

# *Union-Find Disjoint Sets*

## Definição e Implementação

---

Prof. Edson Alves – UnB/FCTE

1. Definição
2. Implementação
3. Otimizações e Aplicações

## Definição

---

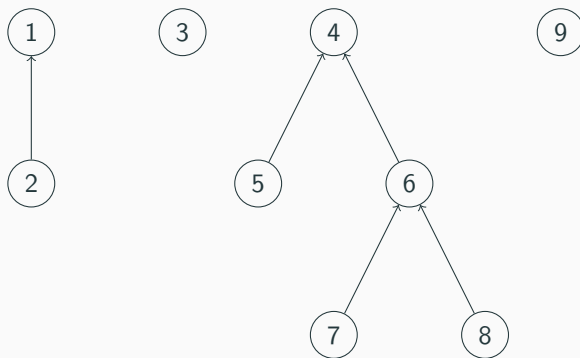
## Definição

*Union-find disjoint set* (UFDS) é uma estrutura de dados que mantém uma coleção de conjuntos  $\{S_1, S_2, \dots, S_N\}$  disjuntos, isto é,  $S_i \cap S_j = \emptyset$  se  $i \neq j$ . Há duas operações básicas, com complexidade  $O(\log N)$ : a **união** (`union_set(A, B)`) de dois conjuntos disjuntos e a identificação do **representante** da união de conjuntos que o conjunto  $S$  pertence (`find_set(S)`).

# Características da UFDS

- A UFDS foi proposta em 1971 por J. D. Hopcroft e J. D. Ullman
- Ela é composta por uma floresta de árvores
- Cada árvore representa uma união de subconjuntos
- A raiz de cada árvore é o representante da união
- Cada nó da árvore representa um dos conjuntos que compõem a união
- Conjuntos representados por árvores distintas são disjuntos
- Duas árvores podem ser unidas tornando a raiz de uma delas filha da raiz da outra
- Se a árvore com o menor número de elementos for incorporada na árvore com o maior número de elementos, a união terá complexidade  $O(\log N)$

## Visualização de uma UFDS



# Implementação

---

# Construtor da UFDS

- Considere que cada um dos  $N$  conjuntos a serem representados na UFDS sejam identificados pelos inteiros de 1 a  $N$
- A UFDS tem dois vetores membros: *size* e *ps*
- *size*[ $i$ ] corresponde ao número de nós da árvore que tem  $i$  como raiz
- *ps*[ $i$ ] é o pai de  $i$  na árvore que ele está contido
- Como inicialmente todos os conjuntos são disjuntos, temos *size*[ $i$ ] = 1 e *ps*[ $i$ ] =  $i$ , para  $i = 1, 2, \dots, N$
- Observe que, ao contrário da convenção das árvores em computação, o pai do nó raiz é ele próprio, o que simplifica a implementação



# Implementação do construtor da UFDS

```
1 #ifndef UNION_FIND_H
2 #define UNION_FIND_H
3
4 #include <vector>
5 #include <numeric>
6
7 class UFDS
8 {
9 private:
10     std::vector<int> size, ps;
11
12 public:
13     UFDS(int N) : size(N + 1, 1), ps(N + 1)
14     {
15         // ps = {0, 1, 2, 3, ..., N}
16         std::iota(ps.begin(), ps.end(), 0);
17     }
```

## Identificação dos representantes

- O método `find_set(x)` retorna o representante (raiz) da árvore onde  $x$  se encontra
- Para isso, basta seguir a cadeia de pais até localizar um nó cujo pai é ele mesmo
- Se a união for baseada no tamanho das árvores, o tamanho de cada árvore tende a  $O(\log N)$ , de modo que este método também tem complexidade  $O(\log N)$
- O método `same_set(x, y)` retorna verdadeiro se  $x$  e  $y$  pertencem a mesma árvore, ou falso, caso contrário
- A implementação é simples: basta confrontar os representantes de cada conjunto

# Implementação da identificação de representantes

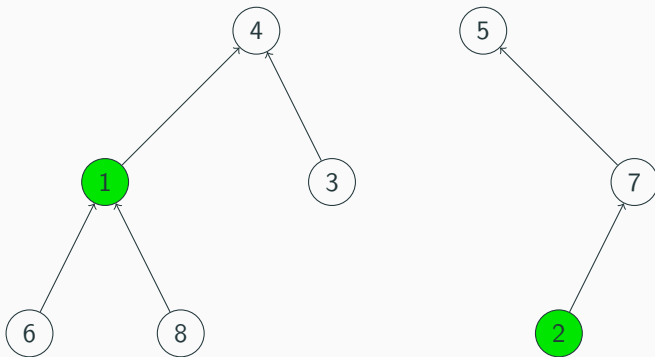
```
19  int find_set(int x) const
20  {
21      return x == ps[x] ? x : find_set(ps[x]);
22  }
23
24  bool same_set(int x, int y)
25  {
26      return find_set(x) == find_set(y);
27  }
```

# União de conjunto disjuntos

- O método `union_set(x, y)` une as árvores onde  $x$  e  $y$  estão localizados
- Se  $x$  e  $y$  já estão na mesma árvore, nada deve ser feito
- Caso contrário considere, sem perda de generalidade, que o número de nós da árvore que contém  $x$  seja maior ou igual que o número de nós da árvore que contém  $y$
- Neste caso, a raiz da árvore de  $x$  passa ser o pai da raiz da árvore de  $y$
- Como ambas raízes devem ser localizadas previamente, a complexidade deste método também é  $O(\log N)$

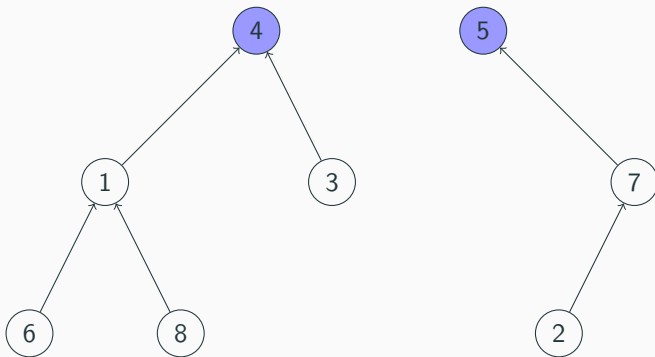
## Visualização da união de árvores

`union_set(1, 2)`



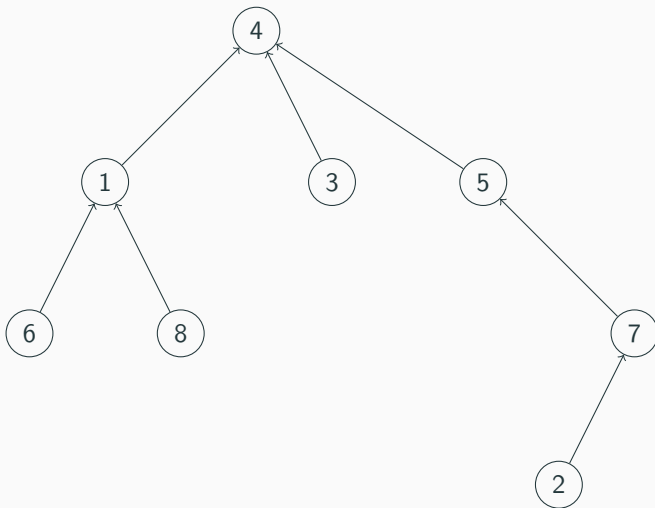
## Visualização da união de árvores

`union_set(1, 2)`



## Visualização da união de árvores

`union_set(1, 2)`



# Implementação da união de conjuntos disjuntos

```
29 void union_set(int x, int y)
30 {
31     if (same_set(x, y))
32         return;
33
34     int p = find_set(x);
35     int q = find_set(y);
36
37     if (size[p] < size[q])
38         std::swap(p, q);
39
40     ps[q] = p;
41     size[p] += size[q];
42 }
43 };
44
45 #endif
```



# Otimizações e Aplicações

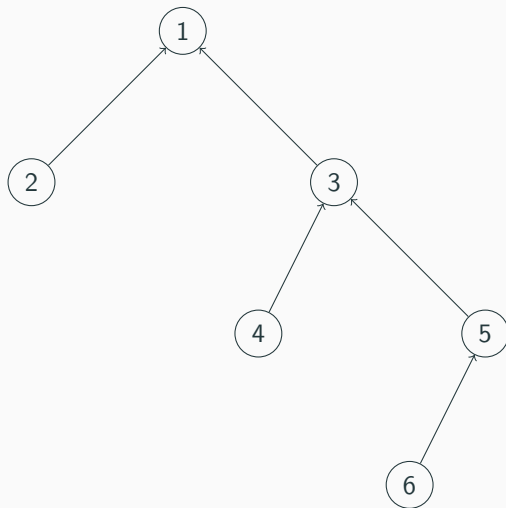
---

## Compressão de caminho

- O método `find_set(x)` pode ser modificado para que, ao percorrer o caminho de  $x$  até a raiz, os nós intermediários passem a ter a raiz da árvore como pai
- Esta técnica é conhecida como compressão de caminho
- A medida que o método for invocado para diferentes valores de  $x$ , as árvores tendem a ter apenas dois níveis
- Usada em conjunto com a união por tamanho (ou por ranqueamento), a complexidade amortizada de ambas operações (união e identificação de representante) passa a ser  $O(\alpha(n))$ , onde  $\alpha(n)$  é a função inversa de Ackermann
- Para qualquer  $n$  representável no universo físico,  $\alpha(n) < 5$ , de modo que as operações tem, na essência, complexidade constante

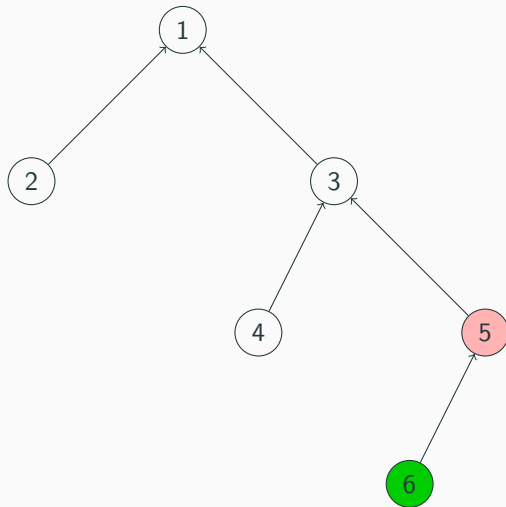
# Visualização da compressão de caminho

`find_set(6)`



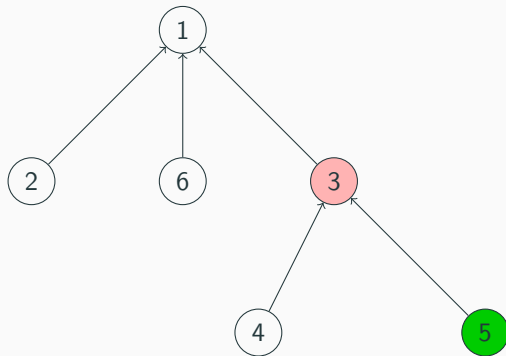
# Visualização da compressão de caminho

`find_set(6)`



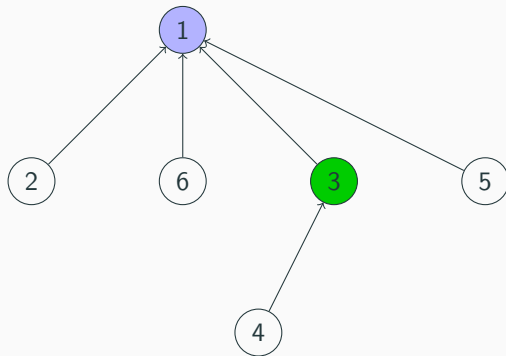
## Visualização da compressão de caminho

`find_set(6)`



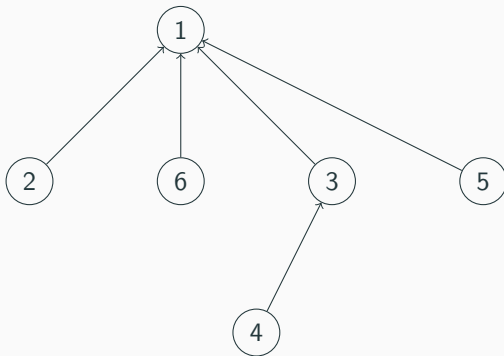
## Visualização da compressão de caminho

`find_set(6)`



## Visualização da compressão de caminho

`find_set(6) = 1`



# Implementação da compressão de caminho

```
1  int find_set(int x)
2  {
3      return x == ps[x] ? x : (ps[x] = find_set(ps[x]));
4  }
```



- A UFDS pode ser aplicada em vários contextos
- Uma aplicação comum é a implementação do algoritmo de Kruskall para determinar a árvore mínima geradora (MST) de um grafo
- De modo geral, ela pode ser usada para identificar os componentes conectados de um grafo não-direcionado
- Por fim, seja  $R$  uma relação de equivalência
- Se  $(a, b) \in R$  (isto é,  $a$  está relacionado a  $b$ ), então `same_set(a, b)` retorna verdadeiro

1. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
2. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.
3. **SEDGEWICK**, Robert. *Algorithms, 4th Edition*, 976 pgs, Addison-Wesley Professional, 2011.
4. **SKIENA**, Steven S.; **REVILLA**, Miguel A. *Programming Challenges*, 2003.
5. Wikipédia. [Disjoint-set data structure](#), acesso em 05/03/2020.