DSU & tricks

introdução, rollback, aumento, small-to-large , min(x,n-x) trick

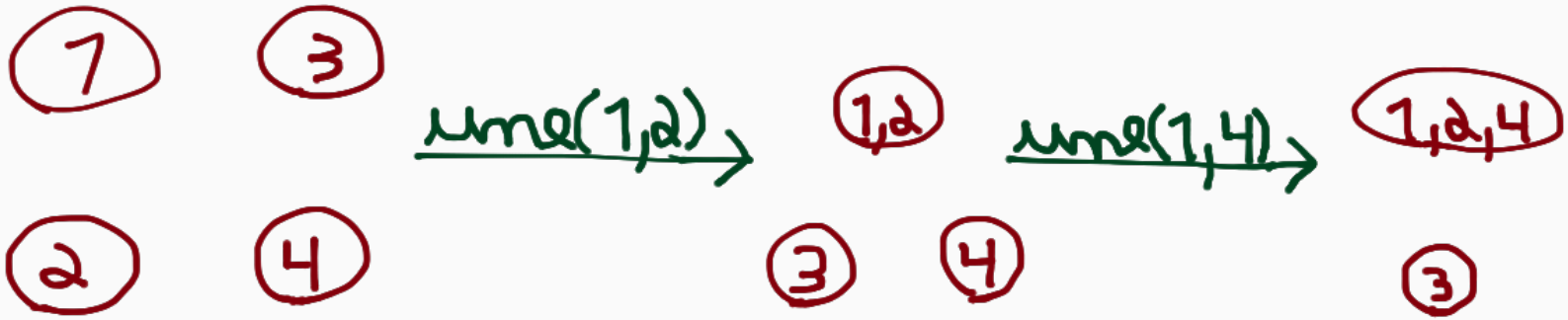
---

Tiago Domingos

DSU

Apresentação do problema

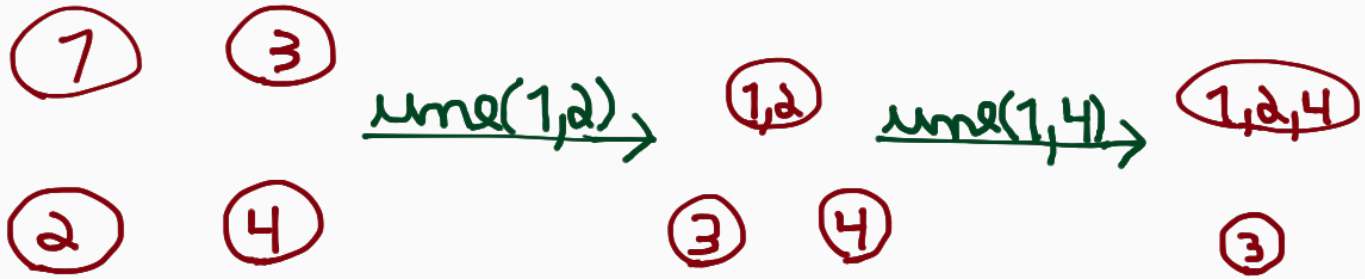
- Temos vários conjuntos (disjuntos) de elementos
  - Inicialmente, cada conjunto contém precisamente um elemento
  - Queremos lidar (rapidamente) com dois tipos de operações
- Duas operações:
  - `une(a, b)`: une os conjuntos aos quais *a* e *b* pertencem
  - `same(a, b)`: verifica se *a* e *b* pertencem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



Tentativa (um pouco) ingênua de solução

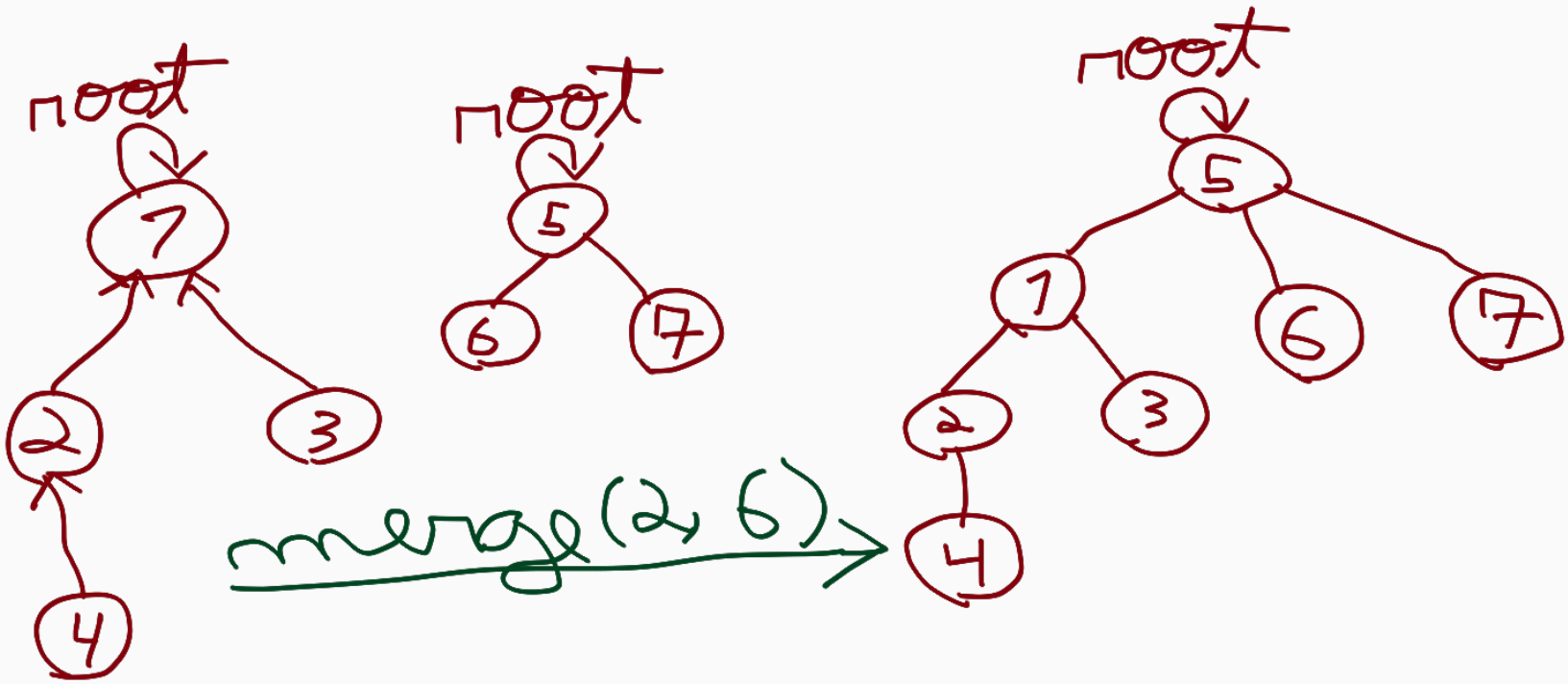
Ideia

- Vamos eleger um **líder** para cada conjunto
- Duas operações:
  - `une(a, b)`: une os conjuntos aos quais *a* e *b* pertencem, o líder da união será o líder do conjunto b.
  - `find(a)`: encontra o líder do conjunto que a está contido.
- `same(a,b)` –> `find(a) == find(b)`
- Exemplo
  - Esquerda: `find(i) = {1,2,3,4}`
  - Meio: `find(i) = {2,2,3,4}`
  - Direita: `find(i) = {4,4,3,4}`



Tentativa (um pouco) ingênua de solução

Exemplo

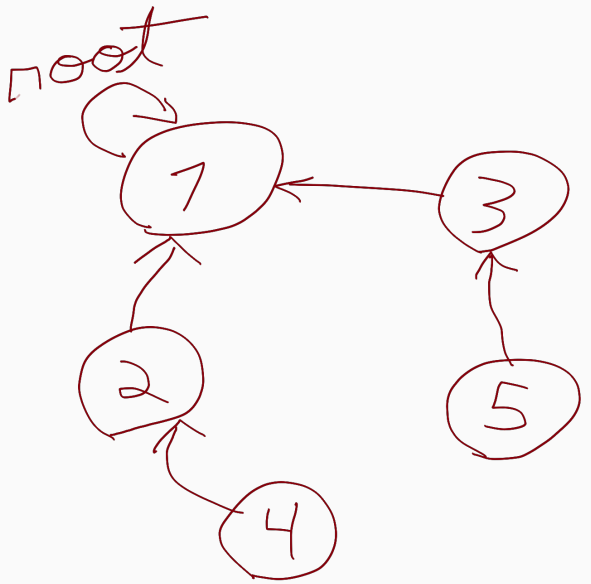


Tentativa (um pouco) ingênua de solução

Ideia da implementação

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

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



Tentativa (um pouco) ingênua de solução

```
int pai[N];
void init(){
    for(int i = 0 ; i < N ; i ++)
        pai[i] = i;
}
int find(int x){
    if(pai[x] == x)
        return x;
    else
        return find(pai[x]);
}
void une(int x , int y){
    x = find(x) , y = find(y);
    pai[x] = y;
}
```

Tentativa (um pouco) ingênua de solução

Quebrando essa ideia

Considere a seguinte sequência de operações: `une(1,2)` , `une(2,3)` , ... , `une(n-1,n)`

`find(1)` é $O(n)$

Compressão de caminho

Seja preguiçoso

- Estamos repetindo operações sem que o resultado mude
  - Podemos trabalhar apenas quando necessário
- Note que podemos salvar para onde `pai[i]` vai apontar após o final de find(i)

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

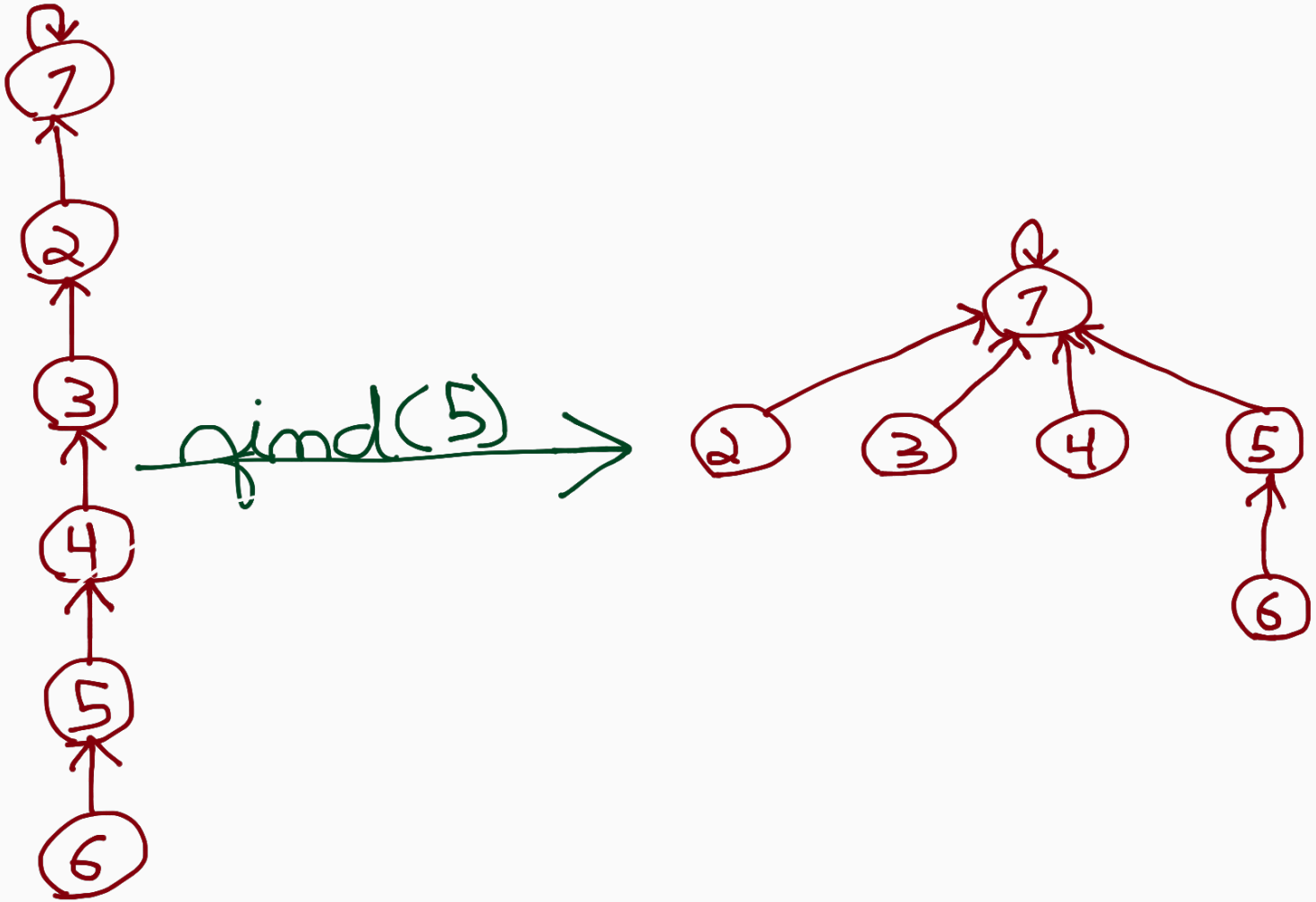
Compressão de caminho

Teoria

- A cada operação `find`: apontar todos os nós acessados 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 $O(n + m \log n)$

Compressão de caminho

Exemplo



Compressão de caminho

Prova
Podemos provar usando argumentos similares ao de HLD, caso alguém tenha curiosidade a prova pode ser realizada no final da aula.

União por tamanho

Teoria

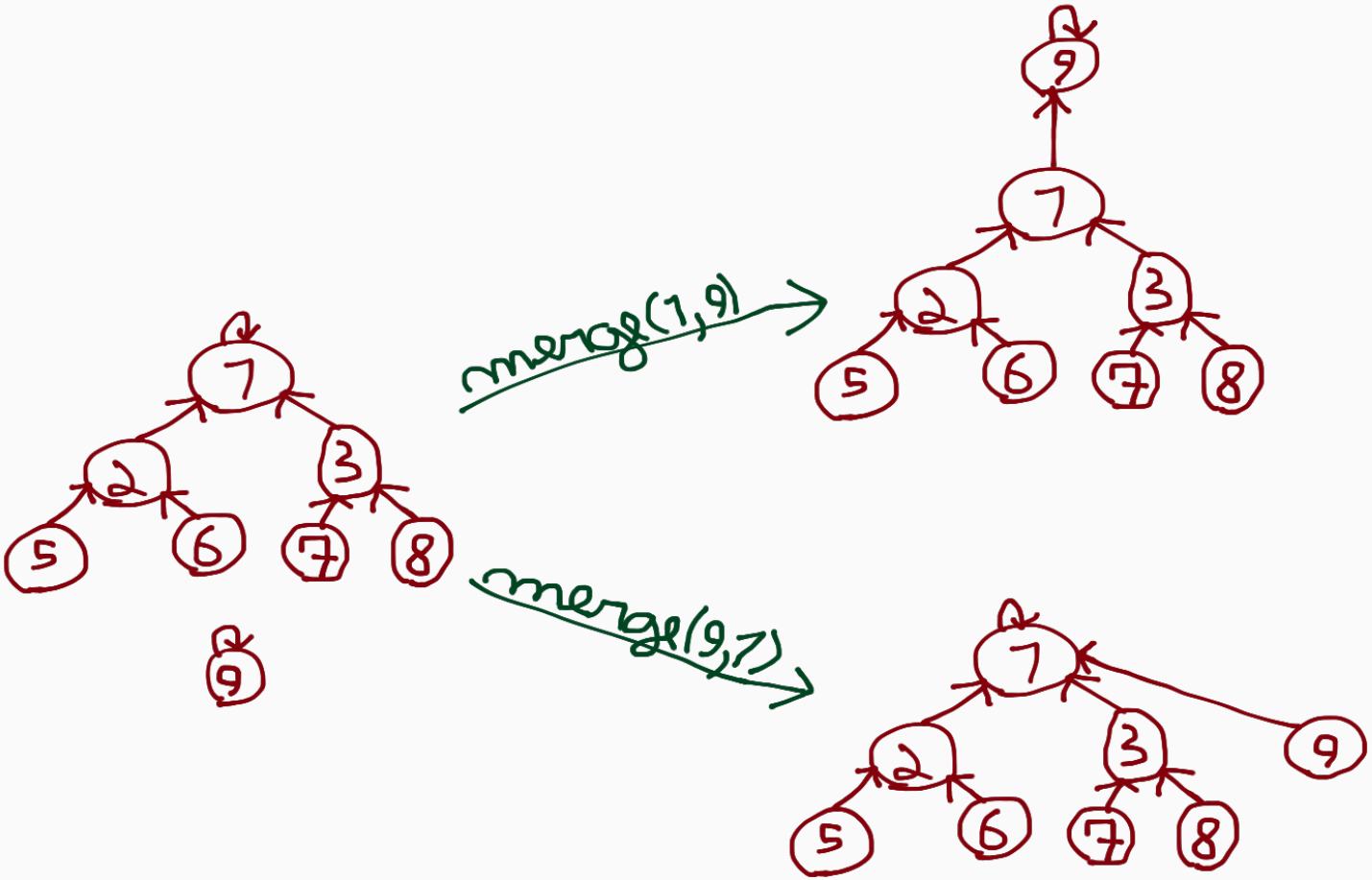
- Unir conjuntos do menor para o maior
  - É necessário manter o tamanho de cada conjunto v
- Intuição: minimizar a profundidade dos conjuntos
- Tempo por operação: $O(\log n)$
  - Provaremos em breve

União por tamanho (implementação)

```
int pai[N] , peso[N];
void init(){
    for(int i = 0 ; i < N ; i ++)
        pai[i] = i , peso[i] = 1;
}
int find(int x){
    if(pai[x] == x)
        return x;
    return find(pai[x]);
}
void une(int x , int y){
    x = find(x) , y = find(y);
    if(x == y)
        return;
    if(peso[x] > peso[y]) swap(x,y);
```

União por tamanho

Exemplo



União por tamanho

Análise de complexidade

- Se X o tamanho da menor componente, e Y o tamanho da maior componente:
  - $2 \cdot X \leq X + Y$
- Considere um valor de altura de cada nó, inicialmente em 0.
  - A cada `une(X,Y)`, considerando $|X| < |Y|$, a altura de cada nó em X aumenta em 1
- Uma componente só pode ser a "menor" no une $log_2(N)$ vezes, a cada passo que ela é a menor, o tamanho dela vai pelo menos dobrar.
- Concluimos que a maior altura de um nó é $log_2(N)$, a complexidade do `find(i)` agora é $O(log_2(N))$, e o $une(a, b)$ continua $O(1)$.

Inverso de Ackermann

Teoria

- Se usarmos tanto a compressão por caminho quanto a união por tamanho simultaneamente, a complexidade fica $O((n+m) \cdot \alpha(n))$ amortizado
- $\alpha(m)$ é a inversa da função de Ackermann, que tem crescimento absurdamente devagar, podemos considerar que qualquer valor computavel tem $\alpha(m)$ menor que 4.
- A prova é bastante teórica e foi omitida.

Problema: Network Breakdown Link

- Temos inicialmente um grafo com $n \leq 10^5$ nós e $m \leq 10^5$ arestas e $k \leq 10^5$ operações
- Cada operação escolhemos uma aresta e removemos ela do grafo
- Diga o número de componentes conectadas após cada operação.
- Outro problema similar. Link

Problema: Reformas em Königsberg

- Temos inicialmente um grafo com $n \leq 10^5$ nós e $m \leq 10^5$ arestas, além disso $q \leq 10^5$ operações do tipo:
  - Adicionar uma aresta entre a e b (é garantido que essa aresta não existia antes)
- Após cada pergunta queremos saber o número de pontes no grafo
  - Podemos resolver offline! (com alguma estrutura..) Link
  - Podemos resolver online? Link

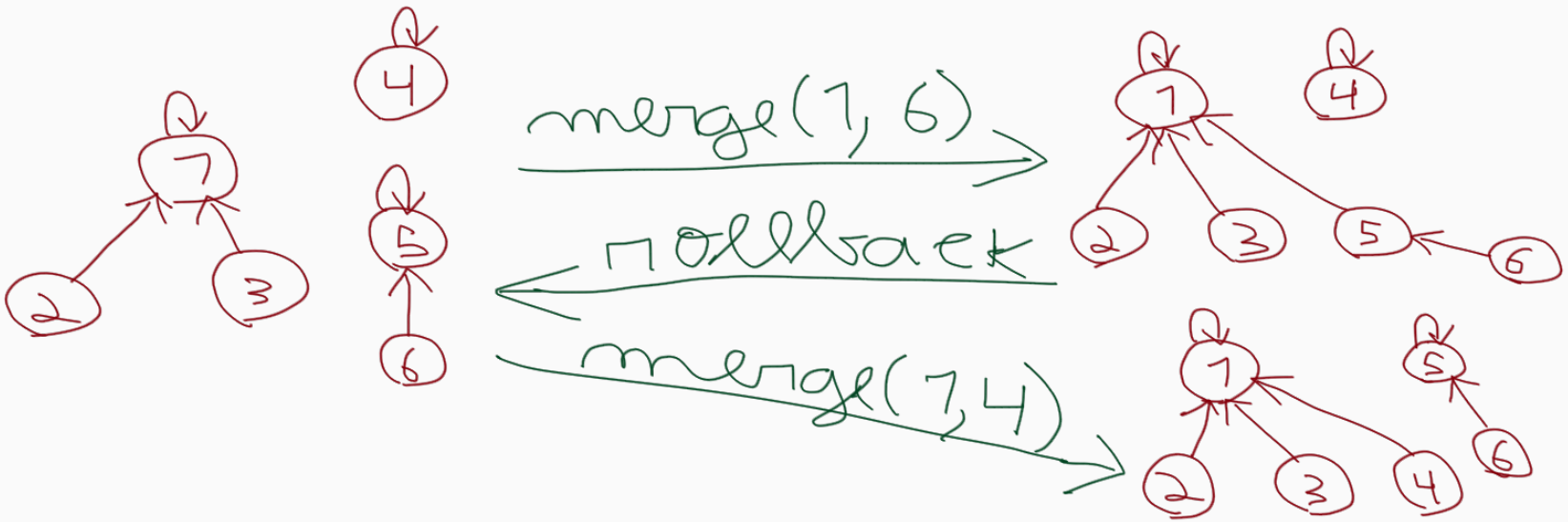
Problema: teste seu código

Teste sua implementação de DSU aqui.

Rollback

Rollback com *união de tamanho*

- Podemos desfazer as uniões imediatamente anteriores



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

Rollback com *união por tamanho*

Implementação (parte 1)

```
stack< pair<int,int> > old_pai , old_peso;
int find(int x){
    if(pai[x] == x)
        return x;
    return find(pai[x]);
}
void une(int u , int v){
    u = find(u) , v = find(v);
    if(u == v)
        return ;
    if(peso[u] > peso[v]) swap(u,v);
    old.emplace(u , pai[u]), old_peso.empalce(v, peso[v]);
    pai[u] = v , peso[v] += peso[u];
}
```

Rollback com *união por tamanho*

Implementação (parte 2)

```
void rollback(){
    pai[old_pai.top().first] = old_pai.top().second;
    peso[old_peso.top().first] = old_peso.top().second;
    old_pai.pop();
    old_peso.pop();
}
```

Rollback com *compressão de caminho*

Teoria

- Amortização quebra!
  - Nem sempre temos $log_2(n)$ operações no find, podemos ter $O(n)$.
  - Possivelmente podemos ter criado n novos snapshots para desfazer...

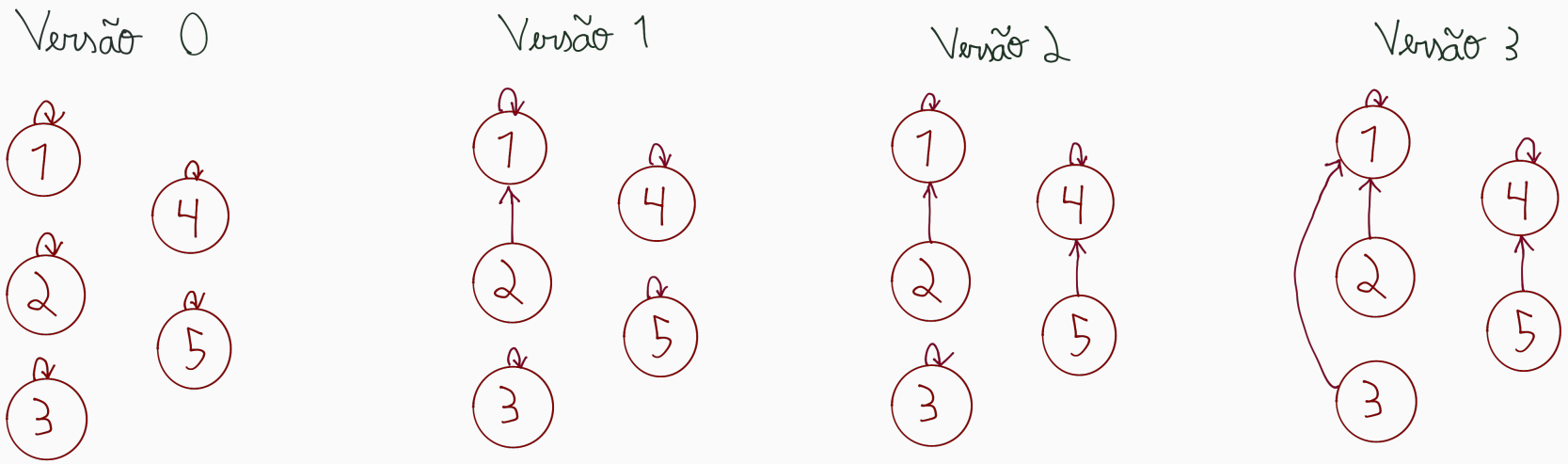
Problema GERALD07 LINK

- Temos um grafo com $n \leq 2 \cdot 10^5$ vértices e um vetor de $m \leq 2 \cdot 10^5$ arestas, além disso $q \leq 2 \cdot 10^5$ querys, cada query temos um par $1 \leq L_i \leq R_i \leq M$.
  - Para cada query responda quantas componentes conexas tem o grafo considerando apenas as arestas de L_i até R_i
- Decomponha as querys em baldes de tamanho $\sqrt{(M)}$, para cada par de baldes (i, j) construa a DSU com arestas de i até j, use rollback para lidar com blocos parcialmente cobertos.
- Tente resolver com Mo's + Rollback!

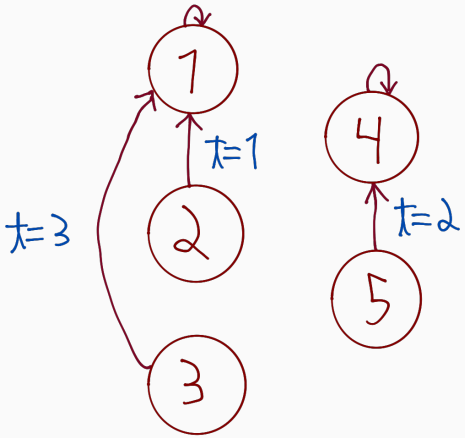
Persistência parcial (e outras histórias..)

Persistência parcial com union-by-size

- Podemos consultar versões anteriores da estrutura

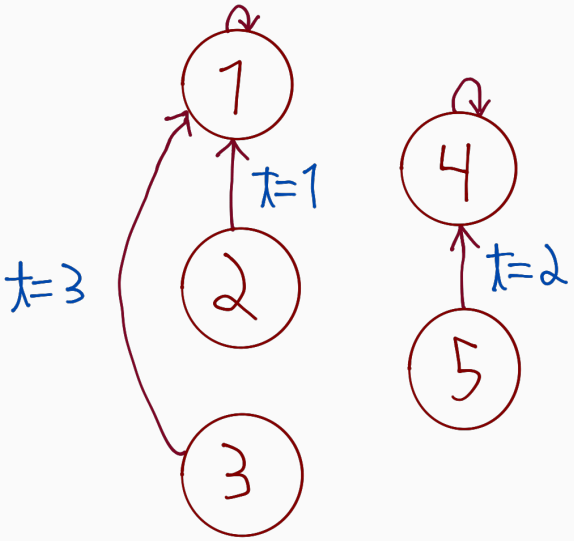


- Ideia: anotar o tempo de cada ligação



Persistência parcial com union-by-size

- Ideia: anotar o tempo de cada ligação



- Operações suportadas:
  - `merge(a, b)`: une os conjuntos aos quais a e b pertencem
  - `find(a, t)`: retorna o líder do conjunto de a no tempo t
  - `same(a, b)`: verifica o menor tempo para a e b pertencerem ao mesmo conjunto.

Persistência parcial com union-by-size

Implementação (parte 1)

```
int find(int u){
    if(pai[u] == u)
        return u;
    return find(pai[u]);
}
void merge(int u , int v , int tempo){
    u = find(u) , v = find(v);
    if(u == v)
        return;
    if(peso[u] > peso[v])
        swap(u,v);
    pai[u] = v , peso[v] += peso[u] , link[u] = tempo;
}
```

Persistência parcial com union-by-size

Implementação (parte 2)

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

Persistência parcial com union-by-size

Implementação (parte 3)

```
void same(int u , int v){
    if(find(u) != find(v))
        return -1; // nunca vão tá conectados
    int tempo = -1; // inicializa como negativo
    while(u != v){
        if(c[u] > c[v])
            swap(u,v);
        tempo = max(tempo , c[u]); // pega a menor opção.
         u = pai[u];
    }
    return tempo;
}
```

Bipartição online

- Queremos lidar com as seguintes operações:
  - Adiciona uma aresta (a,b) **é garantido que essa aresta não existia antes**.
  - Perguntar se a componente que o vértice `A` está é bipartida.

Como fazer? Existe alguma solução $(n + m)\alpha(n)$?

Problemas

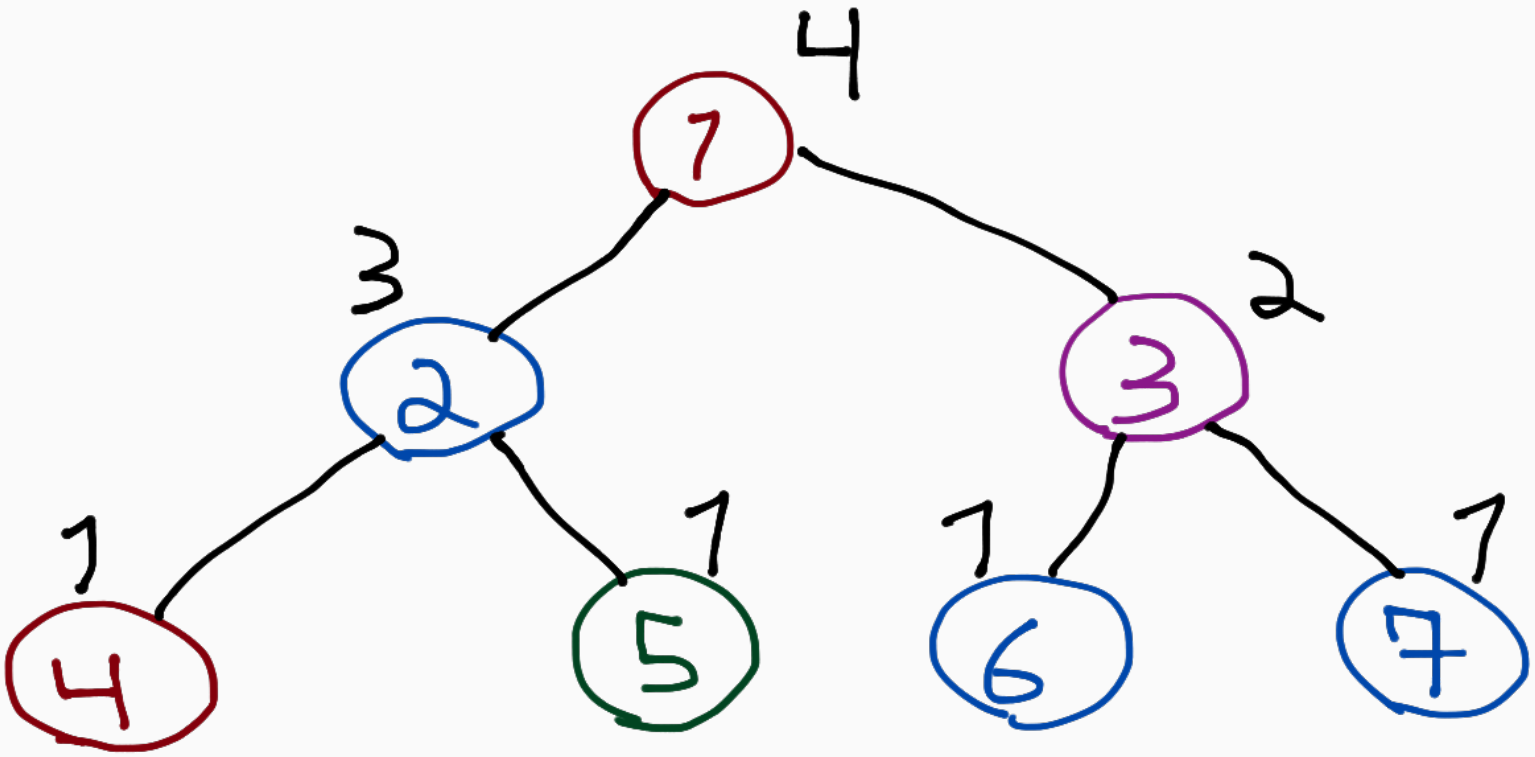
- Problema de persistência parcial (CSES). Link
- DSU + xor path (COCI). Link
- DSU + manhattan (Seletiva IOI 2016). Link

Small-to-large

Cores em subárvores

Descrição do problema

- Entrada: árvore na qual os vértices possuem cores
- Objetivo: encontrar a quantidade de cores **distintas** 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(cor[v]);
    for(auto w : filhos[v]){
        auto p = dfs(w);
        colors.insert(begin(p) , end(p));
    }
}
```

- 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[N];
void dfs(int v){
    colors[v].insert(cor[v]);
    for(auto w : filhos[v]){
        dfs(w);
        if(colors[w].size() > colors[v].size())
            swap(colors[w] , colors[v]); // O(1)
        colors[v].insert(begin(colors[w]),end(colors[w]));
    }
    ans[v] = colors[v].size();
}
```

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

Problemas de Small-To-Large

- Experience (CF EDU) Link.
- Problema apresentado Link.
- IOI 2011 Race Link.
- CSES Two Stacking Sort (Desafio**) Link.

Min(x,n-x) trick

Problema Somos Amigos

- Temos $n \leq 10^5$ vértices ordenados em uma linha de $1..n$
- Temos $m \leq 3 \cdot 10^5$ arestas direcionadas (u, v), não existem duas arestas iguais, nem self-loop.
- Queremos achar dois inteiros $L \leq R$ tal que para todo vértice $L \leq i \leq R$, existe algum $L \leq j \leq R$ tal que i é amigo de j. Além disso, queremos maximizar R - L + 1.

Solução ingênua:

- Passar por todos os L, R possíveis e checar a validade da resposta
  - $O(n^2)$

Solução (não tão) ingênua:

- Note que L, R é uma região contígua, vamos usar isso.
- Divisão e conquista, solve(L,R) retorna a maior resposta possivel considerando os nós de L até R.
  - Basta acharmos algum nó que não cumpre a condição considerando L e R atual, podemos quebrar em dois intervalos disjunto.
- $solve(L, R) = solve(l, x - 1), solve(x + 1, r)$
- Complexidade: $T(n) = T(n - x) + T(x) + O(n)$
- Complexidade: $O(n^2)$. Exemplo??

Solução:

- Intuitivamente, queremos pegar um x rapidamente, ou dividir o intervalo em partes boas.
- Suponha que temos um ponteiro no final e no começo e vamos movimentar ambos simultaneamente
  - Na primeira vez que encontrarmos alguem inválido, já podemos parar de mexer nos ponteiros.
- Complexidade: $T(n) = T(n-x) + T(x) + O(min(x, n-x))$
- Complexidade: $O(n \log(n))$.
- Intuição: Estamos desfazendo small-to-large no ponto certo.
- Prova: Análise da recorrência.

Problemas:

- Somos amigos Link.
- A Story of One Country Codeforces Link.
- Tree Querys Online Link.