GRUPRO — Union-Find e extras

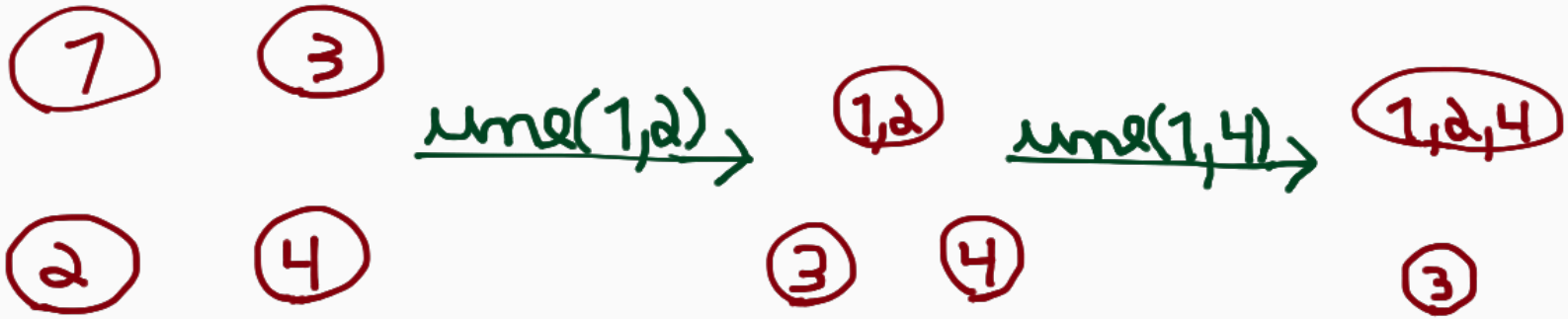
introdução, rollback, small-to-large

Jonathan Queiroz

Union-Find

Apresentação do problema

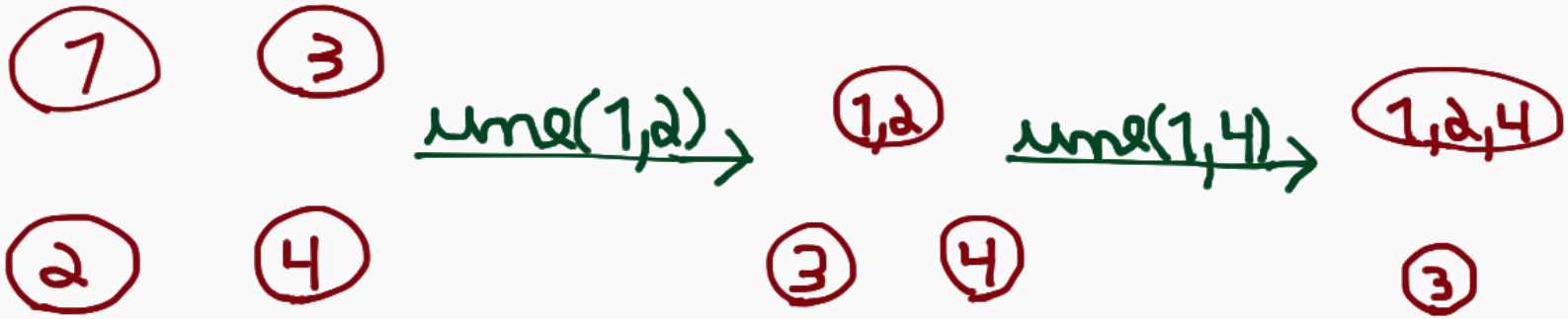
- Temos vários conjuntos (disjuntos) de elementos
  - Inicialmente, cada conjunto contém precisamente um elemento
- Duas operações:
  - `merge(a, b)`: une os conjuntos aos quais *a* e *b* pertencem
  - `same(a, b)`: verifica se *a* e *b* pertencem ao mesmo conjunto
- Exemplo
  - Esquerda: `same(1, 2) = False`
  - Meio: `same(1, 2) = True`
  - Direita: `same(2, 4) = True`



Abordagem (nem tão) ingênua

Teoria

- Podemos eleger um **líder** para cada conjunto
- Reformulando as operações:
  - `merge(a, b)`: une os conjuntos aos quais *a* e *b* pertencem
  - `find(a)`: retorna o líder do conjunto *a*
- Podemos reescrever `same(a, b)` como `find(a) == find(b)`
  - Esquerda: `find(1) = 1` $\neq$ `2 = find(2)` $\therefore$ `same(1, 2) = F`
  - Meio: `find(1) = 1 = find(2)` $\therefore$ `same(1, 2) = T`
  - Direita: `find(2) = 1 = find(4)` $\therefore$ `same(1, 2) = T`

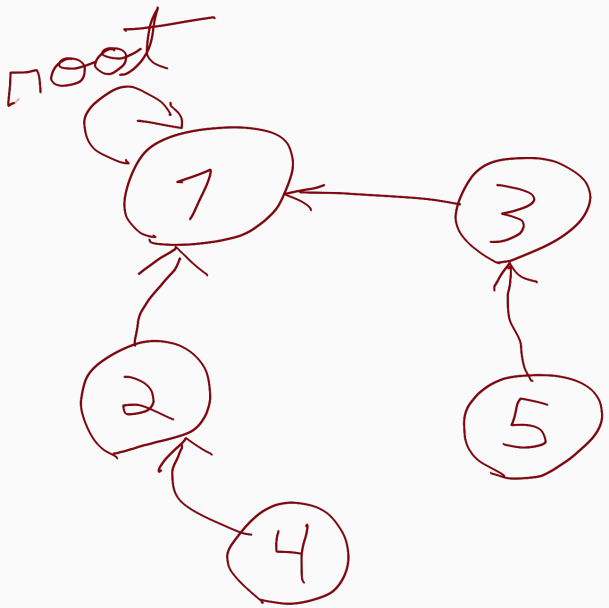


Abordagem (nem tão) ingênua

Ideia da implementação

- Representar cada conjunto como uma árvore
- Basta saber o pai de cada elemento
  - `uf[i]` armazena o índice do pai do i-ésimo elemento
  - `uf[i] = i` indica que o i-ésimo elemento é o líder do seu conjunto

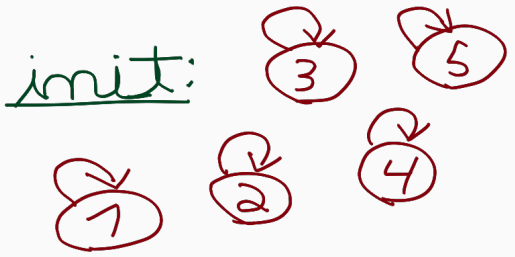
`i`	`uf[i]`
1	1
2	1
3	1
4	2
5	3

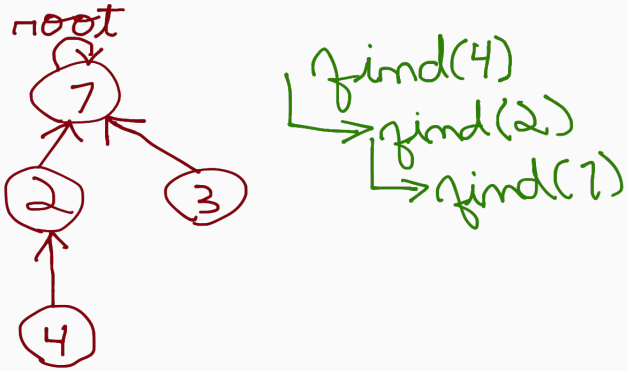


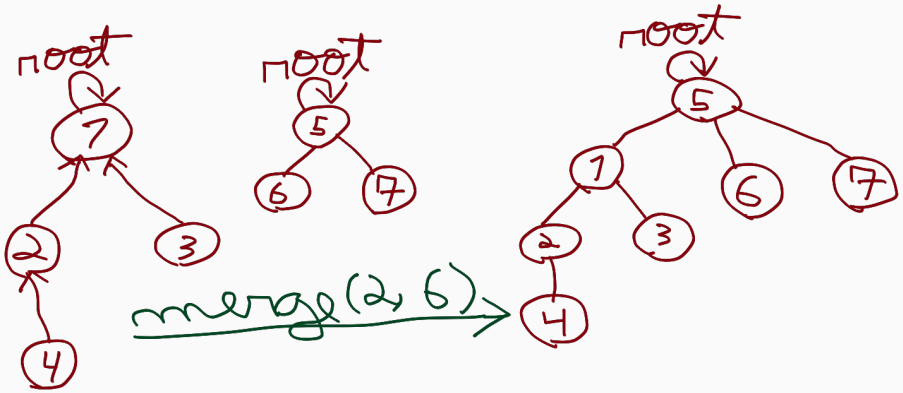
Abordagem (nem tão) ingênua

Implementação

```
int uf[MAX_N+1];
void init() {
    for (int i = 1; i <= MAX_N; ++i)
        uf[i] = i;
}
int find(int x) {
    if (uf[x] == x)
        return x;
    else
        return find(uf[x]);
}
void merge(int u, int v) {
    int a = find(u);
    int b = find(v);
    uf[a] = b;
}
```

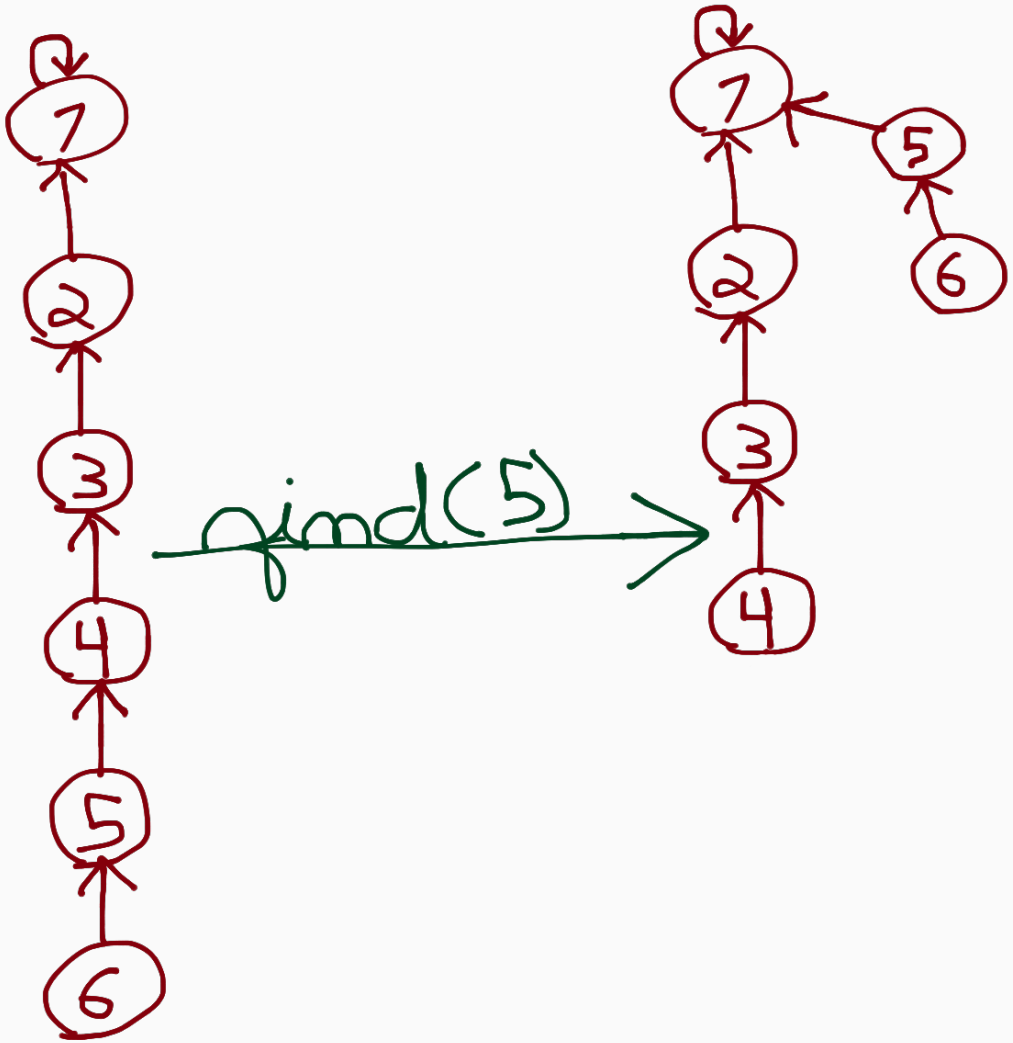






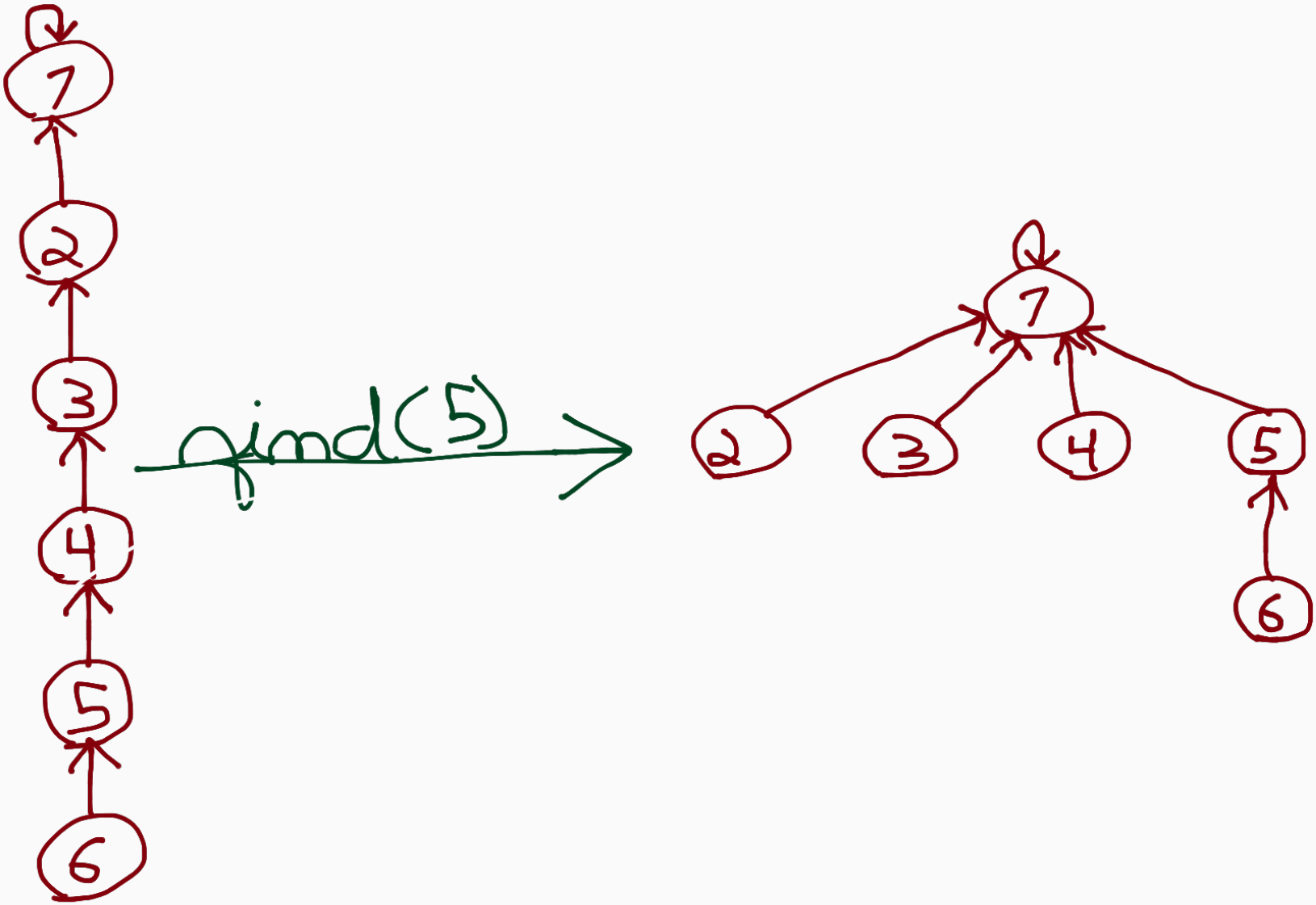
Abordagem (nem tão) ingênua

Como melhorar?



Abordagem (nem tão) ingênua

Como melhorar ainda mais?



Abordagem: path compression

Teoria

- A cada operação `find`: fazer com que todos os nós acessados apontem diretamente para a raiz
- Intuição: buscas custosas ajudam a melhorar a árvore
- Tempo por operação: $O(\log n)$ *amortizado*
    - Operações individuais podem ter custo elevado
        - Exemplo?
    - Mas o custo médio das operações é baixo
    - Formalmente: m operações levam tempo $O(n + m \log n)$
        - Prova relativamente complexa

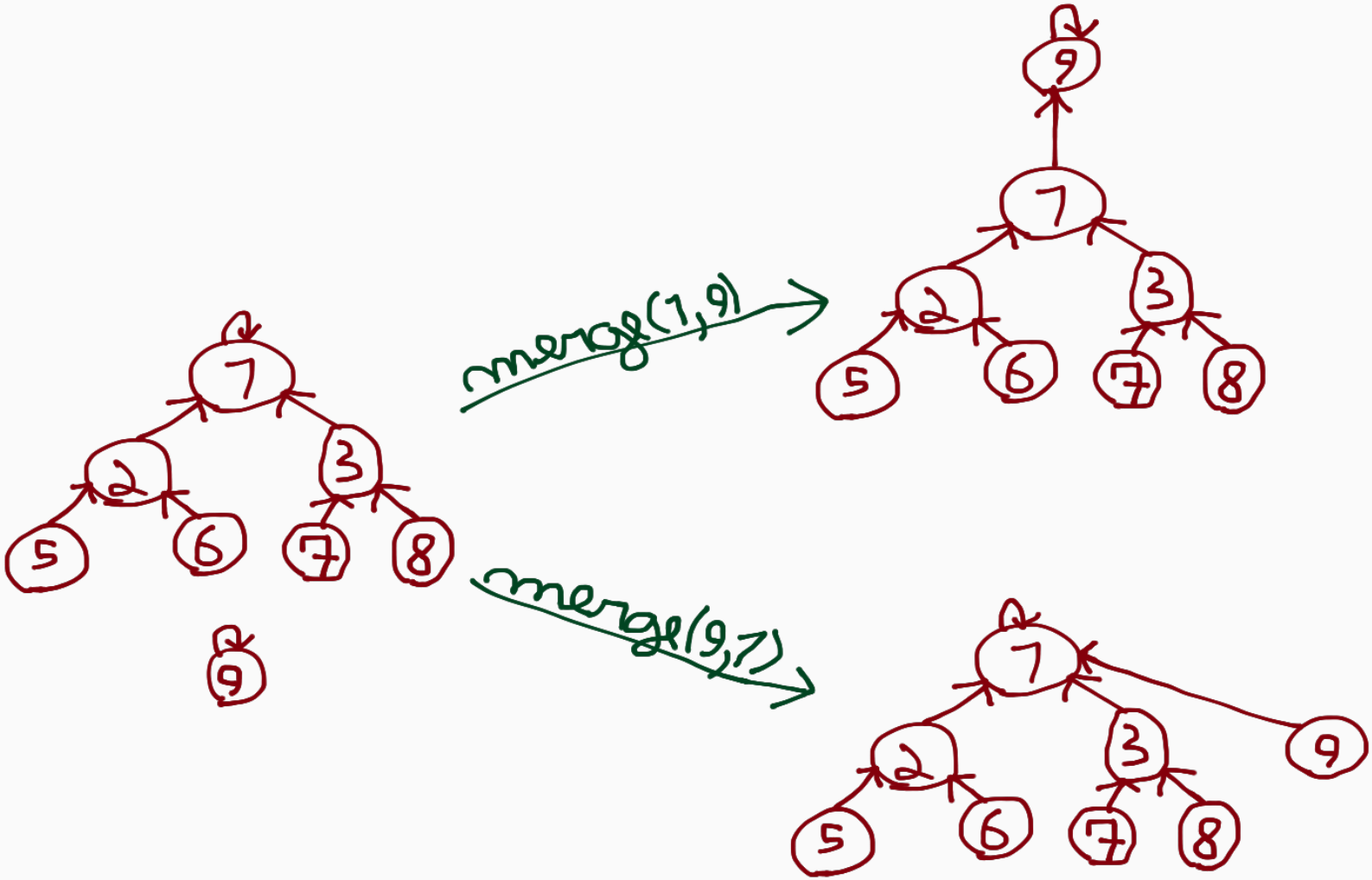
Abordagem: path compression

Implementação

```
int uf[MAX_N+1];
void init() {
    for (int i = 1; i <= MAX_N; ++i)
        uf[i] = i;
}
int find(int x) {
    if (uf[x] == x)
        return x;
    else
        return uf[x] = find(uf[x]);
}
void merge(int u, int v) {
    int a = find(u);
    int b = find(v);
    uf[a] = b;
}
```

Abordagem: union by size

Outra forma de melhorar



Abordagem: union by size

Teoria

- Unir conjuntos do menor para o maior
  - É necessário manter o tamanho de cada conjunto v
- Intuição: minimizar a profundidade dos conjuntos
- Tempo por operação: $O(\log n)$
  - Provaremos em breve

Implementação (parte 1)

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

Abordagem: union by size

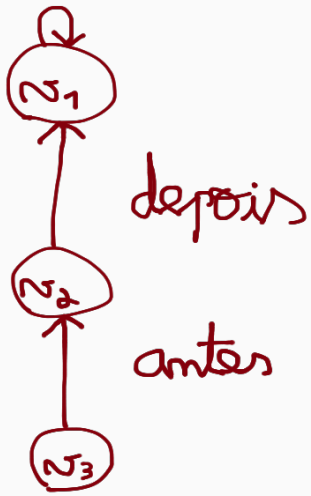
Implementação (parte 2)

```
int find(int x) {
    if (uf[x] == x) return x;
    else            return find(uf[x]);
}
void merge(int u, int v) {
    int a = find(u);
    int b = find(v);
    if (sz[a] < sz[b]) {
        uf[a] = b;
        sz[b] += sz[a];
    } else {
        uf[b] = a;
        sz[a] += sz[b];
    }
}
```

Abordagem: union by size

Análise de complexidade (versão "small-to-large")

- Observação
  - Seja $v_1 v_2 \ldots v_n$ um caminho, sendo v_1 o líder do conjunto
  - Então as uniões foram feitas na seguinte ordem:
    - conjunto de v_n com conjunto de v_{n-1}
    - conjunto de v_{n-1} com conjunto de v_{n-2}
    - ...
    - conjunto de v_3 com conjunto de v_2
    - conjunto de v_2 com conjunto de v_1
  - Por quê?



- Consequência
  - A cada união feita, o tamanho do conjunto pelo menos dobra
  - Logo a profundidade de um nó não pode exceder $\log_2 n$

Abordagem: union by size

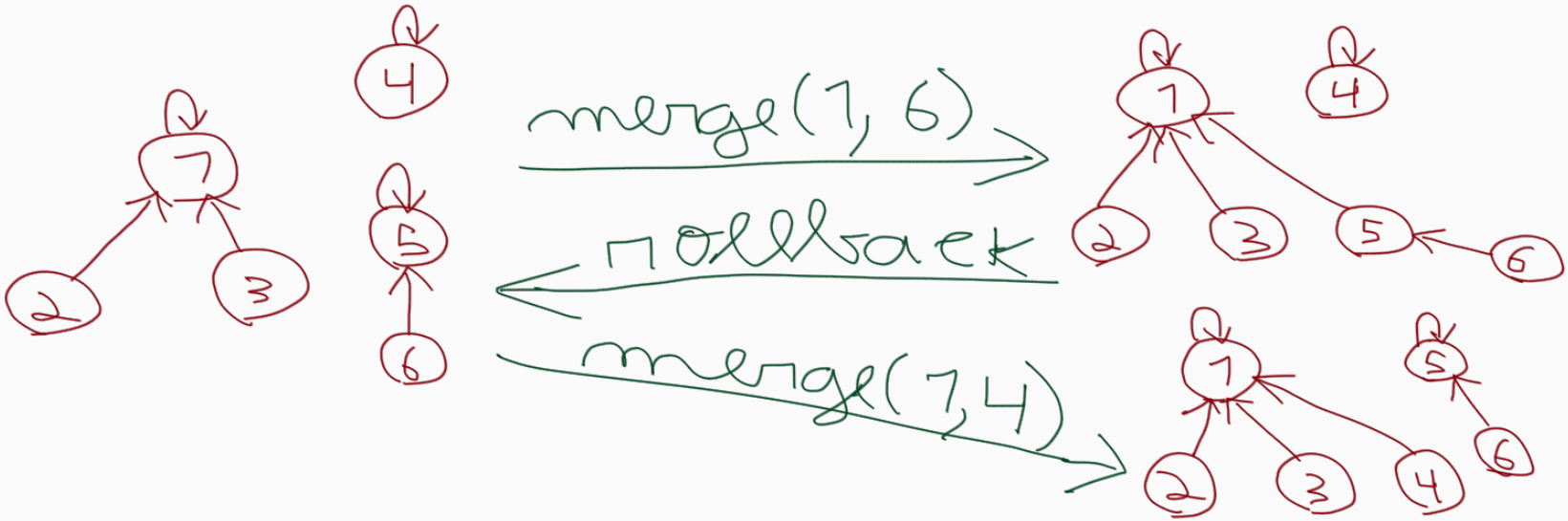
Análise de complexidade (versão alternativa)

- Para cada conjunto v, denotamos:
  - o seu tamanho por s_v; e
  - a sua profundidade por d_v
- Invariante: $(\forall v)\ \ d_v \leq \log_2 s_v$
  - Consequência: $(\forall v)\ \ d_v \leq \log_2 n$
- No início do algoritmo, isso é verdade:
  - $(\forall v)\ \ d_v = 0 \quad \text{e} \quad s_v = 1 \therefore \log_2 s_v = 0$
- A invariante é mantida a cada união:
  - Sejam x e y dois conjuntos com $s_y \leq s_x$
  - Por hipótese: $d_x \leq \log_2 s_x \quad \text{e} \quad d_y \leq \log_2 s_y$
  - Queremos mostrar: $d'_x \leq \log_2(s_x + s_y)$
    - Fato: $d'_x = \max\{d_x, d_y + 1\}$
  - Caso 1: $d'_x = d_x$
    - Fácil: $d'_x = d_x \leq_{\text{HIP}} \log_2 s_x \leq \log_2(s_x + s_y)$
  - Caso 2: $d'_x = d_y + 1$
    - $d'_x = d_y + 1 \leq_{\text{HIP}} \log_2 s_y + 1 = \log_2(s_y + s_y) \leq \log_2(s_x + s_y)$

Rollback

Rollback com union by size

- Podemos desfazer as uniões imediatamente anteriores



- A cada operação `merge`, são feitas duas atribuições:
  - `uf[i] = x;`
  - `sz[j] = y;`
- Basta salvar os valores antigos em uma pilha!

Rollback com union by size

Implementação (parte 1)

```
int uf[MAX_N+1];
int sz[MAX_N+1];
stack<pair<int, int>> old_uf;
stack<pair<int, int>> old_sz;
void init() {
    for (int i = 1; i <= MAX_N; ++i) {
        uf[i] = i;
        sz[i] = 1;
    }
}
int find(int x) {
    if (uf[x] == x) return x;
    else            return find(uf[x]);
}
```

Rollback com union by size

Implementação (parte 2)

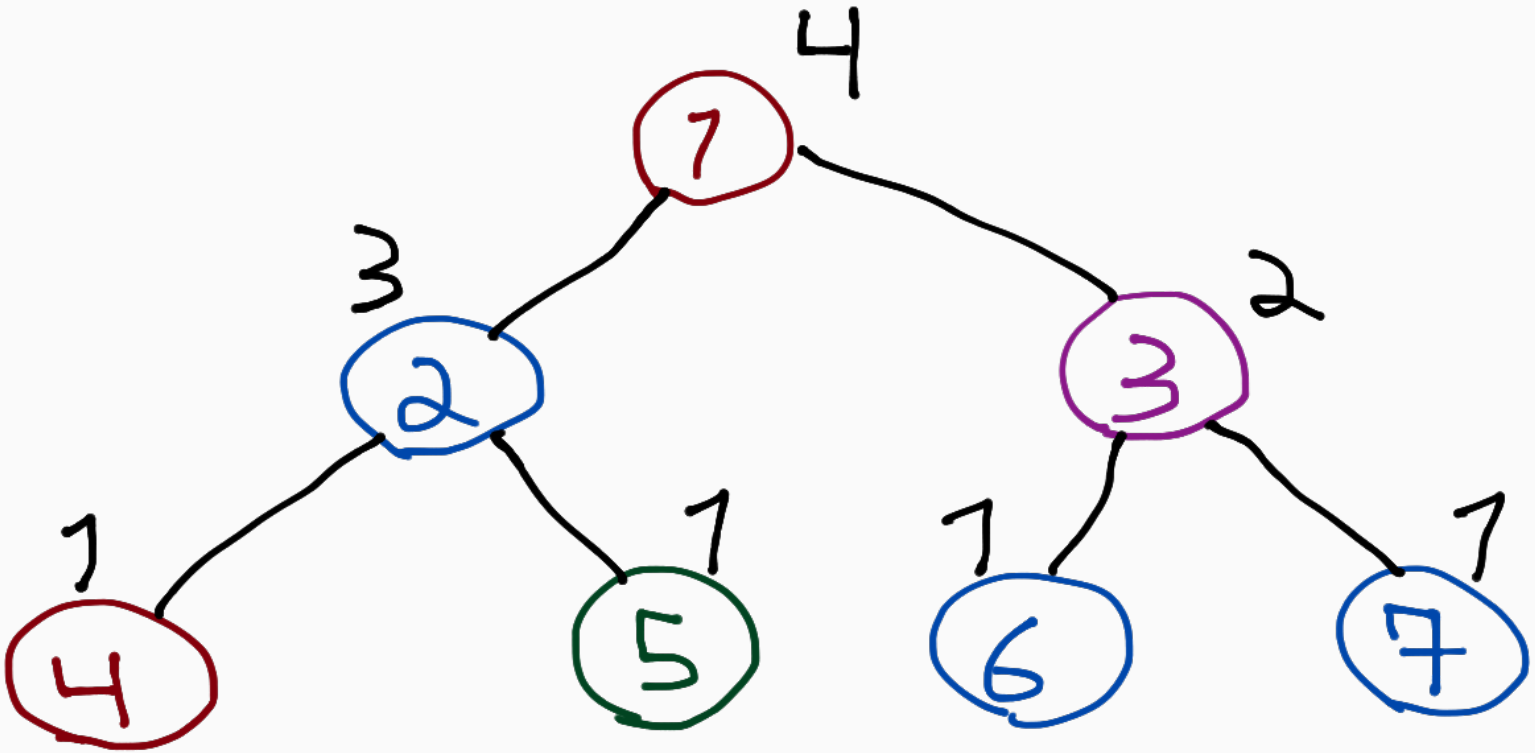
```
void merge(int u, int v) {
    int a = find(u);
    int b = find(v);
    if (sz[a] > sz[b]) swap(a, b);
    old_uf.emplace(a, uf[a]);
    old_sz.emplace(b, sz[b]);
    uf[a] = b;
    sz[b] += sz[a];
}
void rollback() {
    uf[old_uf.top().first] = old_uf.top().second;
    sz[old_sz.top().first] = old_sz.top().second;
    old_uf.pop();
    old_sz.pop();
}
```

Small-to-large

Cores em subárvores

Descrição do problema

- Entrada: árvore na qual os vértices possuem cores
- Objetivo: encontrar a quantidade de cores da subárvore enraizada em cada vértice



Cores em subárvores

Abordagem ingênua

- Busca em profundidade
- Para cada nó, unir os conjuntos de cores dos seus filhos

```
set<int> dfs(int v) {
    set<int> colors;
    colors.insert(color[v]);
    for (int w : adj[v])
        for (int c : dfs(w))
            colors.insert(c);
    ans[v] = colors.size();
    return colors;
}
```

- Complexidade no pior caso: $\Theta(n^2 \log n)$
  - Como construir tal caso?

Cores em subárvores

Abordagem "small-to-large"

- Mesma ideia, mas unindo conjuntos do menor para o maior

```
set<int> colors[MAX_N+1];
void dfs(int v) {
    colors[v].insert(color[v]);
    for (int w : adj[v]) {
        dfs(w);
        if (colors[w].size() > colors[v].size())
            swap(colors[v], colors[w]);
        for (int c : colors[w])
            colors[v].insert(c);
    }
    ans[v] = colors[v].size();
}
```

- Complexidade no pior caso: $\Theta(n \log^2 n)$
  - Mesmo argumento do *union by size*!

Exercícios

Exercícios (links clicáveis)

Competição no codepit

- Link: http://goo.gl/d2yQAT

Union-Find

- Fusões (SPOJ-BR)
- Famílias de Troia (SPOJ-BR)
- Gincana (SPOJ-BR)
- Energia (SPOJ-BR)
- Learning languages (codeforces) — desafio
- War (UVa) — desafio

Rollback

- Coleção de livros (codeforces)
    - necessário entrar no grupo do GRUPRO no codeforces (clique aqui)

Small-to-large

- Tree (e-olymp) — não consta no codepit
- Lowest Common Ancestor (SPOJ) — desafio
- Lomsat gelral (codeforces) — super-desafio