

Disjoint-set (Union-find)

Laboratório de Programação Competitiva - 2020

Pedro Henrique Paiola

Introdução

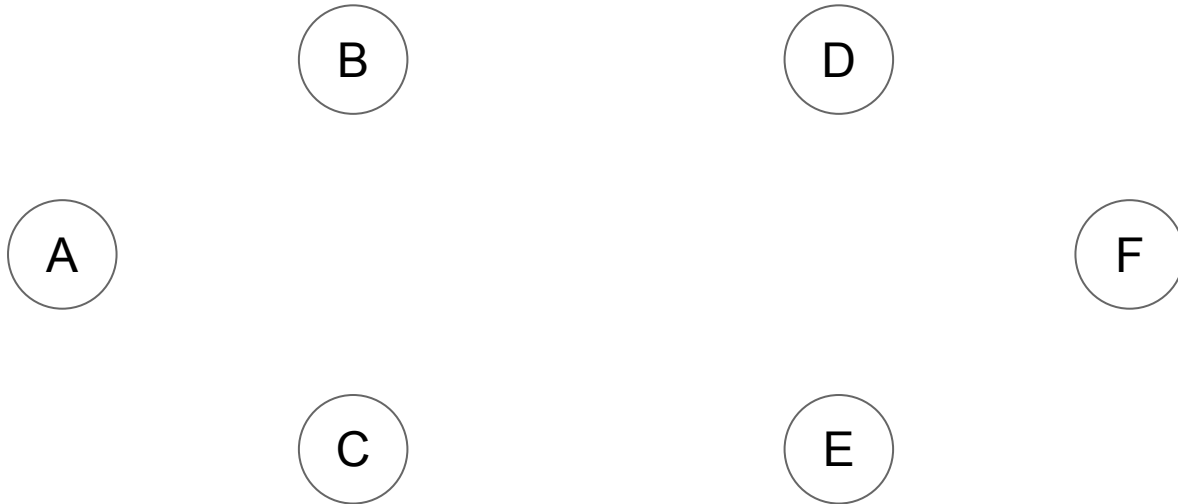
- **Disjoint Set Union (DSU)**, também chamada de **Union-find**, devido as operações que esta estrutura de dados permite
- Esta estrutura armazena vários conjuntos disjuntos de elementos
 - Inicialmente, cada conjunto contém precisamente um elemento

Introdução

- Permite a realização de duas operações:
 - **merge(a, b)**: une os conjuntos aos quais a e b pertencem
 - **find(a)**: determina a qual conjunto o elemento a pertence
- A partir do *find*, normalmente define-se a função *same*:
 - **same(a, b)**: determina se a e b pertencem ao mesmo conjunto

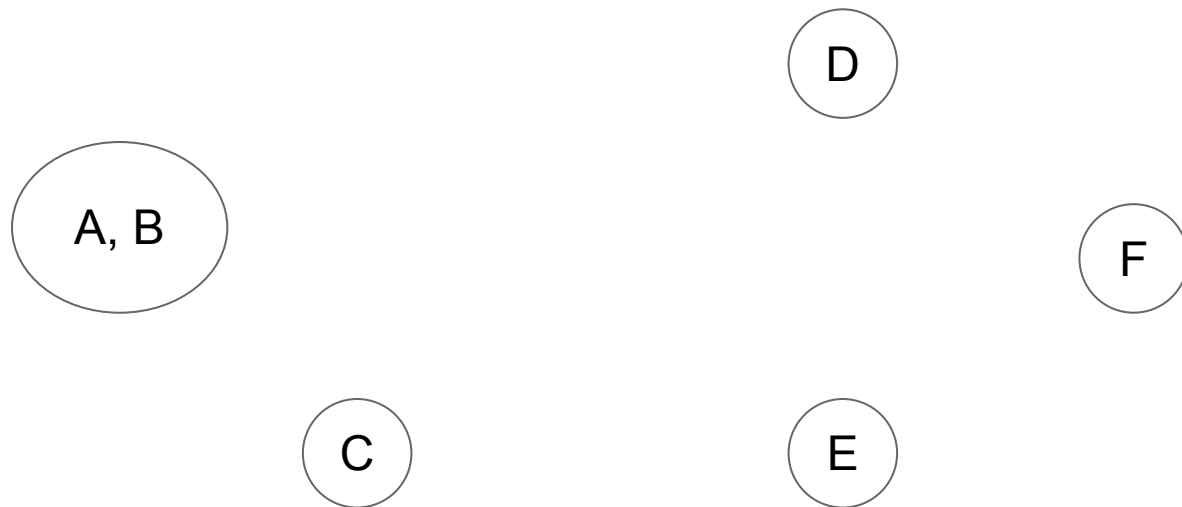
Exemplo

same(A, B) \Rightarrow False



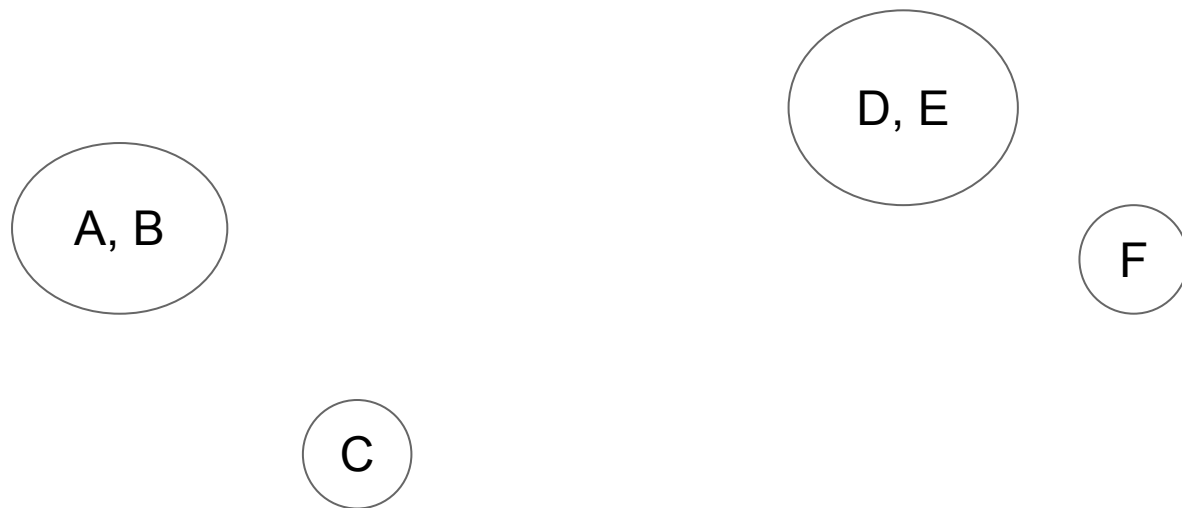
Exemplo

$\text{union}(A, B)$



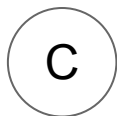
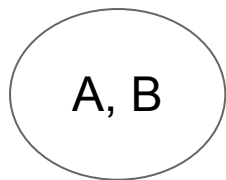
Exemplo

$\text{union}(D, E)$



Exemplo

$\text{union}(D, F)$

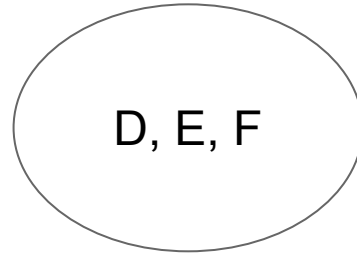
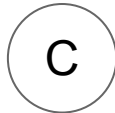
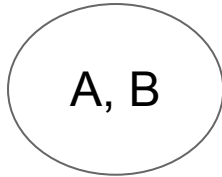


Exemplo

same(A, B) => True

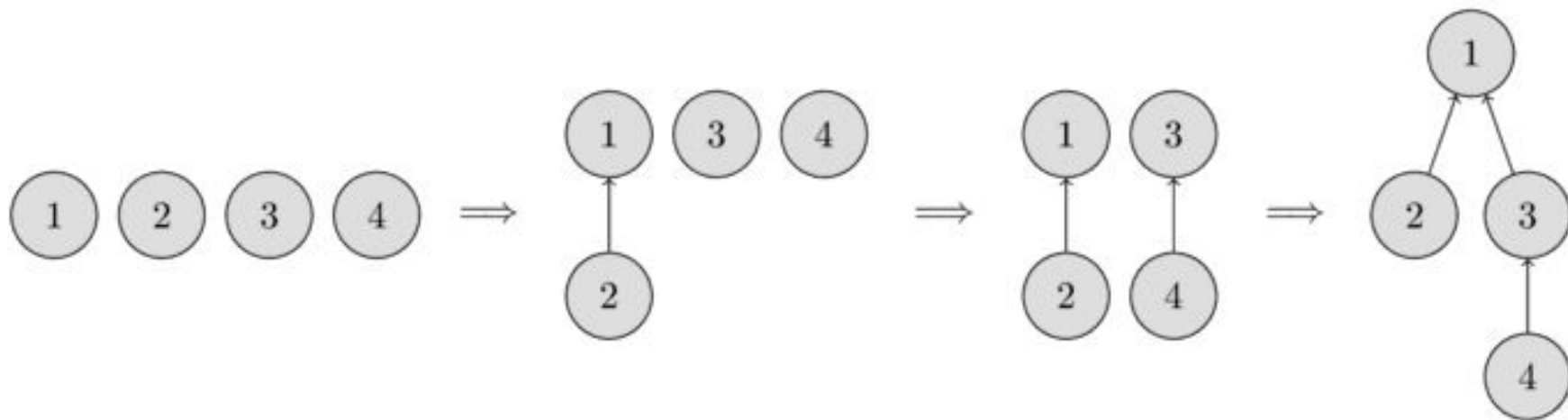
same(E, F) => True

same(A, D) => False



Estruturando a solução

- Para implementar uma DSU, cada conjunto será representado por uma **árvore**, onde a **raiz** da árvore será o representante/**líder** do conjunto.



Implementação (nem tão) ingênua

- **find(x)**: retorna o líder do conjunto a que x pertence
- **merge(x, y)**: conecta os líderes. Supondo que X seja líder de x e Y líder de y, vamos “eleger” Y como líder de X.

Implementação (nem tão) ingênua

```
int pai[MAX_N + 1];
```

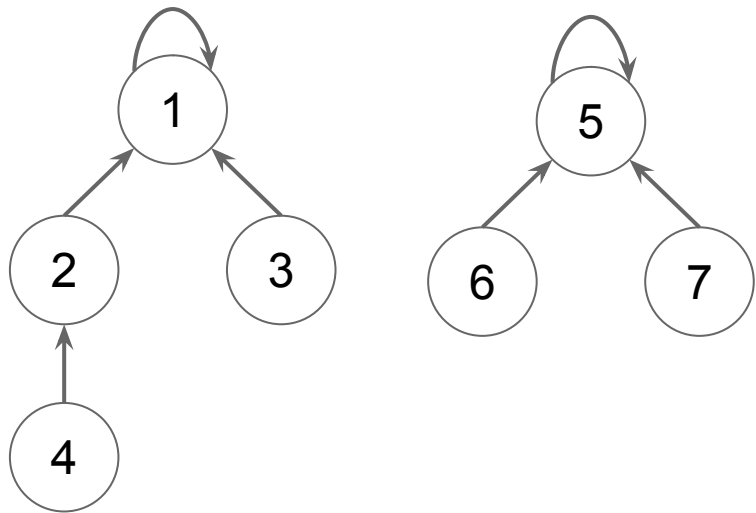
```
void init() {  
    for(int i = 1; i <= MAX_N; i++)  
        pai[i] = i;  
}
```

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

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

```
void merge(int u, int v) {  
    int a = find(u);  
    int b = find(v);  
    pai[a] = b;  
}
```

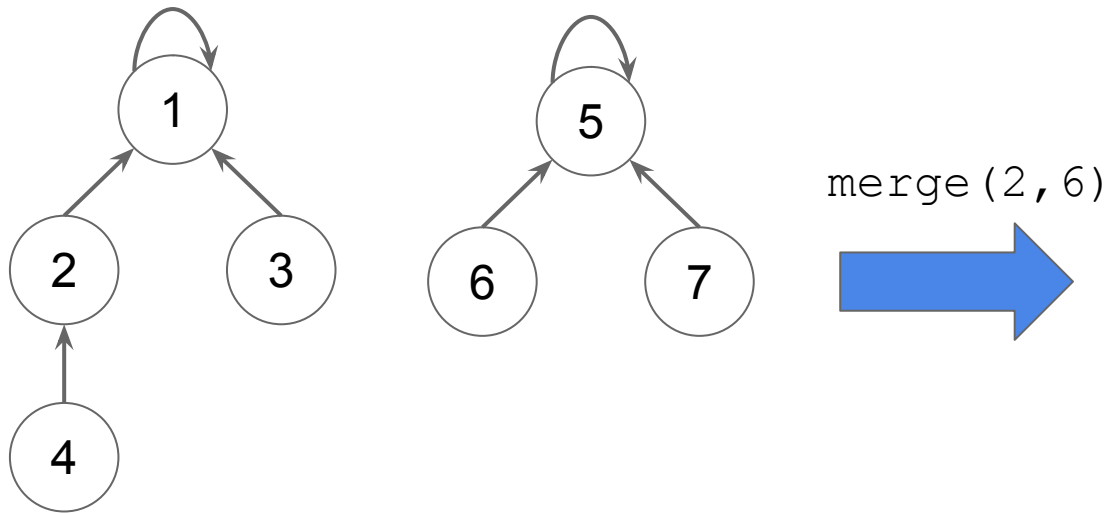
Implementação (nem tão) ingênua



pai =

1	1	1	2	5	5	5
---	---	---	---	---	---	---

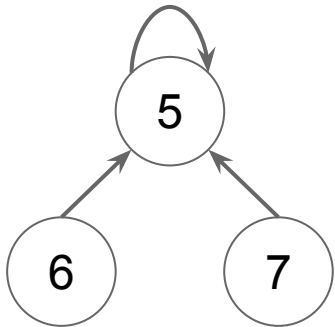
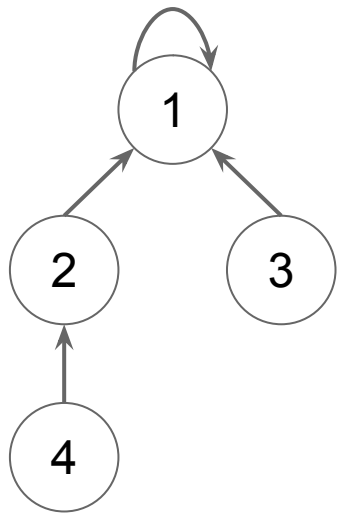
Implementação (nem tão) ingênua



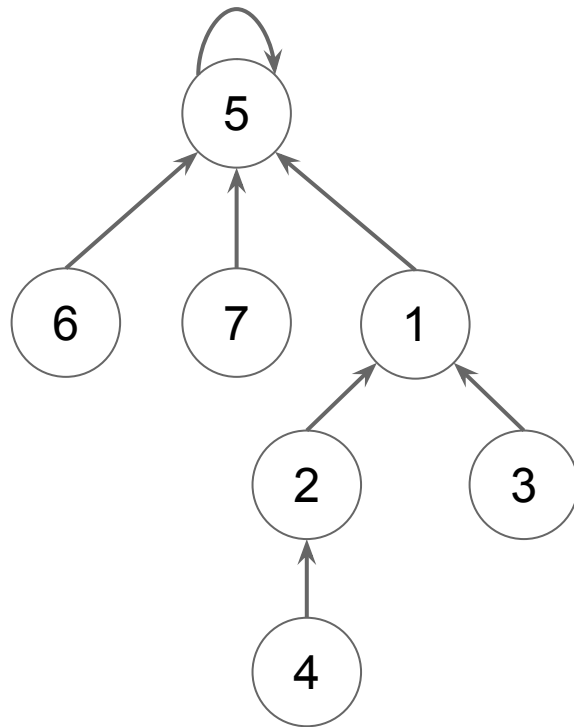
pai =

1	1	1	2	5	5	5
---	---	---	---	---	---	---

Implementação (nem tão) ingênua



merge (2, 6)



pai =

5	1	1	2	5	5	5
---	---	---	---	---	---	---

Implementação (nem tão) ingênua

- Qual o problema dessa implementação?
 - Casos degenerados que a tornam ineficiente

merge (5, 4)

merge (5, 3)

merge (3, 2)

merge (5, 1)

1

2

3

4

5

Implementação (nem tão) ingênua

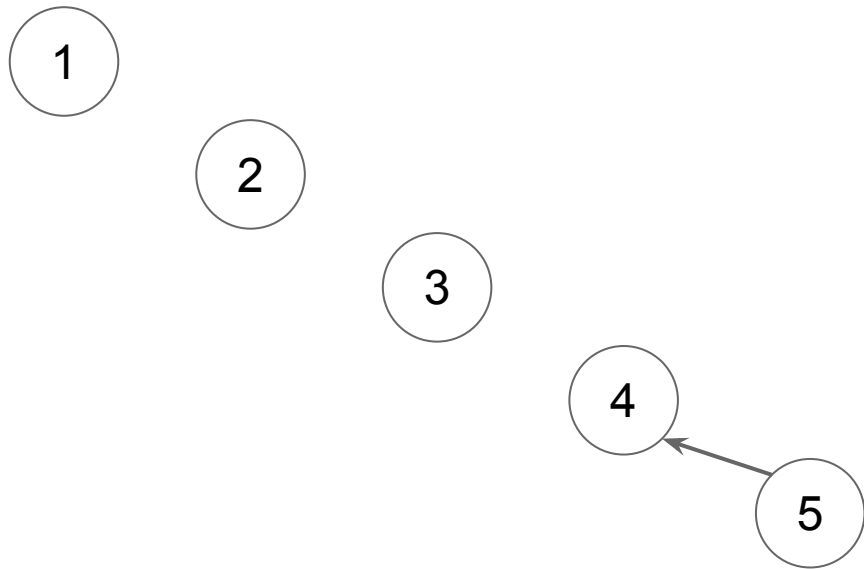
- Qual o problema dessa implementação?
 - Casos degenerados que a tornam ineficiente

merge (5, 4)

merge (5, 3)

merge (3, 2)

merge (5, 1)



Implementação (nem tão) ingênua

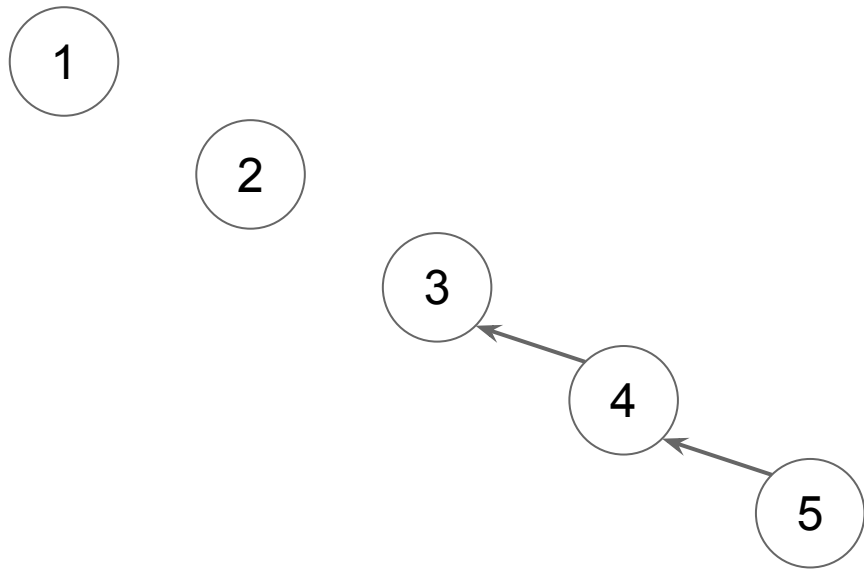
- Qual o problema dessa implementação?
 - Casos degenerados que a tornam ineficiente

merge (5, 4)

merge (5, 3)

merge (3, 2)

merge (5, 1)



Implementação (nem tão) ingênua

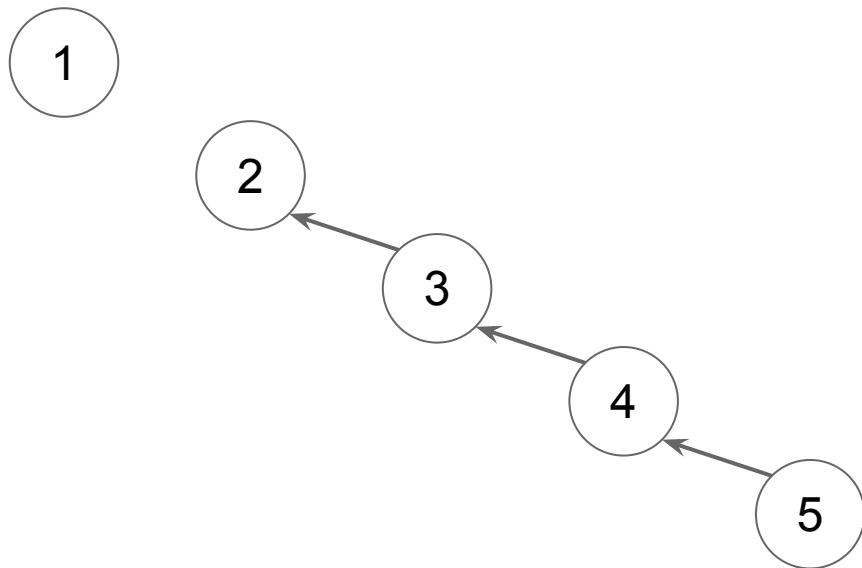
- Qual o problema dessa implementação?
 - Casos degenerados que a tornam ineficiente

merge (5, 4)

merge (5, 3)

merge (3, 2)

merge (5, 1)



Implementação (nem tão) ingênua

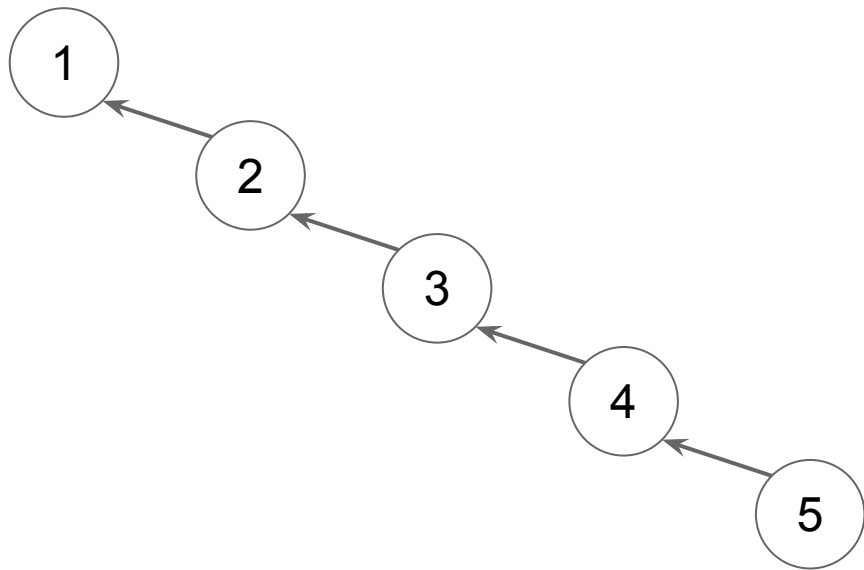
- Qual o problema dessa implementação?
 - Casos degenerados que a tornam ineficiente
 - Tempo de busca: $O(n)$

merge (5, 4)

merge (5, 3)

merge (3, 2)

merge (5, 1)



Path Compression

- Ideia: comprimir os caminhos, fazendo todos os elementos do conjunto apontarem para o líder diretamente.
- Uma espécie de Programação Dinâmica.
- Tempo por operação: $O(\log n)$ amortizado

Path Compression

```
int pai[MAX_N + 1];
```

```
void init() {  
    for(int i = 1; i <= MAX_N; i++)  
        pai[i] = i;  
}
```

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

```
void merge(int u, int v) {  
    int a = find(u);  
    int b = find(v);  
    pai[a] = b;  
}
```

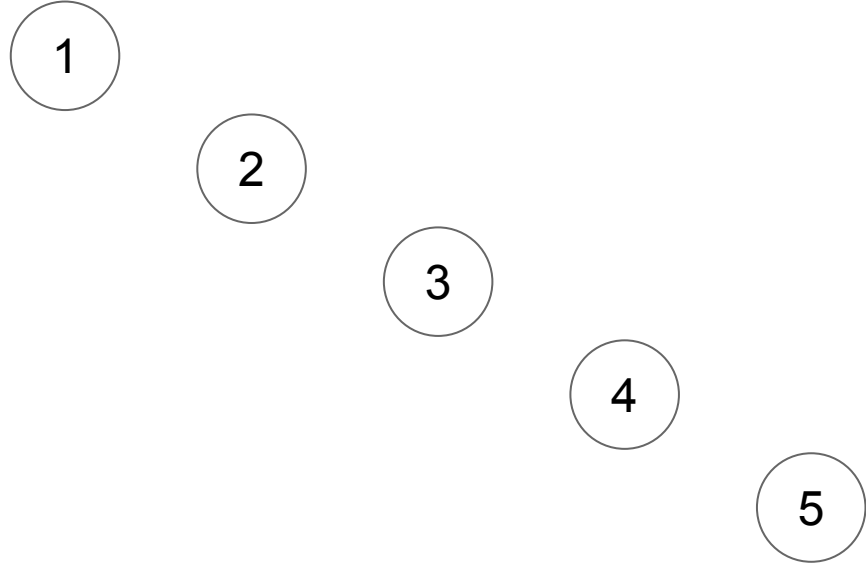
Path Compression

merge (5, 4)

merge (5, 3)

merge (5, 2)

merge (5, 1)



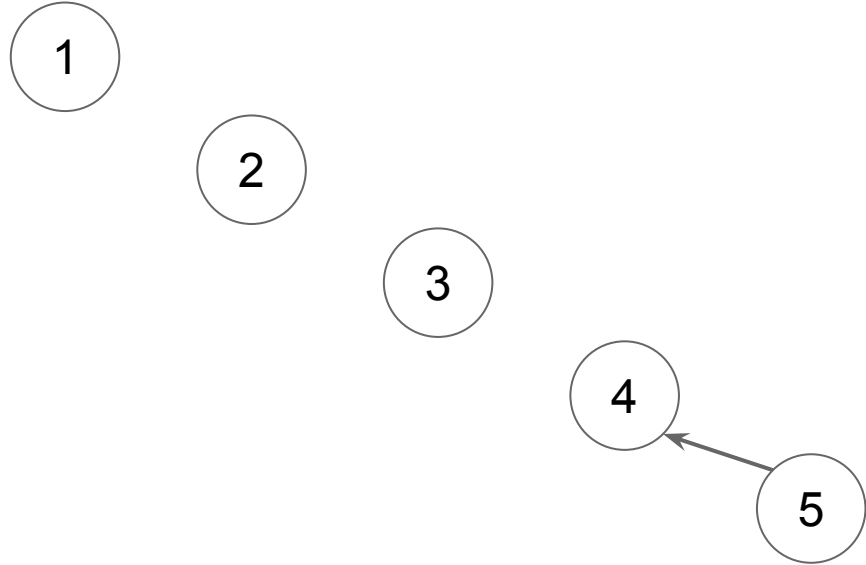
Path Compression

merge (5, 4)

merge (5, 3)

merge (5, 2)

merge (5, 1)



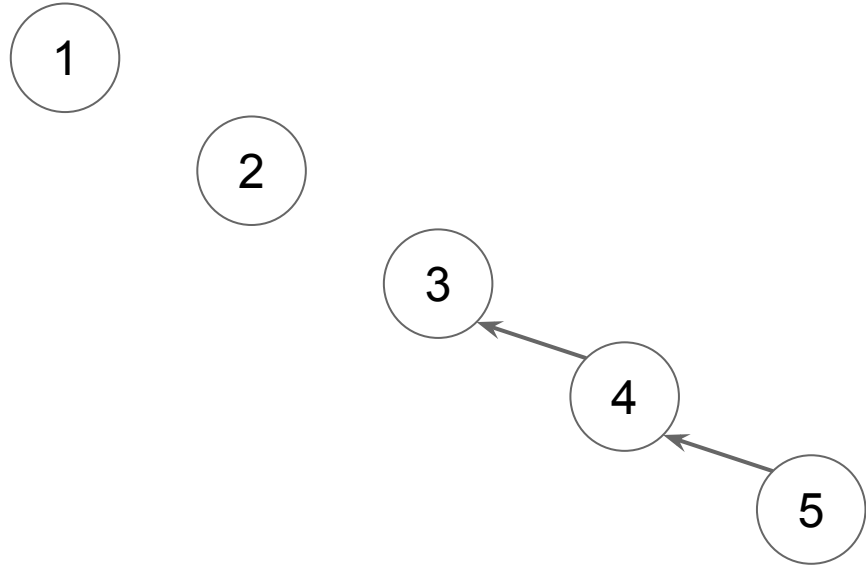
Path Compression

merge (5, 4)

merge (5, 3)

merge (5, 2)

merge (5, 1)



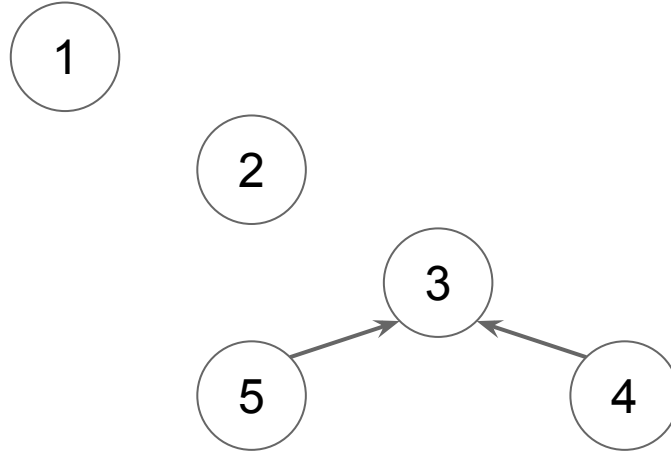
Path Compression

merge (5, 4)

merge (5, 3)

merge (5, 2)

merge (5, 1)



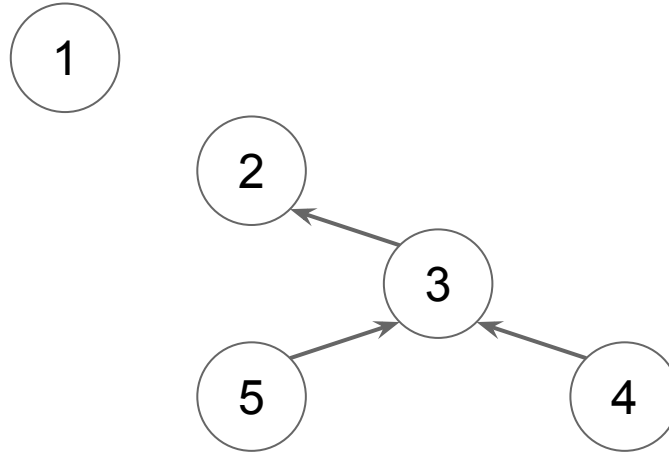
Path Compression

merge (5, 4)

merge (5, 3)

merge (5, 2)

merge (5, 1)



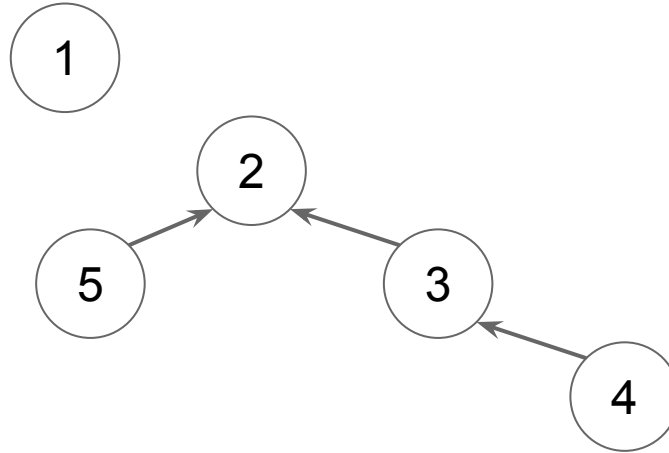
Path Compression

merge (5, 4)

merge (5, 3)

merge (5, 2)

merge (5, 1)



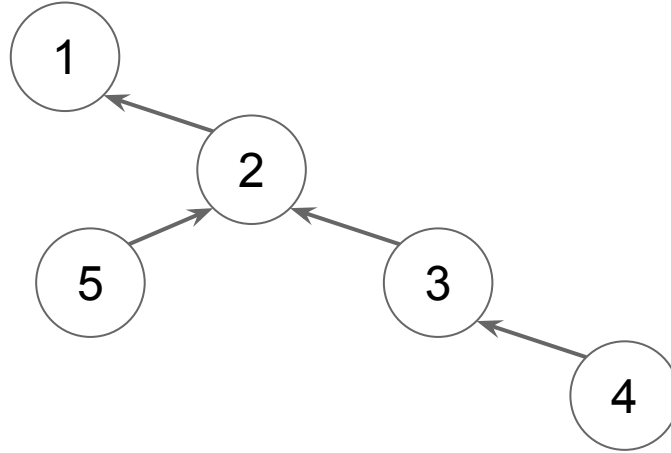
Path Compression

merge (5, 4)

merge (5, 3)

merge (5, 2)

merge (5, 1)



Union by size

- Ideia: unir os conjuntos do menor para o maior, minimizando a profundidade dos conjuntos.
- Estratégia “small-to-large”
- Tempo por operação: $O(\log n)$

Union by size

```
int pai[MAX_N + 1];
int tam[MAX_N + 1];

void init()
{
    for(int i = 1; i <= MAX_N; i++)
    {
        pai[i] = i;
        tam[i] = 1;
    }
}
```

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

void merge(int u, int v)
{
    int a = find(u);
    int b = find(v);
    if (tam[a] > tam[b])
        swap(a,b);
    pai[a] = b;
    tam[b] += tam[a];
}
```

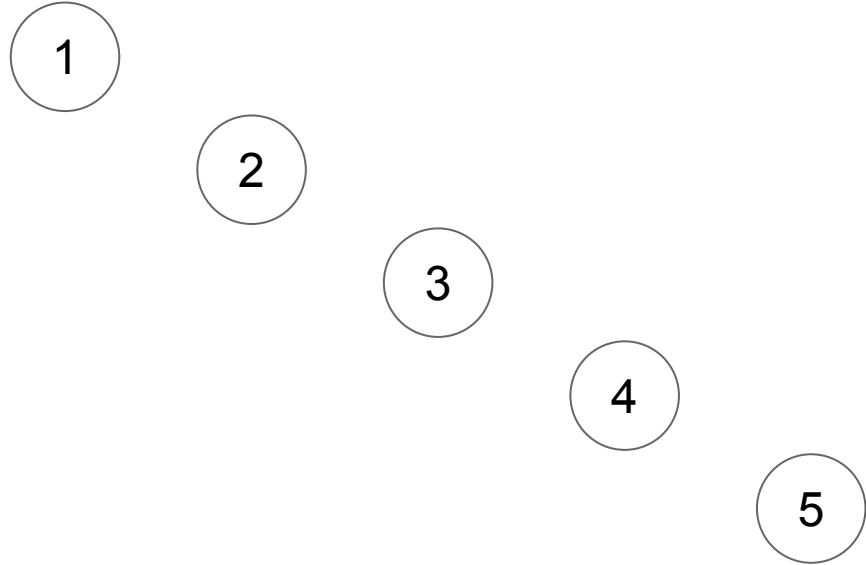
Union by size

`merge (5, 4)`

`merge (5, 3)`

`merge (2, 1)`

`merge (5, 1)`



`tam =`

1	1	1	1	1
---	---	---	---	---

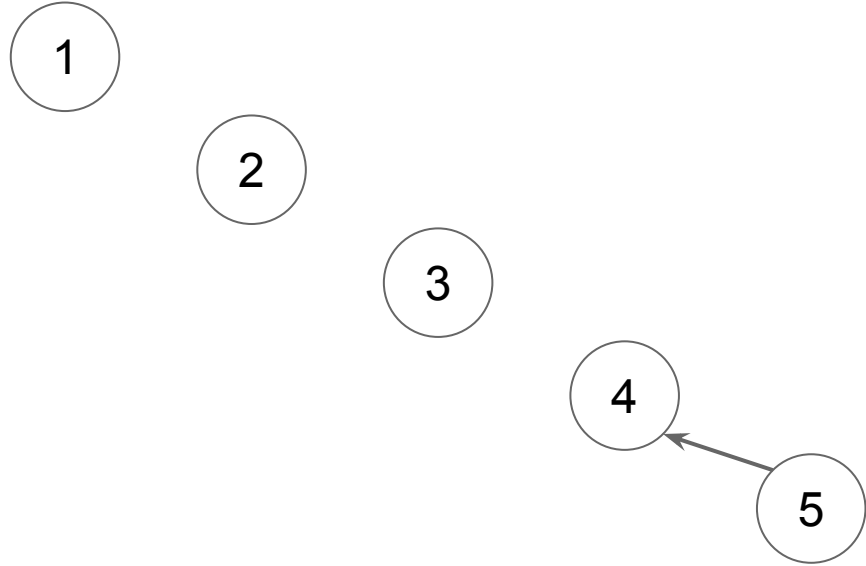
Union by size

merge (5, 4)

merge (5, 3)

merge (2, 1)

merge (5, 1)



tam =

1	1	1	2	1
---	---	---	---	---

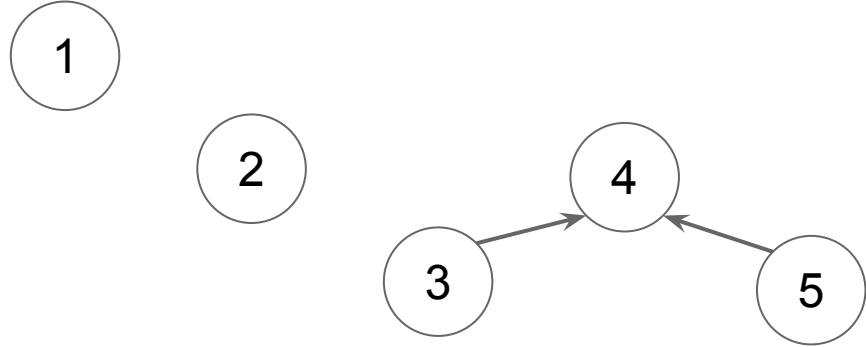
Union by size

merge (5, 4)

merge (5, 3)

merge (2, 1)

merge (5, 1)



tam =

1	1	1	3	1
---	---	---	---	---

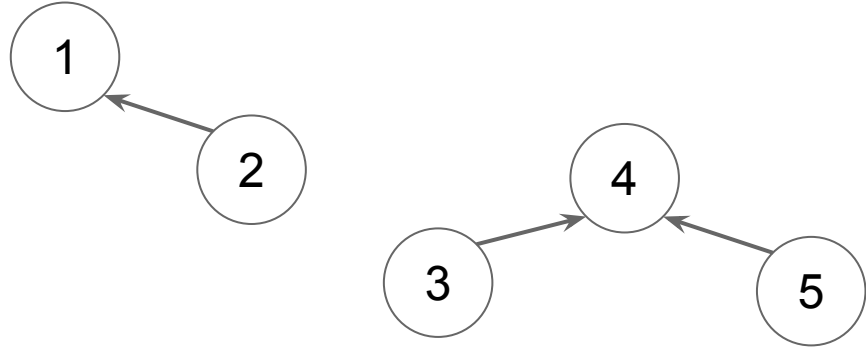
Union by size

merge (5, 4)

merge (5, 3)

merge (2, 1)

merge (5, 1)



tam =

2	1	1	3	1
---	---	---	---	---

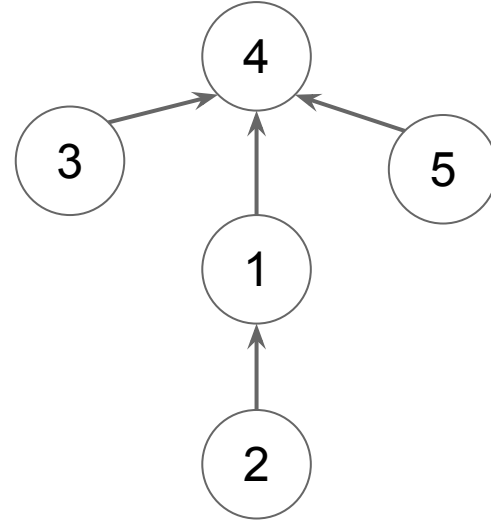
Union by size

merge (5, 4)

merge (5, 3)

merge (2, 1)

merge (5, 1)



tam =

2	1	1	5	1
---	---	---	---	---

Path Compression + Union by size

- Unindo ambas as técnicas garante-se complexidade quase constante, sendo a implementação mais recomendada.
- Outras técnicas de melhorias (na união dos conjuntos)
 - Union by rank
 - Linking by index
 - Coin-flip linking

Problema: WAR (UVa - 10158)

- Conjunto n de pessoas de diferentes países.
- Duas pessoas são consideradas amigas se são do mesmo país, ou inimigas se são diferentes países.
- Neste problema, podem ser feitas 4 tipos de operações:
 - setFriends(x, y)
 - setEnemies(x, y)
 - areFriends(x, y)
 - areEnemies(x,y)
- Se uma operação contradizer alguma anterior, ela não é realizada e imprimimos -1 na tela

Problema: WAR (UVa - 10158)

- Propriedades da amizade \sim :
 - Se $x \sim y$ e $y \sim z$, então $x \sim z$
 - Se $x \sim y$ então $y \sim x$
 - $x \sim x$
- Propriedades da inimizade $*$:
 - Se $x * y$ então $y * x$
 - Não acontece $x * x$
- E também
 - Se $x * y$ e $y * z$, então $x \sim z$ (o inimigo do meu inimigo é meu amigo)
 - Se $x \sim y$ e $y * z$, então $x * z$ (o inimigo do meu amigo é meu inimigo)

Problema: WAR (UVa - 10158)

- A partir dessas propriedades, podemos modelar o problema utilizando disjoint-sets. Cada pessoa x possui dois conjuntos associados, o conjunto $\text{amigos}(x)$ e o conjunto $\text{inimigos}(x)$. E estes conjuntos devem satisfazer as propriedades anteriores.
- Detalhe de implementação: nos exemplos que vimos até agora, criamos um vetor *pai* de tamanho N , de forma que cada elemento começa associado a um conjunto. Neste exercício vamos criar um vetor de tamanho $2*N$, onde a primeira metade são os conjuntos de amigos e a segunda de inimigos

Problema: WAR (UVa - 10158)

- Inicializando os conjuntos:
 - Toda pessoa é amiga dela mesma ($x \sim x$)
 - `pai[amigos(x)] = x;`
 - Ninguém é inimigo de si mesmo (Não $x * x$)
 - `pai[inimigos(x)] = 0; //Considerando as pessoas numeradas de 1 a n`

Problema: WAR (UVa - 10158)

- setFriends(x, y)
 - Primeiro, precisamos verificar se x e y não são inimigos, o que iria gerar uma contradição
 - Caso não, então fazemos
 - merge(amigos(x), amigos(y))
 - merge(inimigos(x), inimigos(y))

Problema: WAR (UVa - 10158)

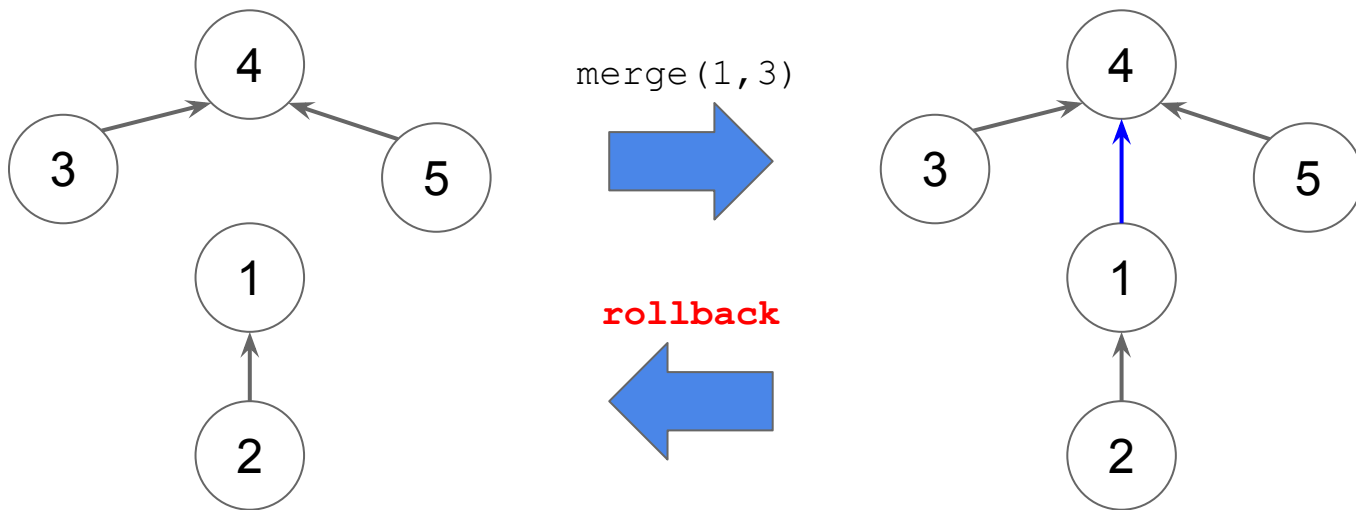
- setEnemies(x, y)
 - Primeiro, precisamos verificar se x e y não são amigos, o que iria gerar uma contradição
 - Caso não, então fazemos
 - merge(amigos(x), inimigos(y))
 - merge(inimigos(x), amigos(y))

Problema: WAR (UVa - 10158)

- areFriends(x, y)
 - same(amigos(x), amigos(y))
- areEnemies(x, y)
 - same(amigos(x), inimigos(y))

Rollback

- Em alguns problemas, pode ser necessário realizar um *rollback*, desfazendo uniões imediatamente anteriores



Rollback

- A cada operação *merge*, são feitas duas atribuições:
 - `pai[i] = x;`
 - `tam[j] = y;`
- Sendo assim, basta salvar os valores antigos em uma pilha, para restaurar se for necessário.
 - Uma pilha para o vetor `pai`: $\langle i, \text{pai}[i] \rangle$
 - Uma pilha para o vetor `tam`: $\langle j, \text{tam}[j] \rangle$
- Não permite usar *path compression*.

Rollback

```
int pai[MAX_N + 1]
int tam[MAX_N + 1]
stack<pair<int, int> > old_pai;
stack<pair<int, int> > old_tam;

void init()
{
    for(int i = 1; i <= MAX_N; i++)
    {
        pai[i] = i;
        tam[i] = 1;
    }
}
```

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

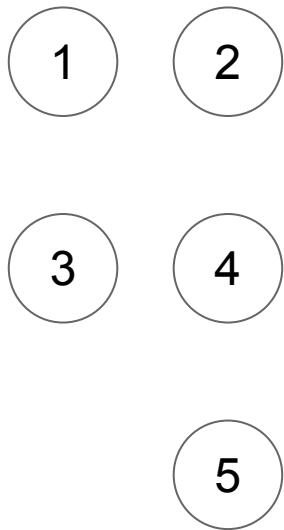
Rollback

```
void merge(int u, int v)
{
    int a = find(u);
    int b = find(v);
    if (tam[a] > tam[b])
        swap(a,b);
    old_pai.emplace(a, pai[a]);
    old_tam.emplace(b, tam[b]);
    pai[a] = b;
    tam[b] += tam[a];
}
```

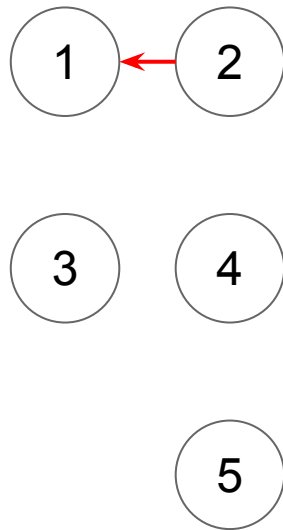
```
void roolback()
{
    auto paiAnt = old_pai.top();
    auto tamAnt = old_tam.top();
    pai[paiAnt.first] = paiAnt.second();
    tam[tamAnt.first] = tamAnt.second();
    old_pai.pop();
    old_tam.pop();
}
```

Persistência parcial

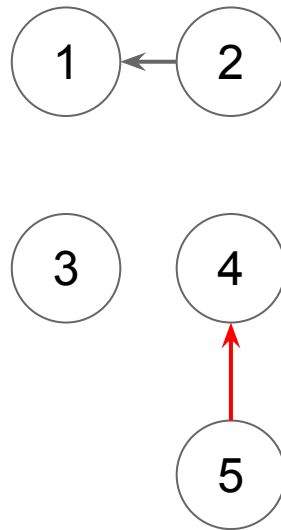
- O objetivo é poder consultar versões anteriores da estrutura.



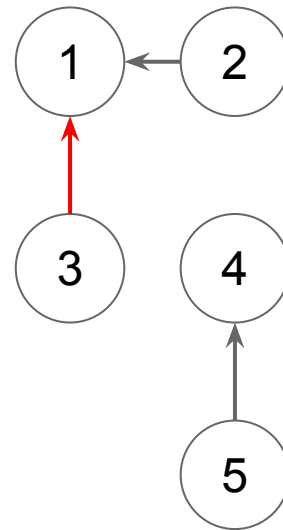
Versão 0



Versão 1



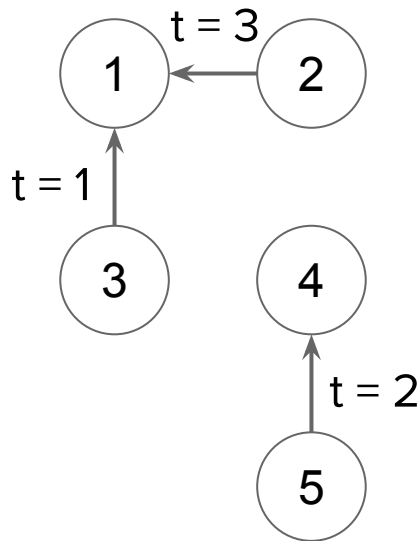
Versão 2



Versão 3

Persistência parcial

- Ideia: anotar o tempo de cada ligação



Persistência parcial

- Operações básicas:
 - **merge(x, y)**: conecta os conjuntos de **x** e **y** (criando uma nova “versão”)
 - **find(x, t)**: retorna a qual conjunto **x** pertence no momento **t**
 - **same(x, y, t)**: verifica se **x** e **y** pertencem ao mesmo conjunto no momento **t**

Persistência parcial

```
int pai[MAX_N + 1];
int tam[MAX_N + 1];
int his[MAX_N + 1];
int tempo;

void init()
{
    tempo = 0;
    for(int i = 1; i <= MAX_N; i++)
    {
        pai[i] = i;
        tam[i] = 1;
        his[i] = 0;
    }
}
```

```
int find(int x, int t)
{
    if (pai[x] == x) return x;
    if (his[x] > t) return x;
    return find(pai[x]);
}

void merge(int u, int v)
{
    tempo++;
    int a = find(u, tempo);
    int b = find(v, tempo);
    if (tam[a] > tam[b])
        swap(a,b);
    pai[a] = b;
    his[a] = tempo;
    tam[b] += tam[a];
}
```

Pictionary (Gym - 102078A)

- Neste problema, temos um conjunto de N cidades, inicialmente todas desconectadas.
- Rodovias são construídas entre as cidades em M dias. Em um dia i , é construída uma estrada entre a e b se $\text{gcd}(a, b) = M - i + 1$.
- São feitas Q *queries*, constituídas por pares de cidades. O resultado de cada *query* é o número mínimo de dias necessários para conectar o par de cidades (direta ou indiretamente)

Pictionary (Gym - 102078A)

- Usaremos disjoint-sets com persistência parcial para unir as cidades marcando o momento em que as uniões foram feitas.
 - Para cada tia d , vamos conectar as cidades com $\gcd(a,b) = M - d + 1$, que vamos chamar de x . Iterando sobre d , vamos realizar um merge de x com todos os seus múltiplos (até n).
 - Mas e se duas cidades $(x, k.x)$ já tiverem sido conectadas indiretamente antes? Sem problemas, o merge vai verificar que as cidades já foram unidas em um momento anterior e não vai fazer nada.
- Para realizar as *queries*, executaremos uma busca binária para descobrir o número mínimo de dias necessários para conectar as cidades a e b .

Referências

<https://files.johnjq.com/slides/summer/union-find.pdf>

https://github.com/icmcgema/gema/blob/master/XX-Union_Find.md

https://cp-algorithms.com/data_structures/disjoint_set_union.html

<https://www.youtube.com/watch?v=E33jZUw2l9Q>