ در این صورت، لازم است هرجا که نام گروهی B است، آن را به A تغییر دهیم.

اینک، روشی متفاوت برای حل این مسأله ارائه می‌کنیم. به جای ساده‌سازی عمل find، عمل union را با یاری نشانی غیرمستقیم، ساده می‌کنیم. هر خانه‌ی آرایه، رکوردی است شامل نام عنصر و اشاره‌گری به یک رکورد دیگر. در آغاز، همه‌ی اشاره‌گرها nil هستند. عمل union(A,B) اشاره‌گر درون رکورد B را چنان تغییر می‌دهد که به رکورد شامل A اشاره کند و یا برعکس (به زودی در این مورد بحث خواهیم کرد). پس از چند عمل union، ساختمان‌داده به مجموعه‌ای از درخت‌ها مانند شکل ۴-۱۶ تبدیل خواهد شد. هر درخت، متناظر با یک گروه و هر گره، متناظر با یک عنصر است. نام هر گروه، از ریشه‌ی درخت متناظر با آن گرفته می‌شود. برای یافتن گروهی که شامل عنصر G است، از اشاره‌گر G شروع می‌کنیم تا به ریشه برسیم. (ریشه، گرهی است که اشاره‌گر آن nil است.) این فرایند، شبیه تغییر نشانی پستی است که در آن، به جای اعلام نشانی تازه به همه، نامه‌های نشانی پیشین به نشانی تازه فرستاده می‌شوند؛ البته یافتن نشانی درست دشوارتر می‌شود، یعنی کارایی عمل find کم‌تر می‌گردد. این ناکارآمدی هنگامی بیش‌تر می‌شود که عمل union، سبب ساخت درخت‌هایی بلند گردد.



شکل ۴-۱۶ نمایش مسأله‌ی union-find

ایده‌ی اصلی برای کارآمد کردن این ساختمان‌داده، متوازن ساختن و هرس کردن درخت‌هاست. به تازگی دیدیم که می‌ارزد، زمانی بیش‌تر را صرف توازن ساختمان‌داده کنیم. عمل union(A,B) را در شکل ۴-۱۶ در نظر بگیرید. دو حالت ممکن است: یا اشاره‌گر B را به گونه‌ای تنظیم می‌کنیم که به A اشاره کند، یا اشاره‌گر A را به گونه‌ای تنظیم می‌کنیم که به B اشاره کند. روشن است که گزینه‌ی نخست منجر به تشکیل درختی متوازن‌تر می‌گردد. پس در رکورد متناظر با ریشه، علاوه بر نام گروه، تعداد عناصر آن را نیز نگه‌داری می‌کنیم تا به سرعت تشخیص دهیم کدام درخت متوازن‌تر است.

تعریف توازن: هنگام انجام عمل union، اشاره‌گر گروه کوچک‌تر به گونه‌ای تنظیم می‌شود که به گروه بزرگ‌تر اشاره کند (هنگام هم‌اندازه بودنِ هر دو گروه، یکی را به

دل‌خواه برمی‌گزینیم). اندازه‌ی گروه ترکیبی حاصل نیز محاسبه شده، در میدان مناسبی از رکورد ریشه قرار می‌گیرد.

اگر عمل union، بنا به تعریف توازن (چنان که گفته شد) رفتار کند، ارتفاع درخت‌ها هرگز از $\log_2^n$ بیش‌تر نخواهد شد. این موضوع در قضیه‌ی ۴-۲ نشان داده شده است.

☐ قضیه‌ی ۴-۲

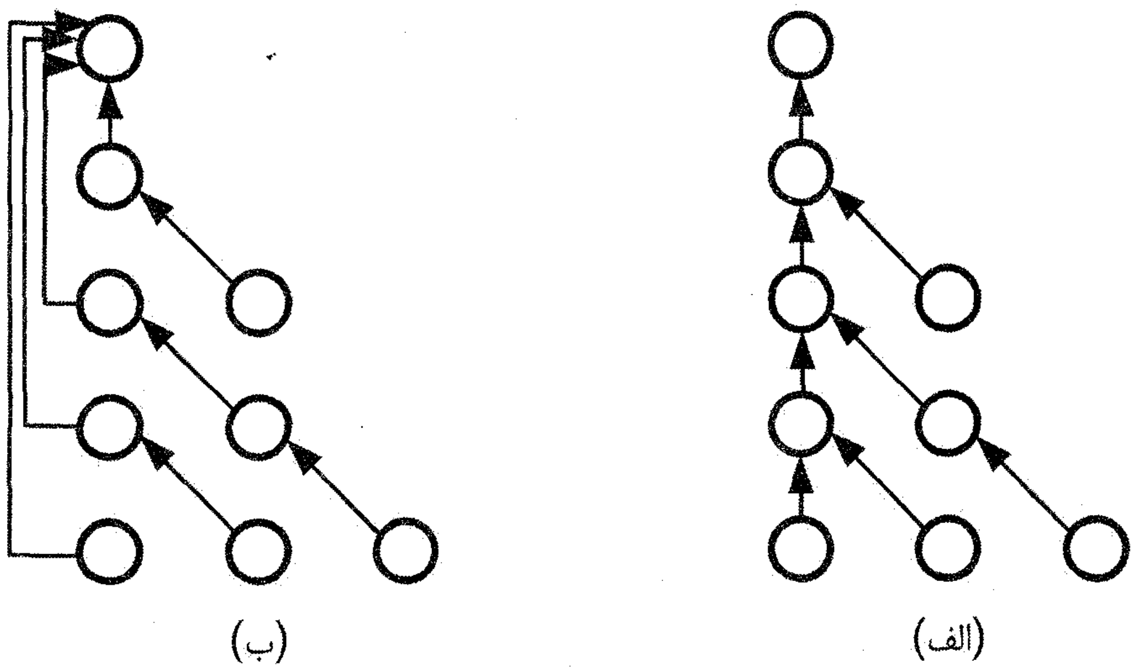
هر درخت متوازن به ارتفاع h، دست‌کم 2^h عنصر دارد.

برهان: اثبات با استقرا روی تعداد اعمال union است. روشن است که قضیه برای نخستین union که سبب ایجاد درختی با دو عنصر و به ارتفاع یک می‌شود، درست است. عمل union(A,B) را در نظر بگیرید و فرض کنید A، گروه بزرگ‌تر باشد؛ یعنی B باید به A اشاره کند. ارتفاع درخت‌های متناظر با گروه‌های A و B را به ترتیب با h(A) و h(B) نشان می‌دهیم. ارتفاع درخت ترکیبی حاصل، بیشینه‌ی h(A) و h(B)+1 است. اگر h(A) بزرگ‌تر باشد، آنگاه درخت ترکیبی حاصل، هم‌ارتفاع با درخت A ولی با تعداد عناصر بیش‌تری است؛ در این حالت، به روشنی قضیه برقرار است. در حالت دیگر، تعداد عناصر درخت ترکیبی حاصل، دست‌کم دو برابر تعداد عناصر درخت B (زیرا B کوچک‌تر از A فرض شده بود) و ارتفاعش یکی بیش از ارتفاع اولیه‌ی B است. بازهم می‌بینیم که قضیه برقرار است.

☐

از قضیه‌ی ۴-۲ نتیجه می‌شود که عمل find، حداکثر از $\log_2^n$ اشاره‌گر عبور می‌کند. عمل union، همواره زمان ثابتی می‌گیرد. در نتیجه، حداکثر گام‌های هر دنباله‌ای از m عمل (چه union، چه find) هنگامی که $m \geq n$، از O(mlogn) خواهد بود.

می‌توان کارایی ساختمان‌داده‌ی union-find را بهبود بخشید. دوباره مثال فرستادن نامه‌های یک نشانی پستی به یک نشانی دیگر را در نظر بگیرید. اگر چندین تغییر نشانی روی دهد، نامه از یک نشانی به نشانی دیگر می‌رود تا سرانجام به مقصد برسد. خوب است به همه‌ی ایستگاه‌های پستی که عمل ارسال نامه را به نشانی بعدی انجام می‌دهند، اعلام کنیم که مقصد نهایی کجاست. در این صورت، این ایستگاه‌ها می‌توانند نامه‌ها را یک‌راست به مقصد نهایی بفرستند. در مورد این ساختمان‌داده، ما می‌توانیم پس از پیمایش اشاره‌گرها از یک رکورد به ریشه، اشاره‌گرهای درون مسیر را به گونه‌ای تغییر دهیم که مستقیماً به ریشه اشاره کنند (شکل ۴-۱۷ را ببینید). به این عمل، فشرده‌سازی مسیر گفته می‌شود. پیمایش دوباره‌ی مسیر برای انجام این کار، تعداد گام‌ها را دو برابر می‌کند؛ بنابراین پیچیدگی مجانبی زمان عمل find همان مقدار پیش خواهد بود. می‌توانیم هر بار که عمل find را انجام دادیم، فشرده‌سازی مسیر را نیز انجام دهیم. قضیه‌ی بعد که اثباتش نخواهیم کرد، حد بالای خوبی برای بدترین حالت ارائه می‌دهد.



شکل ۴-۱۷ (الف) پیش از فشرده‌سازی مسیر؛ (ب) پس از فشرده‌سازی مسیر (در این شکل، تنها یک مسیر فشرده شده است - مترجمان)

□ قضیه‌ی ۴-۳

اگر هر دو عمل توازن و فشرده‌سازی با هم به کار گرفته شوند، آنگاه تعداد گام‌ها در بدترین حالت، برای هر دنباله‌ای از m عمل (خواه find، خواه union) که m≥n، از $O(m\log^* n)$ خواهد بود که $\log^* n$ تابع لگاریتم پی‌درپی است و این‌گونه تعریف می‌شود: log*1=log*2=1 و برای هر n>2، $\log^* n = 1 + \log^*(\lceil \log_2^n \rceil)$.

□

برای مثال، log*4=1+log*2=2، log*14=1+log*4=3 و log*16=4=log*60000=1+. برای هر $n \le 2^{65536}$ (که تمام کاربردهای عملی را دربرمی‌گیرد) داریم: log*n≤5. بنابراین پیچیدگی هر دنباله‌ای از اعمال union و find تقریباً خطی است (و در عمل هم واقعاً خطی است). توجه کنید اگرچه هنوز هم یک عمل find خاص ممکن است به O(logn) گام نیاز داشته باشد، اما تعداد کل گام‌های O(n) عمل find از O(nlog*n) خواهد بود. این مورد، نمونه‌ای خوب از تحلیل سرشکن‌شده است. در این روش به جای محاسبه‌ی جداگانه‌ی تک‌تک گام‌ها، همه‌ی آن‌ها را با هم می‌شماریم. هنوز هم طراحی الگوریتمی با زمان خطی برای این مسأله، یک مسأله‌ی باز و حل‌نشده است.

۴-۶ گراف‌ها

یک فصل کامل (فصل ۷) را به الگوریتم‌های گراف اختصاص خواهیم داد. در این بخش، درباره‌ی ساختمان‌داده‌هایی بحث می‌کنیم که برای ذخیره‌ی گراف‌ها به کار می‌روند. گراف G=(V,E) از