Union Find Tree(素集合データ構造)

Union Find Treeとは

- 単にUnion Findと呼ばれることも
- グループ分けを管理するデータ構造
- 素集合: 重なりの無い(互いに素な)集合たち.
- 次の操作がほぼ定数時間で行える
    - **グループを併合する(Unite)**
      ※併合はできても分割できない
    - **ある要素とある要素が同じグループか(Same)**
- **グラフの連結に関する問題を扱うときによく使う**

まずはこの問題を考えてみる

- ABC062A: Grouping
- 問題:
  x,yが与えられるのでx,yが同じグループかどうか判定せよ
  - {1, 3, 5, 7, 8, 10, 12}は同じグループ
  - {4, 6, 9, 11}は同じグループ
  - {2}は同じグループ

- 愚直に全探索しても間に合う.
    i. グループの名前をそれぞれ0,1,2と決める.
        - グループ0: {1, 3, 5, 7, 8, 10, 12}
        - グループ1: {4, 6, 9, 11}
        - グループ2: {2}
    ii. xがどのグループに属しているか判定
    iii. yがどのグループに属しているか判定
    iv. グループ名が同じかどうか比較

- 愚直解

```
vector<vector<int>> g {
    {1, 3, 5, 7, 8, 10, 12},
    {4, 6, 9, 11},
    {2}
};
int find(int x)
{
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < g[i].size(); j++) {
      if (g[i][j] == x) return i;
    }
  }
}
int main()
{
  int x, y;
  cin >> x >> y;
  if (find(x) == find(y)) cout << "Yes" << endl;
  else cout << "No" << endl;
}
```

- vectorの文法については，「initializer list」でググって

別の見方(1)

- グループのリーダー(親)をなんでもいいので一人決める.
- 「〇〇さんの親は××」という情報を入れた配列を作る
- 親が一致していれば同じグループだと分かる.

(例):

- {1, 3, 5, 7, 8, 10, 12} の親は1
- {4, 6, 9, 11} の親は4
- {2} の親は2
  と決める

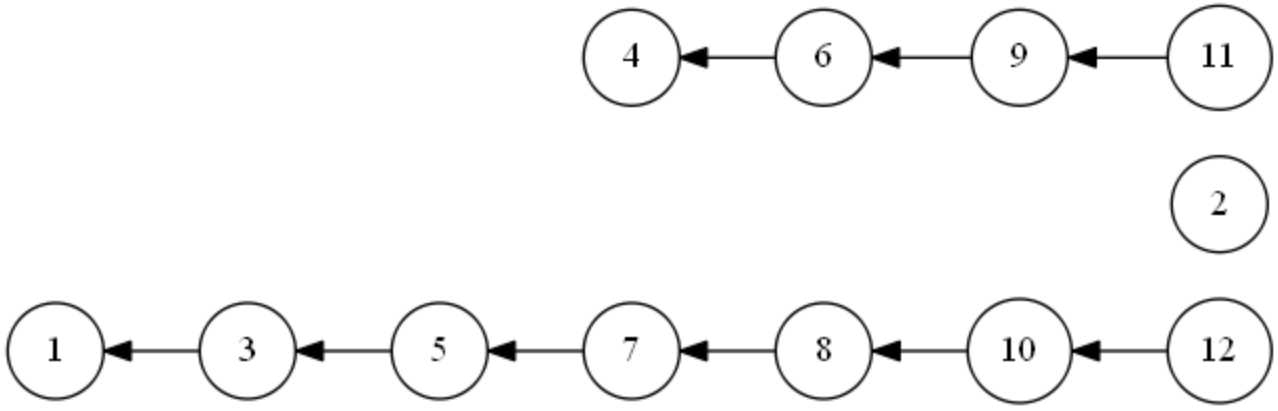
node	1	2	3	4	5	6	7	8	9	10	11	12
parent	1	2	1	4	1	4	1	1	4	1	4	1

```
int main()
{
  vector<int> par{0, 1, 2, 1, 4, 1, 4, 1, 1, 4, 1, 4, 1};
  int x, y;
  cin >> x >> y;
  if (par[x] == par[y]) cout << "Yes" << endl;
  else cout << "No" << endl;
}
```

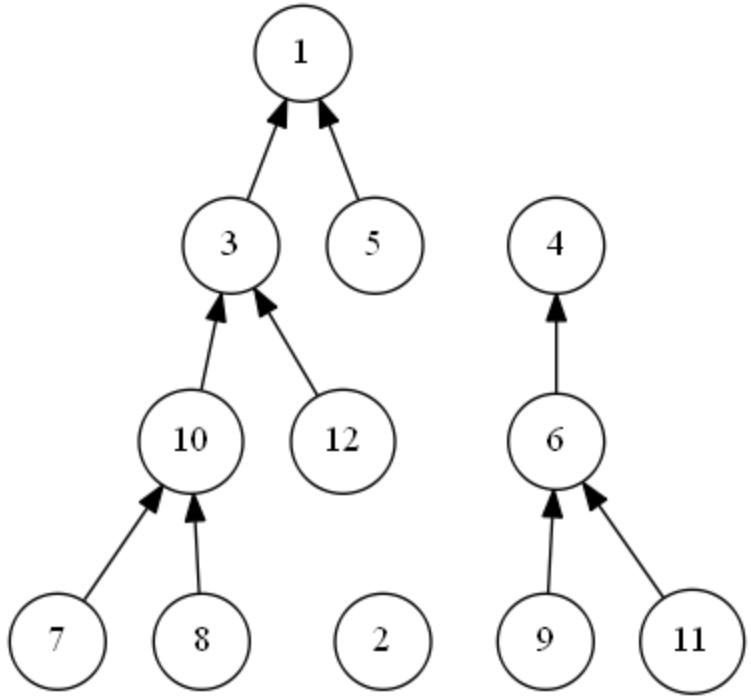
別の見方(2)

- 親子関係を表した配列を用意すると,親をどんどん辿って最後に行き着く「先祖」が一致していれば
  おなじグループに属している
- aはbの親であることをa←bと書くことにすると,
  1←3←5←7←8←10←12
- 12の先祖は1,7の先祖は1 ⇒ 12と7は同じグループ

- (親)←(子)で辺を貼り,グラフで表現できる



- こんな感じの親子関係でもよい
  (厳密にはこれは有向木の辺を逆向きに張ったverであるが,ここでは単に**木**と略記する.)
- グラフでいうと,「先祖」とは**根**のこと



まとめると

- グループは(親)→(子)で有向辺をつないだ木で表される
- グループのリーダー(祖先)とは,木の根のことである

以下グラフ理論の用語で説明する.

Union Find Treeに戻る

次の操作の基本的な考え方

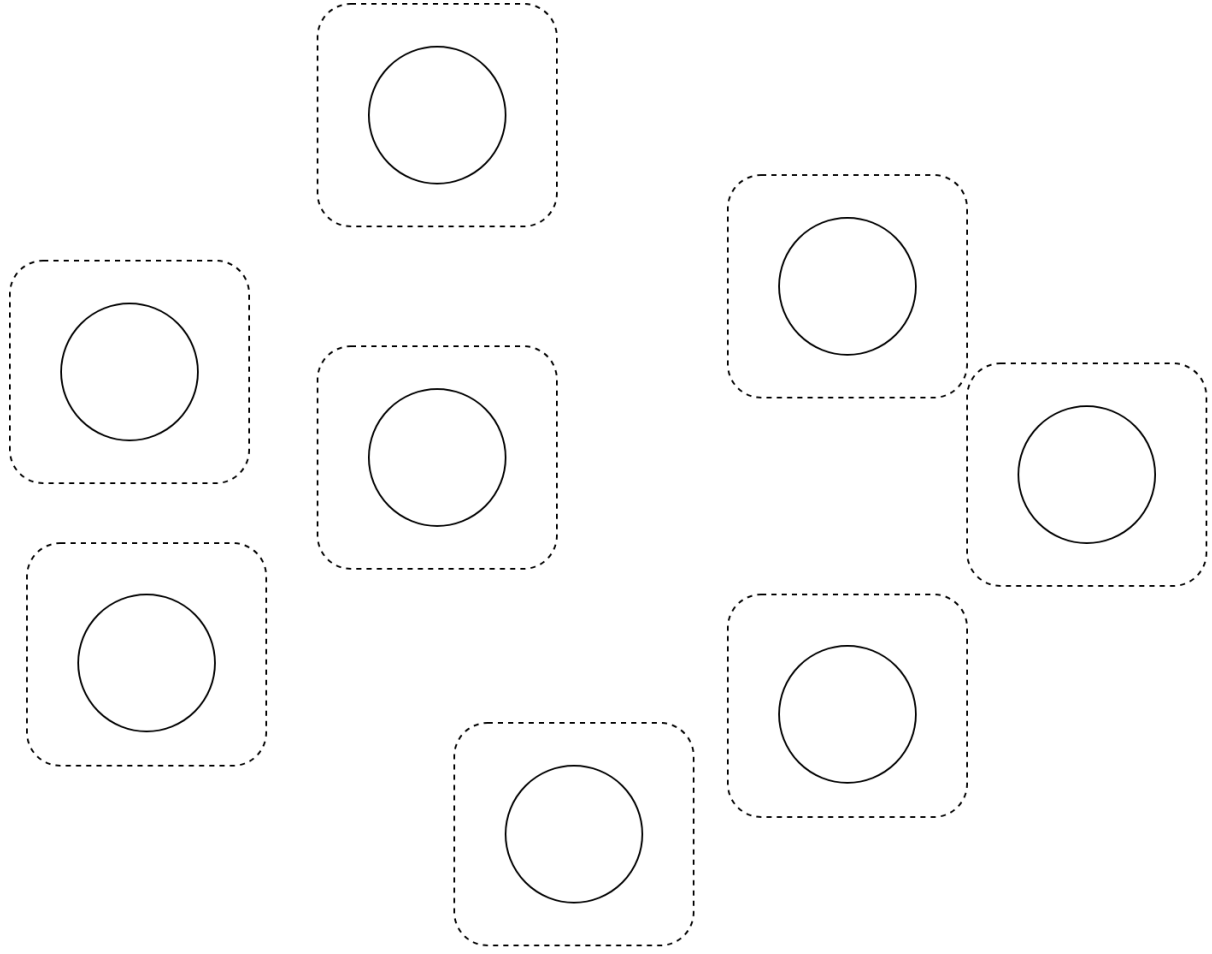
- xのグループとyのグループを併合(Unite)
  ⇒ xが属する木の根,yが属する木の根を求め,根同士に辺を張る
- xとyが同じグループか(Same) :
  ⇒ xが属する木の根,yが属する木の根を求め,それらが一致しているかどうか判定

これを実現するために,

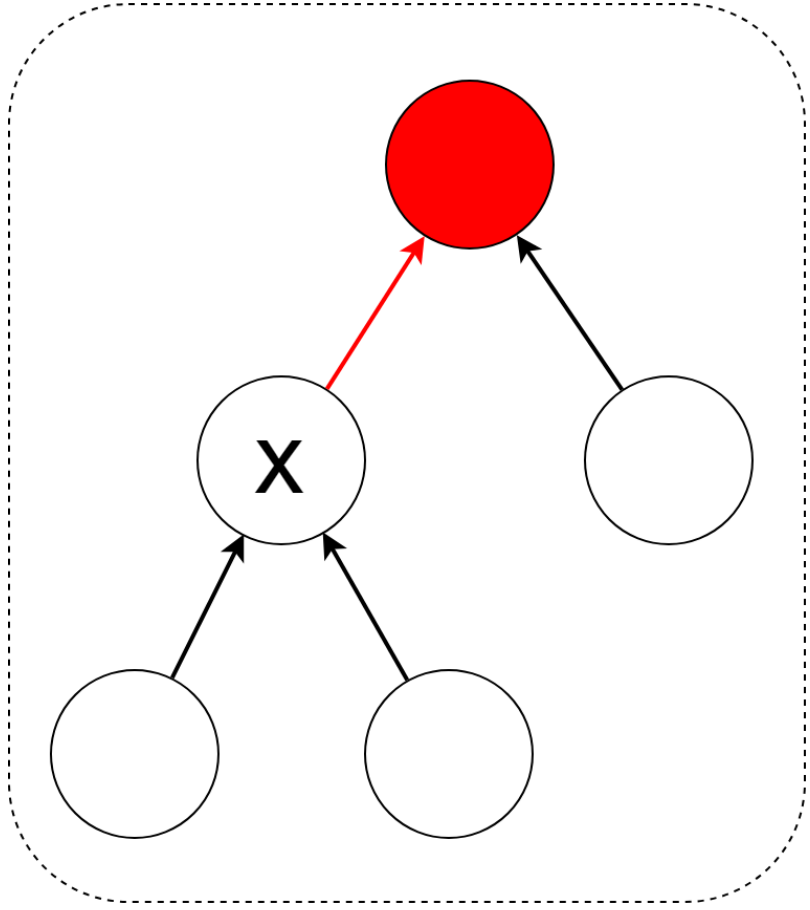
- xが属する木の根は何か(Find)

という関数を用意する.

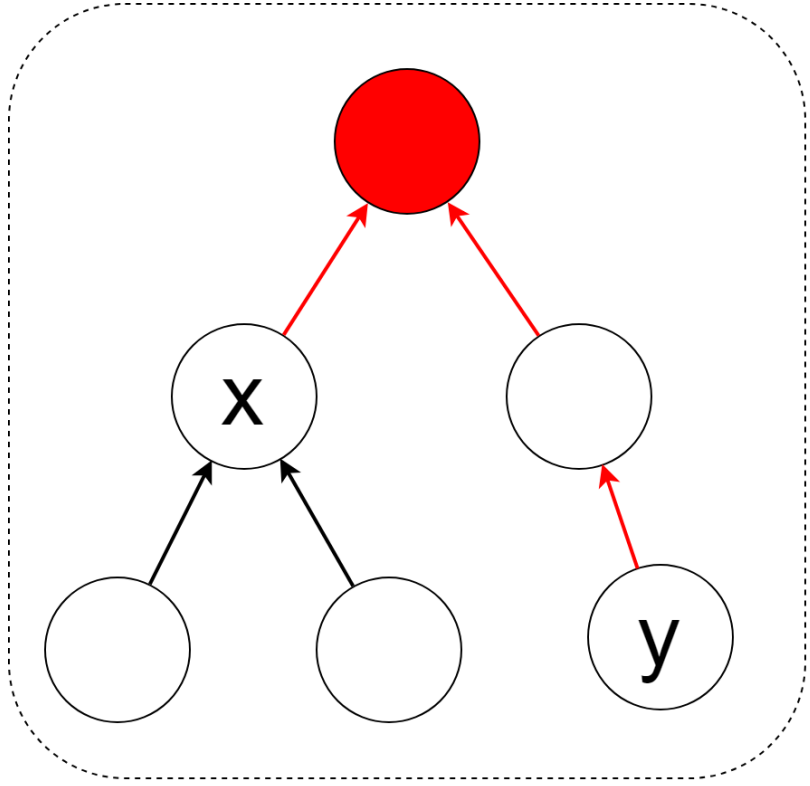
- Union Find Treeの初期状態
- みんな別々のグループ



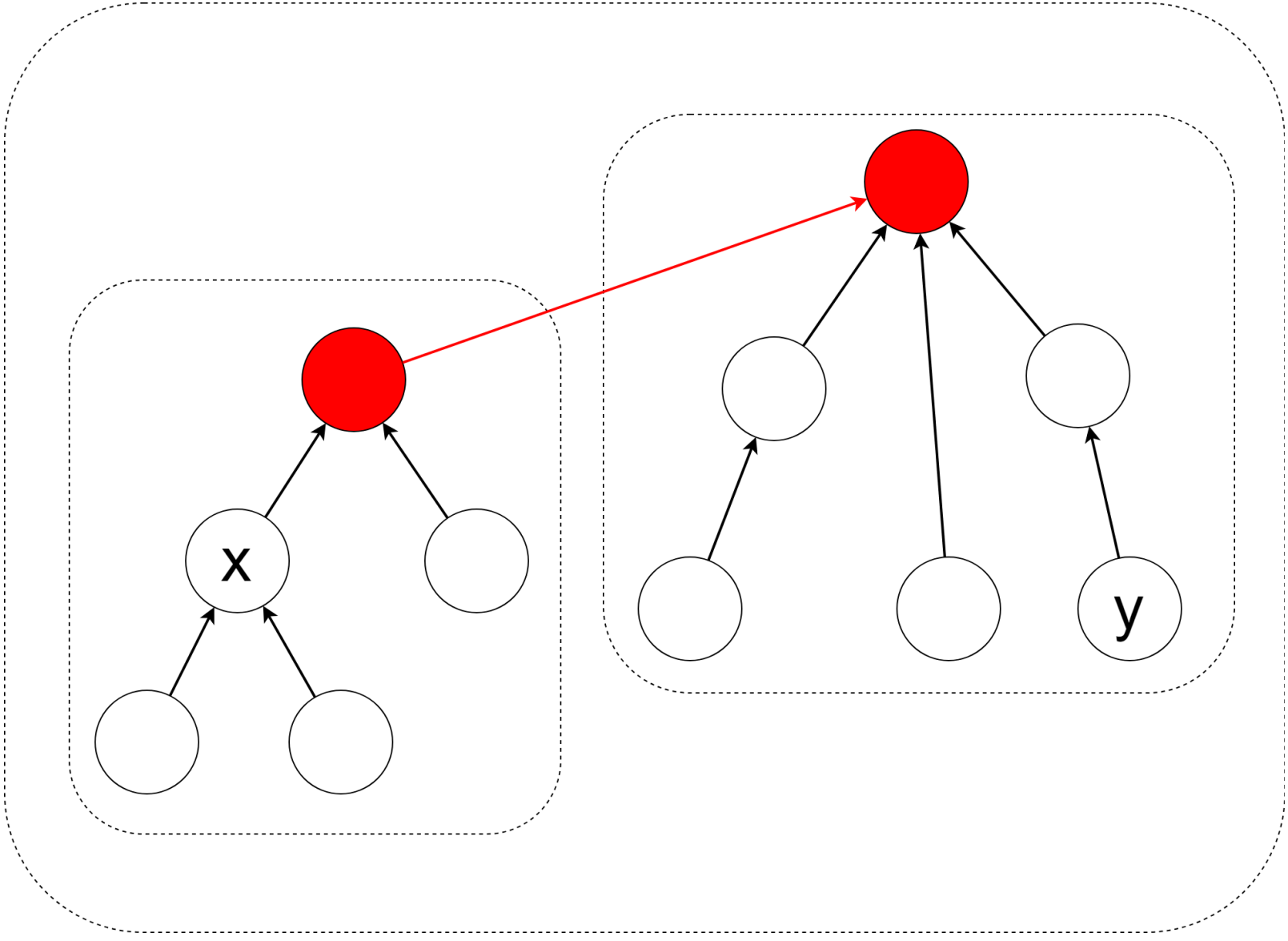
- Find



- Same



- Unite



実装

- 実装(1)(2)(3)の順で行い,徐々に計算量を改善する形で説明します.

実装(1)

- xの親を求める配列を用意する
- 初期化関数を用意する
- 初めは自分自身が親である

```
int par[110000]; //問題に応じてサイズを変える
void init(int n)
{
  for (int i = 0; i <= n; i++) {
    par[i] = i;
  }
}
```

- xが属する木の根を求める関数find(x)
  xの親は何か,その親は何か,...と再帰関数で潜っていく

```
int find(int x)
{
  if (par[x] == x) return x;
  return find(par[x]);
}
```

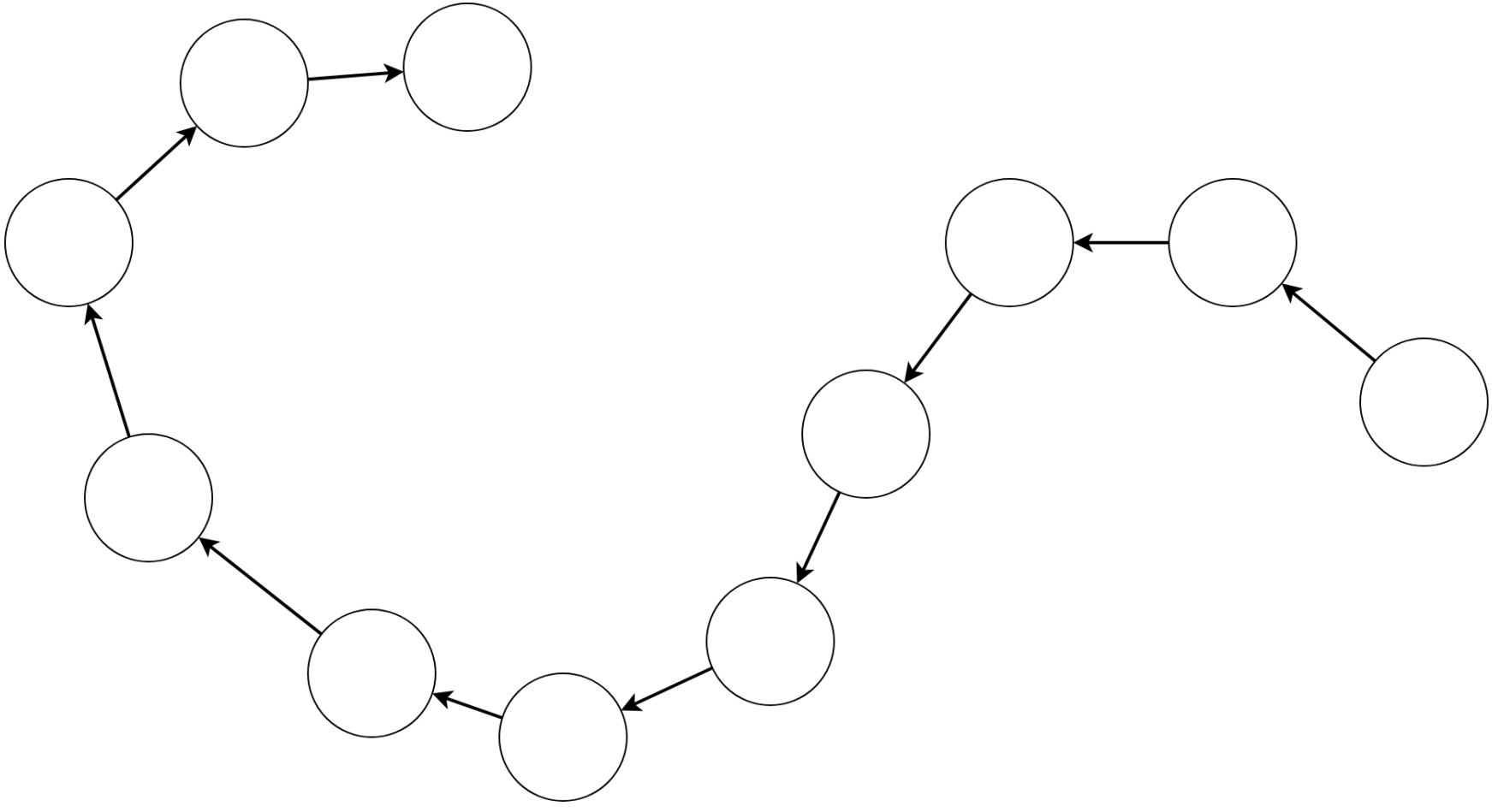
- xとyが同じグループかどうかを求める関数same(x, y)
  find(x) == find(y)を判定するだけ

```
bool same(int x, int y)
{
  return find(x) == find(y);
}
```

- xが属すグループとyが属すグループを併合する関数unite(x, y)
  find(x)とfinc(y)の親子関係を決める
- x,yどちらを親にするかは,今は考えないことにする

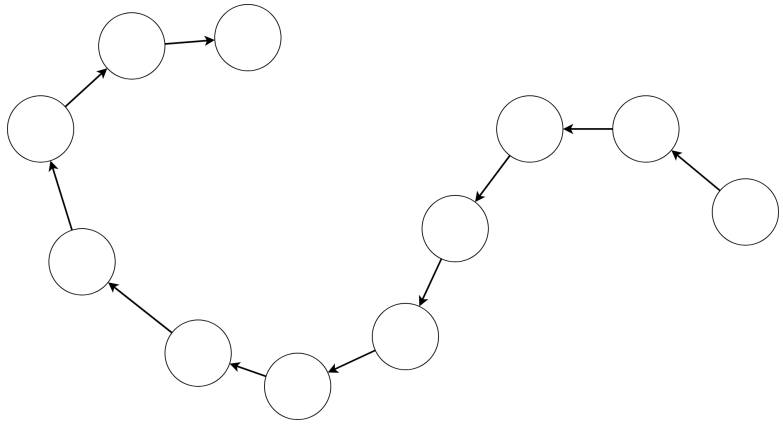
```
void unite(int x, int y)
{
  x = find(x);
  y = find(y);
  par[x] = y;
}
```

- 計算量は$O(|V|)$くらい.
  最悪なのは全ての点が一直線につながったケース
- もうちょい早くならないか

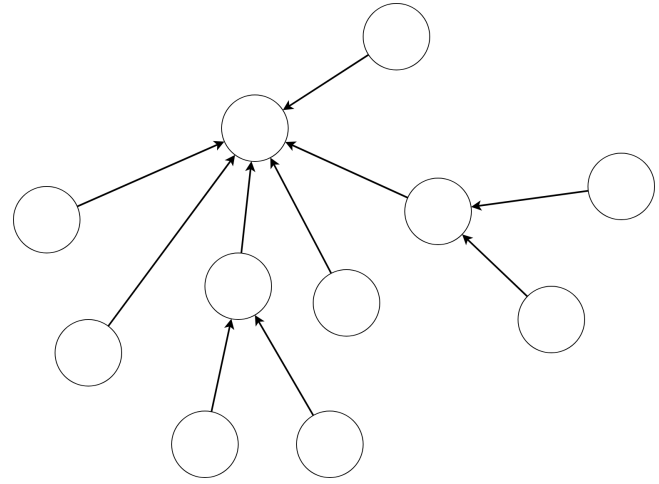


実装(2)

- 点が根から遠いと辿るのに時間がかかる
- できるだけ点は根に近い方が良い.



悪いケース



良いケース

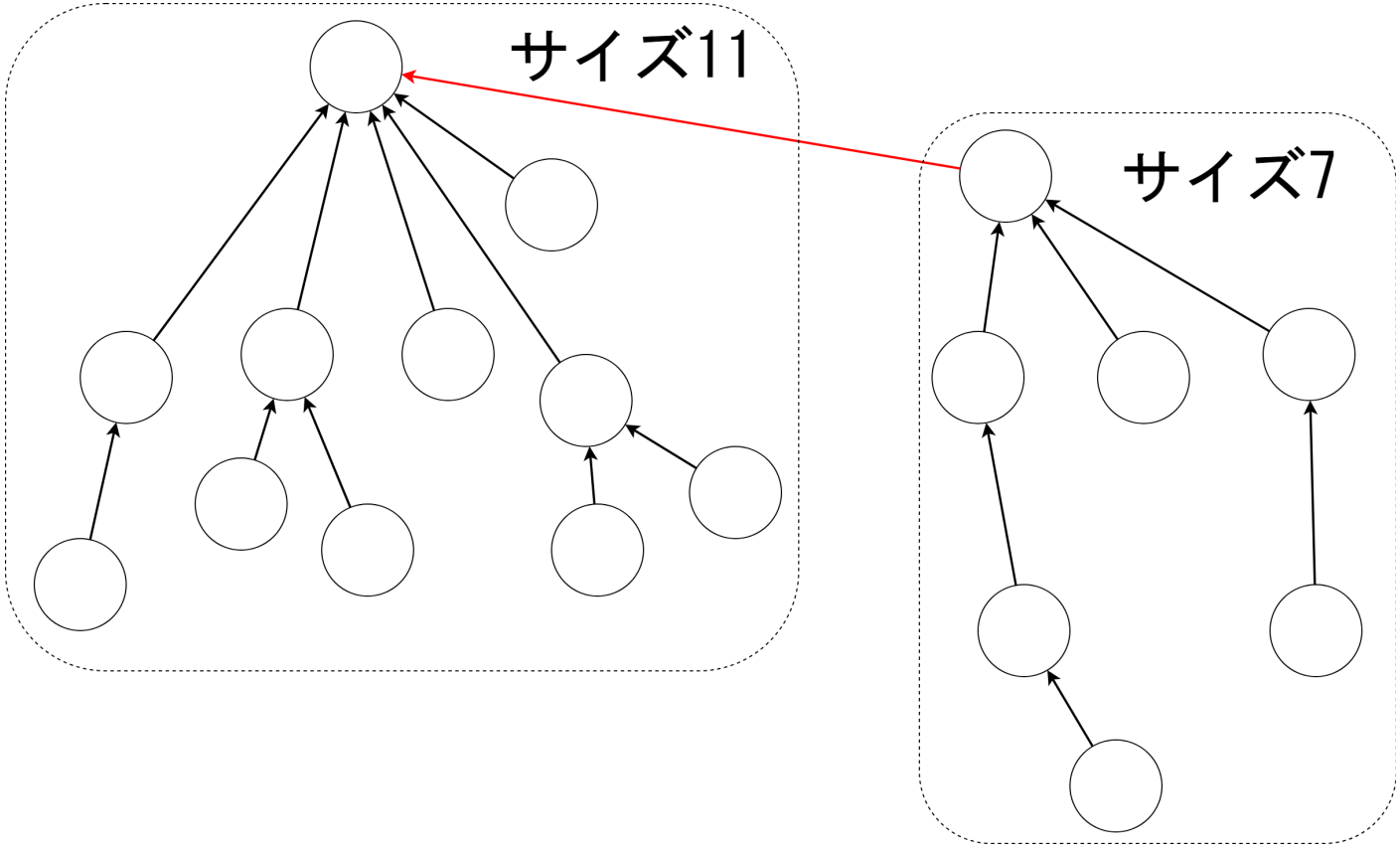
- なのでfind関数呼び出し時に,辿った点をすべて根に張り直して
  なるべく点を根に近くしよう(これを**経路圧縮**という)
- find関数の改善:returnの部分に注目

```
int find(int x)
{
  if (par[x] == x) return x;
  return par[x] = find(par[x]); // 返ってきた値を代入して返す
}
```

- この工夫で,**平均時間計算量**$O(log|V|)$であることが知られている
- これだけでも十分なことが多い
- でももっと速くできます

実装(3)

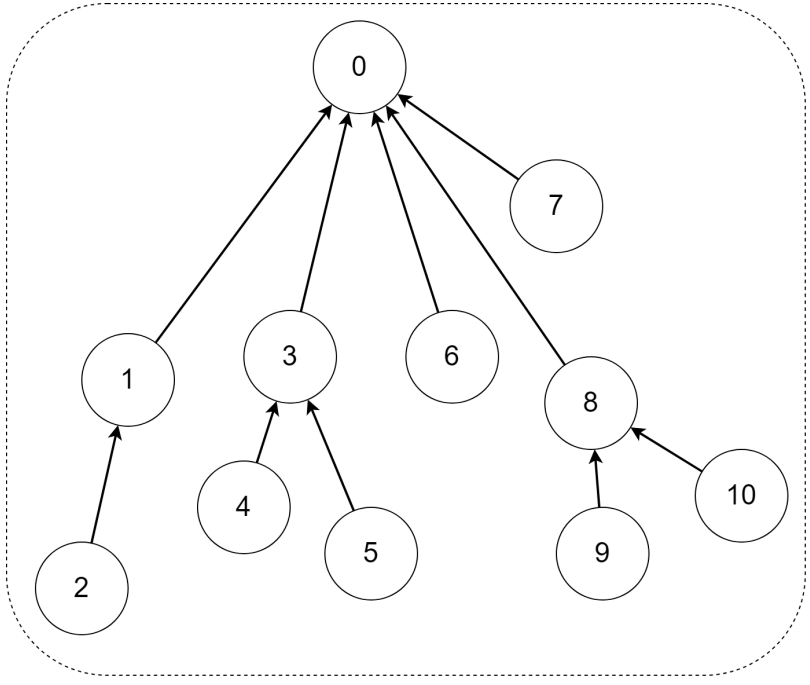
- 木のサイズ(点の数)の大きなほうから小さな方へつなげば,
  偏りが少なくなりそう



- 実はこれで計算量が改善することが知られている.
- では木のサイズをどう管理するか?

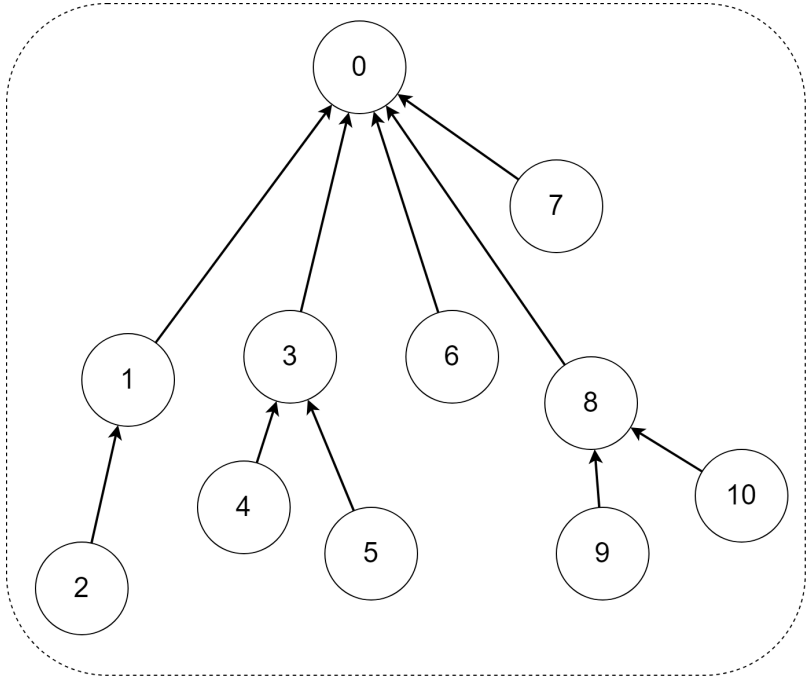
- 根の親は根自身だった
  ⇒無駄なので,ここにサイズの情報を付加しよう
- par[i]が表すのがサイズなのか親なのかは符号で判断するように実装する ⇒サイズは負,親は正

i	0	1	2	3	4	5	6	7	8	9	10
par[i]	0	0	1	0	3	3	0	0	0	8	8



- 根の親は根自身だった
  ⇒無駄なので,ここにサイズの情報を付加しよう
- par[i]が表すのがサイズなのか親なのかは符号で判断するように実装する ⇒サイズは負,親は正

i	0	1	2	3	4	5	6	7	8	9	10
par[i]	-11	0	1	0	3	3	0	0	0	8	8



- 初期化関数: 最初は自身が根,サイズは1なので,初期値に-1

```
int par[110000];
void init(int n)
{
  for (int i = 0; i < n; i++) {
    par[i] = -1;
  }
}
```

- find関数: par[i]が負なら親,そうでないなら再帰

```
int find(int x)
{
  if (par[x] < 0) return x;
  return par[x] = find(par[x]);
}
```

- unite関数: 辺のサイズを調べ,大きい方に小さいほうをくっつける

```
int unite(int x, int y)
{
  x = find(x);
  y = find(y);
  if (par[x] > par[y]) swap(x, y);
  par[x] += par[y]; // (xを含む木のサイズ) += (yを含む木のサイズ)
  par[y] = x;       // yの親をxに設定
}
```

- グラフのサイズを取得する関数sizeも作っておくと役立つことがある

```
int ufsize(int x)
{
  return -par[find(x)];
}
```

- 平均時間計算量は$O(\alpha(n))$となることが知られている
  $\alpha(n)$とはアッカーマン関数$Ack(n, n)$の逆関数
- アッカーマン関数は小さな変数で値がすごく大きくなる数
  それの逆関数なので,**ほぼ定数時間**だと思ってよい.すごい.

実装まとめ

```
int par[110000];
void init(int n) {
  for (int i = 0; i < n; i++) {
    par[i] = -1;
  }
}
int find(int x) {
  if (par[x] < 0) return x;
  return par[x] = find(par[x]);
}
int unite(int x, int y) {
  x = find(x);
  y = find(y);
  if (par[x] > par[y]) swap(x, y);
  par[x] += par[y];
  par[y] = x;
}
bool same(int x, int y) {
  return find(x) == find(y);
}
int ufsize(int x) {
  return -par[find(x)];
}
```

これを構造体にまとめる

次のやつをライブラリとして持ってると良いです

```cpp
struct UnionFind {
  vector<int> par;

  UnionFind(int n): par(n, -1) { }
  int find(int x) {
    if (par[x] < 0) return x;
    else return par[x] = find(par[x]);
  }
  void unite(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (par[x] > par[y]) swap(x, y);
    par[x] += par[y];
    par[y] = x;
  }
  bool same(int x, int y) {
    return find(x) == find(y);
  }
  int size(int x) {
    return -par[find(x)];
  }
};
```

使い方

```
int main()
{
  UnionFind uf(100); // 要素数100
  uf.unite(1, 2); // 1が属すグループと2が属すグループをくっつける
  uf.unite(1, 5); // 1が属すグループと5が属すグループをくっつける
  uf.unite(2, 3); // 2が属すグループと3が属すグループをくっつける

  // この時点で{1,2,3,5}が同じグループ

  // 1と3は同じグループ -> true(= 1)が返ってくる
  cout << uf.(1, 3) << endl;
  // 2と5は同じグループ -> true(= 1)が返ってくる
  cout << uf.(2, 5) << endl;
  // 1と90は同じグループでない -> false(= 0)が返ってくる
  cout << uf.(1, 90) << endl;
  // 1が属するグループのサイズである4が返ってくる
  cout << uf.size(2) << endl;
}
```

実行結果

```
1
1
0
4
```

その他の実装方法

- 今回は「サイズの大きい方に小さい方をくっつける」実装
  「木の深さ(**ランク**という)の大きい方に小さい方をくっつける」実装方法もある (蟻本の実装がそれ)
- 他にも色々な実装方法があるので気になる人は英語Wikipedia「Disjoint-set data structure」を参照
- 問題によってはUnion Findに様々な情報を乗せて応用することがある.
  - グリッドの場合は(x,y)を頂点番号としてpairで管理したりする
  - (余談)僕はABC120 DのときUnion By Rankしか持ってなかったので,そこにsizeを乗せた.そういうのもOK.

Union Find木の可視化(おまけ)

- 計算量改善の工夫のおかげで,多くの点が根に繋がる
  ⇒ Union Findは「うにオンファインド」である.
- デモしたかったけどできず

応用

- 頂点同士の細かいつながりは気にせず,
  あくまでグラフの連結成分についてみていきたいときに使われることが多い

演習

- ATC001B: Union Find
  (勉強のためできれば自前で実装してほしい)
  (どうしても分からなければ僕のやつをコピペして使い方だけは覚えましょう)
- ABC075C: Bridges
- ARC032B: 道路工事
- AB
- C097D: Equals

解答

- ATC001B: やるだけ
- ABC097D: グラフ描いて考察,つながっているとどうなのか
- ARC032B: 実はDFS/BFSでも解ける

やっておくとよい問題

- ABC120D: Dacayed Bridges