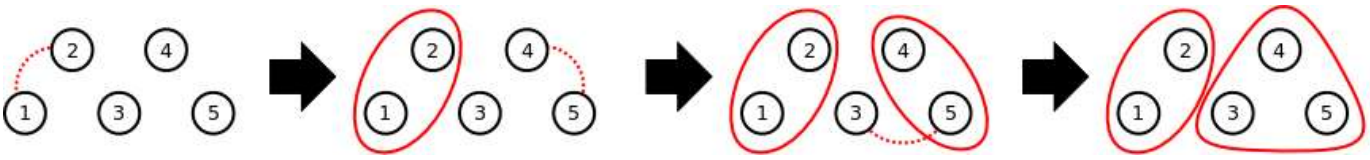
Problem Find-Union

Rozważamy zbiór n elementów, które dla wygody ponumerujemy od 1 do n. Elementy będą łączyć się w coraz większe grupy, jak na rysunku:



Naszym zadaniem jest określać, w dowolnym momencie, czy dwa elementy znajdują się w tej samej grupie. Mówiąc formalnie, musimy zaimplementować dwie operacje:

- *Union(x,y)* — połącz grupy, w których są elementy x i y, w jedną grupę,
- *Find(x)* — znajdź grupę elementu x. W praktyce chodzi o sprawdzenie, czy dwa elementy są w tej samej grupie (sprawdzamy zawsze warunek, czy *Find(x) = Find(y)* dla pewnych dwóch elementów x,y).

Czasami warto jest jeszcze uwzględnić trzecią operację, *Init()*, wywoływaną na początku programu, przed wszystkimi *Find()* i *Union()*. Będzie ona tworzyła niezbędne nam później struktury danych. Naszym celem jest osiągnięcie jak najniższej złożoności obliczeniowej operacji *Find()* i *Union()*.

Implementacja naiwna

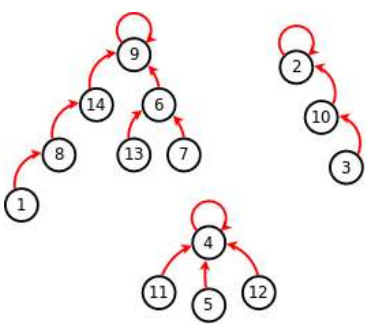
Łatwo zaprojektować działającą implementację obu operacji — po prostu pamiętamy, w której grupie jest każdy z elementów. Operacja *Union(x,y)* sprawdza wszystkie elementy i tym, które są w grupie elementu y, zmienia grupę na tę samą, co element x.

```
Init()
  for i = 1, 2, ..., n
    grupa[x] = x
Find(x)
  return grupa[x]
Union(x,y)
  for i = 1, 2, ..., n
     if grupa[i] = grupa[y]
       grupa[y] = grupa[x]
```

Implementacja ta jest wyjątkowo prosta, jednak złożoność operacji Union wynosi $O(n)$ - za każdym razem musi przeglądnąć wszystkie elementy.

Las zbiorów rozłącznych

Wyobraźmy sobie, że każda grupa jest zorganizowana według pewnej hierarchii. Każdy element *x* "pamięta" numer innego elementu ze swojej grupy, *parent[x]* — nazywamy go **rodzicem** x. W każdej grupie będzie też jeden (i tylko jeden) element, który jest swoim własnym rodzicem — **reprezentant** tej grupy.



Na tym rysunku czerwone strzałki prowadzą od elementu do jego rodzica. Widoczne jest, że w tym zbiorze są trzy grupy — reprezentantem pierwszej jest 9, drugiej 2, zaś trzeciej — 4

Operację *Find(x)* zaprojektujemy tak, aby zwracała reprezentanta grupy, w której jest element x. To oczywiście wystarczy do naszych celów — dwa elementy są w tej samej grupie wtedy i tylko wtedy, gdy

mają tego samego reprezentanta. Aby go znaleźć, przechodzimy od x do jego rodzica, następnie do rodzica jego rodzica, i tak dalej, aż trafimy na element będący własnym rodzicem. Można zapisać tę procedurę w następujący sposób:

```
Find(x)
  a = x
  while a<>parent[a]
     a = parent[a]
   return a
```

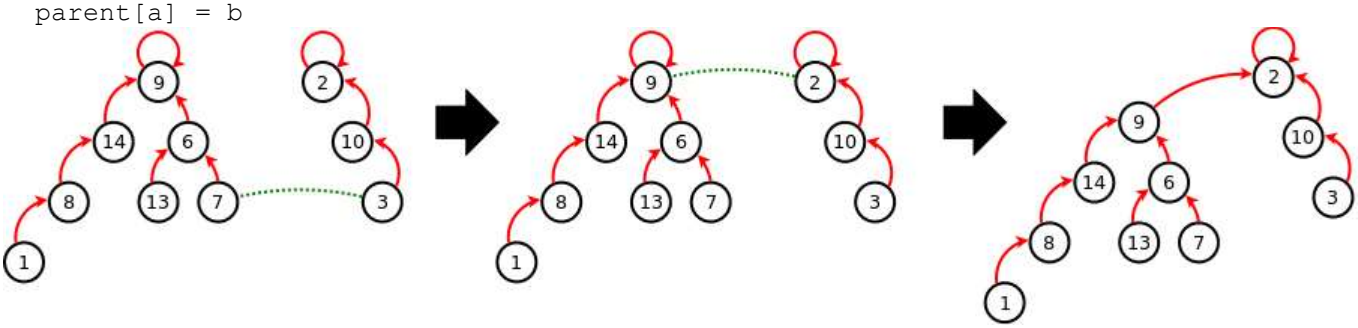
Wygodniej jest jednak użyć rekursji:

```
Find(x)
  if parent[x] = x
    return x
  else
    return Find(parent[x])
```

Zauważmy, że aby to podejście działało, pętla w operacji *Find()* nie może wrócić z powrotem do elementu x, ani żadnego wcześniej odwiedzonego. Zauważmy, że zbiór wszystkich elementów wraz z krawędziami od elementów do ich rodziców jest pewnym grafem, a wymieniona przez nas niepożądana sytuacja to cykl w tym grafie. Dążymy zatem do tego, aby graf ten nie zawierał cyklu — innymi słowy, musi być **lasem**, a każda grupa z osobna — **drzewem**. Stąd pochodzi nazwa tej struktury danych: **las zbiorów rozłącznych**.

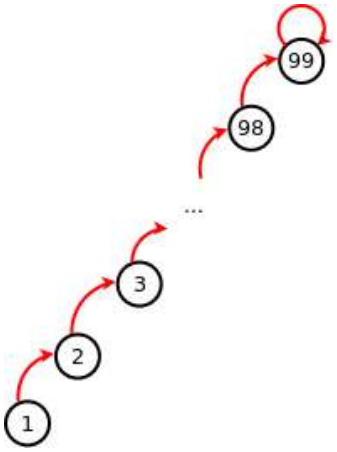
Operacja *Union(x,y)* musi tylko znaleźć reprezentantów obu grup i jednego z nich uczynić rodzicem drugiego:

```
Union(x,y)
  a = Find(x)
  b = Find(y)
  parent[a] = b
```



Wywołanie Union(7,3) spowoduje znalezienie i połączenie ze sobą reprezentantów: 9 i 2.

Łatwo widać, że w ten sposób nigdy nie powstanie cykl. Implementacja ta wygląda na sprytniejszą, ale wciąż może się zdarzyć niekorzystny ciąg operacji, który spowoduje powstanie wysokiego drzewa, na którym operacja *Find()* będzie bardzo czasochłonna.

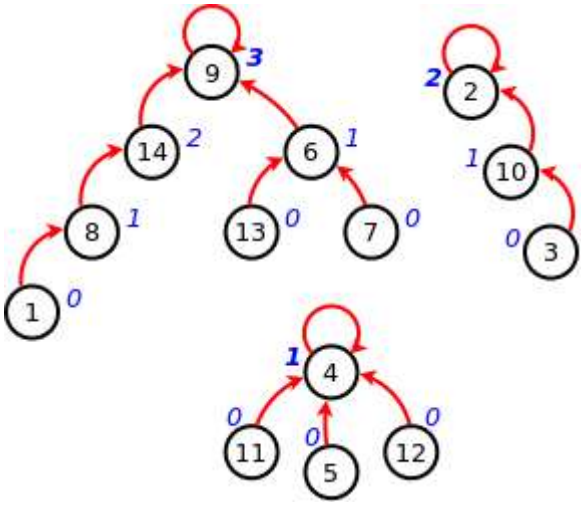


W tej sytuacji, każde wywołanie Find(1) przeszuka całe wysokie drzewo.

Na szczęście, istnieją dwie możliwości bardzo łatwej tej implementacji, aby *Find()* i *Union()* działały szybko. Są to, odpowiednio, **łączenie według rangi** i **kompresja ścieżek.**

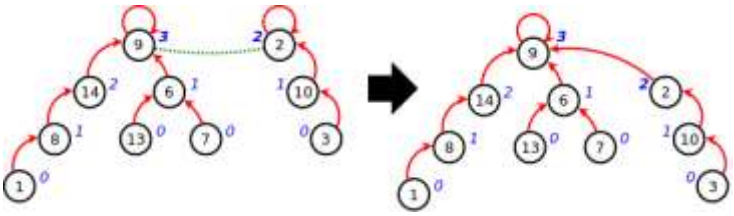
Rangi elementów

Każdemu elementowi x przypisujemy jego **rangę**, oznaczaną *rank[x]*. Ranga określa wysokość drzewa zaczepionego w elemencie x, czyli najdłuższą możliwą ścieżkę od pewnego innego elementu do x. Oznacza to, że operacja *Find(x)* wykona *rank[x]* operacji.

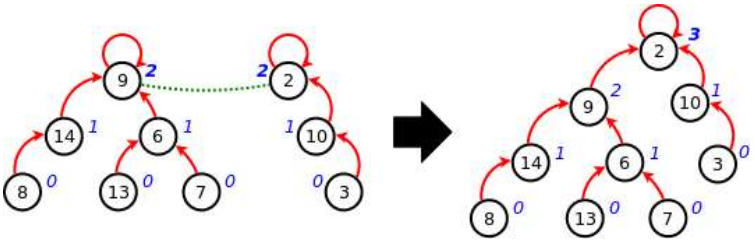


Niebieskie cyfry oznaczają rangę elementu i zarazem jego wysokość. Najbardziej istotne są rangi reprezentantów, zaznaczone pogrubionymi cyframi.

W praktyce rangi są istotne tylko dla elementów będących reprezentantami grup, określają bowiem kierunek ich łączenia. Kiedy łączymy ze sobą dwóch reprezentantów grup o różnych rangach, podczepiamy zawsze niższy do wyższego. W ten sposób nie trzeba zmieniać rangi żadnego z nich — wysokość drzewa nie zmieniła się:



Jeśli jednak łączone są dwa elementy o tej samej randze, wysokość jednego z nich zwiększa się o jeden. Konieczne jest zatem również zwiększenie rangi:



Pseudo-kod operacji Union z łączeniem według rangi wygląda zatem tak:

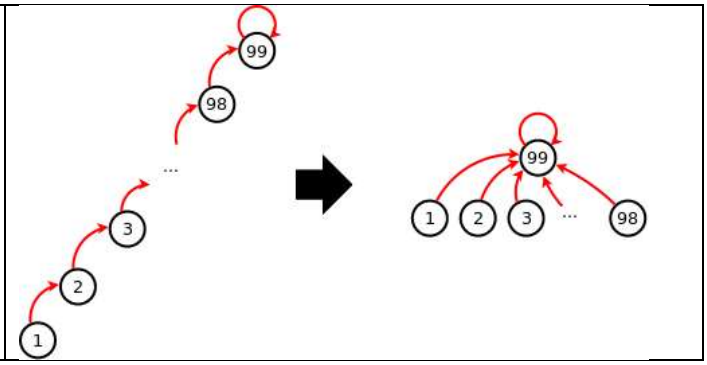
```
Union(x,y)
  a = Find(x)
  b = Find(y)
  if rank[a]<rank[b]
    parent[a] = b
  else
    parent[b] = a;
  if rank[a]=rank[b]
    rank[a] = rank[a]+1
```

Na początku wszystkie elementy mają rangę 0. Zauważmy, że aby jakiś element otrzymał rangę 1, muszą się ze sobą połączyć dwa elementy. Aby jednak element mógł otrzymać rangę 2, konieczne jest połączenie ze sobą dwóch grup z rangą 1, a zatem wynikowa grupa będzie liczyła co najmniej 4 elementy. Idąc dalej, element o randze 3 może się pojawić tylko w grupie o wielkości 8, element o randze 4 w grupie o wielkości 16, i tak dalej… Łatwo widać, że element o randze k zawsze jest w grupie o co najmniej $2k$ elementach. To oznacza, że ranga żadnego elementu nie może przekroczyć $\log n$, gdzie n jest liczbą elementów. Ale wiemy już, że *Find()* wykonuje tyle operacji, ile wynosi ranga pewnego elementu. Zatem jeśli użyjemy łączenia według rang, złożoność operacji *Find()* wynosi $O(\log n)$. Operacja *Union()*, która składa się z dwóch wywołań Find() i jednego lub dwóch przypisań, będzie oczywiście miała tę samą asymptotyczną złożoność.

Kompresja ścieżek

Jest to zaskakująco krótka poprawka operacji *Find()*, oparta na prostym pomyśle: jeśli raz już odkryliśmy, że reprezentantem elementu x jest inny element a, możemy równie dobrze ustawić *parent[x] = a*, aby skrócić wszystkie następne przeszukania. Bardzo łatwo połączyć to z wersją rekurencyjną procedury *Find()*:

```
Find(x)
  if parent[x] <> x
    parent[x] = Find(parent[x])
  return parent[x]
```



Zwróćmy uwagę, że dzięki rekursji stanie się nawet więcej: **wszystkie** elementy na ścieżce od x zostaną podłączone do reprezentanta. "Złośliwe" wysokie drzewo takie, jak na jednym z poprzednich rysunków, po jednym wywołaniu *Find()* zmienia się na w bardzo korzystny układ:

Złożoność

Jaka jest złożoność wywołań Find() i Union(), jeśli użyjemy kompresji ścieżek, na razie zapominając o rangach? Okazuje się, że jeśli wykonamy je k razy, łączna liczba operacji zsumuje się do $k \log n$, dla absolutnie dowolnej liczby k. Może się wprawdzie zdarzyć, że niektóre wywołania "po drodze" mogą zajmą więcej czasu, ale po ostatecznym podliczeniu algorytm zawsze zachowuje się tak, jakby **każdy** *Find()* i każdy *Union()* działał w czasie logarytmicznym. Mówimy, że **amortyzowana złożoność** tych operacji wynosi $O(\log n)$. Dowód tego faktu jest jednak dość długi i wymaga zaawansowanych technik.

Nic nie stoi jednak na przeszkodzie, aby użyć łącznie zarówno rang, jak i kompresji[1]. Oszacowanie złożoności robi się wtedy jeszcze trudniejsze, ale wynik jest zaskakujący.

W 1973 roku John Hopcroft i Jeffrey Ullman udowodnili, że amortyzowana złożoność operacji *Find()* i *Union()* w tym wypadku nie przekracza $O(\log^* n)$, gdzie $\log^* n$ (**logarytm iterowany**) oznacza liczbę razy, jaką trzeba zlogarytmować n, aby otrzymać 1 (na przykład $\log^* 4=2$, $\log^* 16=3$). Najmniejszą liczbą n dla której $\log^* n=5$, jest 2^{65536} — liczba zupełnie abstrakcyjna w kontekście praktyki. Dane, dla których $\log^* n$ byłoby zauważalnie duże, nigdy nie zdarzą się w prawdziwym świecie.

Oszacowanie Hopcrofta i Ullmana okazało się jednak jeszcze zbyt grube. Co jest nawet bardziej zaskakujące, właściwa złożoność jest jeszcze niższa, proporcjonalna do funkcji $\alpha(n)$, odwrotności tzw. funkcji Ackermanna. Dociekliwego czytelnika odsyłamy do źródeł — wynik ten pokazał Robert Endre Tarjan w roku 1975.

Kompresja ścieżek odrobinę "psuje" pojęcie rangi — nie oznacza ona już wysokości drzewa. Ranga pozostaje jednak większa lub równa wysokości, co wystarczy do naszych celów.