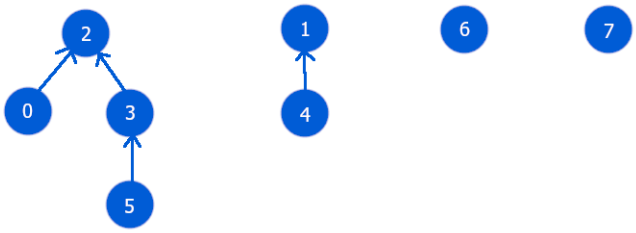
Zbiory rozłączne

#zbiory_rozłączne
#find_union
#funkcja_Ackermana

Wyobraźmy sobie zbiór pewnych elementów (na przykład liczb), który podzielony jest na rozłączne niepuste podzbiory. Taki zbiór będziemy określać mianem *uniwersum*, by zachować zgodność z dostępną literaturą na ten temat. Na tym uniwersum będziemy wykonywać dwa rodzaje operacji: zapytania, czy wskazane dwa elementy należą do tego samego podzbioru (ang. **find**, wymawiaj: *fajnd*), oraz łączenie dwóch podzbiorów (ang. **union**, wymawiaj: *junion*). Ważne jest, aby te zapytania (**find**) nie zmieniały przynależności elementu do danego podzbioru, no i pasowałoby, aby oba typy operacji były wykonywane jak najszybciej.

Każdy podzbiór naszego uniwersum powinien mieć swojego *reprezentanta*, czyli wyróżniony element. Jeśli dla dwóch dowolnie wybranych elementów okaże się, że należą do pozdbiorów z tym samym reprezentantem, będzie to oznaczać, że należą do tego samego podzbioru, gdyż podzbiory uniwersum są z definicji rozłączne, a jego elementy nie mogą się powtarzać.

Załóżmy, że nasze uniwersum to zbiór $\{0, 1, 2, 3, 4, 5, 6, 7\}$ i zastaliśmy następujący podział na podzbiory: $\{0, 2, 3, 5\}$, $\{1, 4\}$, $\{7\}$, $\{6\}$. Jednym ze sposobów zapisania tego podziału jest użycie rozłącznych drzew, w których każdy element „pamięta” swego rodzica (ang. **parent**) – z wyjątkiem tych elementów, które stanowią korzeń drzewa (ang. **root**, wymawiaj: *rut*, z długim *u*).* Oto przykładowa realizacja tych podzbiorów (strzałki pokazują kierunek od syna do rodzica):



W tym przykładzie reprezentantem dla pierwszego podzbioru jest element 2, dla drugiego – element 1, a w pozostałych podzbiorach jedyny należący do nich element jest swoim własnym reprezentantem.

Załóżmy, że informacje o rodzicu przechowujemy w wektorze (lub tablicy) *parent*. Jeśli brak rodzica (charakterystyczne dla reprezentanta) oznaczymy wartością -1, wtedy wektor ten powinien mieć postać:

2	–1	–1	2	1	3	–1	–1
0	1	2	3	4	5	6	7

*Drzewa w informatyce są typowymi drzewami australijskimi, rosną bowiem zielonym do dołu, a korzeniem do góry.

Wyszukiwanie reprezentanta dla zbioru z elementem a może przeprowadzić następująca rekurencyjna funkcja `find_set()`:

```
int find_set(vector<int> &parent, int a)
{
  if(parent[a] == -1) return a;
  return find_set(parent[a]);
}
```

Czyli jeśli element a nie jest reprezentantem, wtedy odwołujemy się do jego rodzica – i tak aż do skutku. Rekurencja na pewno zakończy się, ponieważ idąc po strzałkach do góry na pewno trafimy na element, który nie ma rodzica.

Jeśli zatem zapytamy dla przykładu o reprezentantów dla elementów 0 oraz 5, w obu przypadkach otrzymamy odpowiedź 2, co oznacza, że oba elementy należą do tego samego zbioru. Jeśli natomiast zapytamy o reprezentantów dla elementów 1 oraz 7, orzymamy różne wyniki (odpowiednio: 1 oraz 7), zatem te elementy znajdują się w oddzielnych zbiorach.

No dobrze, znajdowanie reprezentantów jakoś nam poszło, ale co z operacją **union**, czyli łączeniem zbiorów? Naszkicujemy funkcję `union_set()`, która zajmie się tym zadaniem. Załóżmy, że ma ona połączyć dwa podzbiory: do jednego z nich należy element a, a do drugiego element b:

```
void union_set(vector<int> &parent, int a, int b)
{
  . . .
}
```

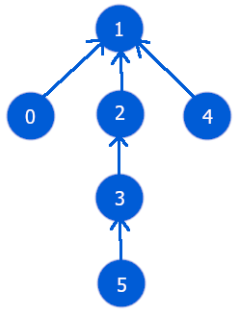
Dość oczywiste jest, że musimy wykonać pewną operację nie tyle na samych elementach a oraz b, ile na reprezentantach zbiorów, do które one należą – zaczniemy zatem od ich znalezienia:

```
int a_root = find_set(parent, a),
    b_root = find_set(parent, b);
```

Jeśliby przez przypadek okazało się, że ci reprezentanci to jeden i ten sam element, wtedy nie wykonujemy żadnej operacji – w przeciwnym razie „podpinamy” jeden zbiór pod drugi:

```
if(a_root != b_root)
  parent[a_root] = b_root;
```

Tym samym element `b_root` staje się reprezentantem również dla zbioru reprezentowanego do tej pory przez element `a_root`. Jeśli dajmy na to $a = 5$, zaś $b = 4$, to w naszym przykładzie dwa pierwsze zbiory połączą się w następujący sposób:

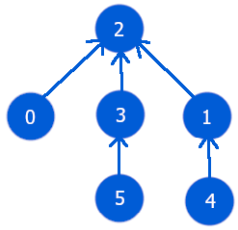


Z punktu widzenia logiki wszystko jest OK, jednak uważny Czytelnik na pewno spostrzeże, że mamy pewien efekt uboczny tego połączenia. Otóż obecnie długość ścieżki od elementu 5 do reprezentanta wydłużyła się – wynosi teraz 3 (tyleż strzałek do przebycia), a wcześniej miała długość 2. Przy większej ilości operacji może to prowadzić do drzew o niezrównoważonej budowie, a więc nieefektywnych w zastosowaniach. Wszakże z punktu widzenia operacji **find** im krótsze ścieżki, tym lepsze.

Zauważmy jednak, że gdybyśmy podpięli zbiory odwrotnie, dostalibyśmy lepszy rezultat. W naszym przykładzie operacja:

```
parent[b_root] = a_root;
```

prowadzi do następującego, bardziej efektywnego połączenia (żadna ścieżka nie ma długości większej niż 2):



Z kronikarskiego zamieścimy jeszcze pełny kod tej pierwszej implementacji operacji **union** (ale już za chwilę zajmiemy się jej podrasowaniem):

```
void union_set(vector<int> &parent, int a, int b)
{
  int a_root = find_set(parent, a),
      b_root = find_set(parent, b);
  if(a_root != b_root)
    parent[a_root] = b_root;
}
```

Wracając do kolejności łączenia: nie jest obojętne, co się do czego podpina.[†] Generalnie lepiej jest podpiąć drzewo o krótszych gałęziach (ścieżkach) do korzenia drzewa większego, bo wtedy nie powiększamy najdłuższej ścieżki. Jeśli oba drzewa mają tę samą wysokość, no to wtedy trudno: podpinamy jak wypadnie, z tym że nie powinno mieć to wpływu na finalną efektywność (tak czy owak najdłuższa ścieżka ulegnie przedłużeniu o 1).

[†]Każdy elektryk to wie...

Taki sposób łączenia drzew nazywamy łączeniem *według rangi*, gdzie słowo „ranga” (ang. **rank**) oznacza długość najdłuższej ścieżki w danym drzewie. Wymaga to dodatkowego wektora (tablicy), nazwiemy go po prostu *rank*.[‡] W tym wektorze będziemy przechowywać rangi dla każdego reprezentanta (przyjmujemy, że dla zbioru jednoelementowego ranga wynosi 0). Zauważmy, że wartość rangi dla elementu niebędącego reprezentantem jest bez znaczenia, ponieważ zbiór reprezentantów będzie się raczej kurczył w wyniku łączenia zbiorów. Zatem wystarczy uaktualniać tylko rangi obecnych reprezentantów.

Nowa wersja funkcji `union_set()` powinna wyglądać tak:

```
void union_set(vector<int> &parent, vector<int> &rank, int a, int b)
{
  int a_root = find_set(parent, a),
      b_root = find_set(parent, b);
  if(a_root == b_root) return;
  if(rank[a_root] > rank[b_root])
    parent[b_root] = a_root;
  else if(rank[a_root] < rank[b_root])
    parent[a_root] = b_root;
  else
  {
    parent[b_root] = a_root;
    rank[a_root]++;
  }
}
```

Potrzebujemy jeszcze funkcji, która inicjalizuje wektory *parent* oraz *rank* – dzieje się to na początku działania programu, gdy wszystkie zbiory są jednoelementowe (każdy element jest swoim własnym reprezentantem). Zwyczajowo nazywa się tę funkcję `make_set()`:

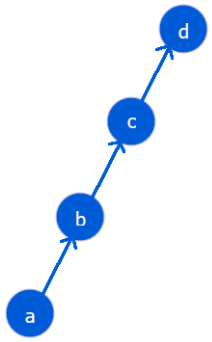
```
void make_set(vector<int> &parent, vector<int> &rank)
{
  fill(parent.begin(), parent.end(), -1);
  fill(rank.begin(), rank.end(), 0);
}
```

Zrobiliśmy już spory krok naprzód, ale możemy pójść znacznie dalej, i to bardzo niskim kosztem. Przyjrzyjmy się dla odmiany funkcji `find_set()`, czy czasem nie potrafimy jej *dobajerzyć*. Mamy tam odwołanie rekurencyjne do potomka tej funkcji, który sięga do rodzica elementu a, kolejny potomek zajmuje się dziadkiem, następny – pradziadkiem, aż w końcu docieramy do reprezentanta. Nic nie stoi na przeszkodzie, aby podczas tego spaceru po pokoleniach przepinać kolejne elementy coraz wyżej, aż do samego reprezentanta. Przyjrzyjmy się, jak to działa:

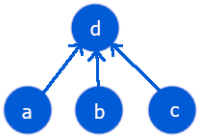
```
int find_set(vector<int> &parent, int a)
{
  if(parent[a] == -1) return a;
  parent[a] = find_set(parent[a]);
  return parent[a];
}
```

[‡]W języku C++ istnieje już klasa o tej nazwie, ale nie będzie nam to zbytnio przeszkadzać.

Jeśli tak zorganizujemy tę funkcję, to w ostatecznym rachunku rodzicem elementu a stanie się reprezentant zbioru, do którego należy ten element – i to samo stanie się z wszystkim elementami na ścieżce do góry (!). Zatem taka przykładowa ścieżka:



zamieni się w coś takiego:



Fachowo nazywa się to *kompresją ścieżki* i wpływa dramatycznie na szybkość operacji na zbiorach rozłącznych. Każde zapytanie **find** powoduje uporządkowanie jednej ścieżki, a większa ilość tych zapytań skutkuje tym, iż kolejne z nich będą wykonywać się w czasie stałym. Jeśli policzy się łączną złożoność tego algorytmu, wyraża się ona przez iloczyn $m \cdot \alpha(m, n)$, gdzie n jest ilością elementów w naszym uniwersum, zaś m jest ilością zapytań **find**. Funkcja $\alpha()$ jest to odwrotna funkcja Ackermana – niezwykle wolno rosnąca funkcja. Dość powiedzieć, że w tak zwanych normalnych warunkach jej wartość nie przekracza 6.[§]

Tak ulepszony algorytm będzie nam potrzebny na przykład przy wyznaczaniu minimalnego drzewa rozpinającego (MST), czemu poświęcony będzie jeden z następnych podrozdziałów.

Jeszcze jeden komentarz: nasze uniwersum zostało podzielone na pewną ilość drzew – a jak nazywa się zbiór drzew? Tak, dobrze myślicie: to *las* (ang. **forest**).

[§]Oszacowanie złożoności tego algorytmu zostało podane przez Roberta Tarjana (wym. *Tardżana*) w 1984 roku, 20 lat po jego ogłoszeniu przez Gallera i Fishera. (Do tego nazwiska jeszcze kiedyś wrócimy.)