MAC0323 Algoritmos e Estruturas de Dados II

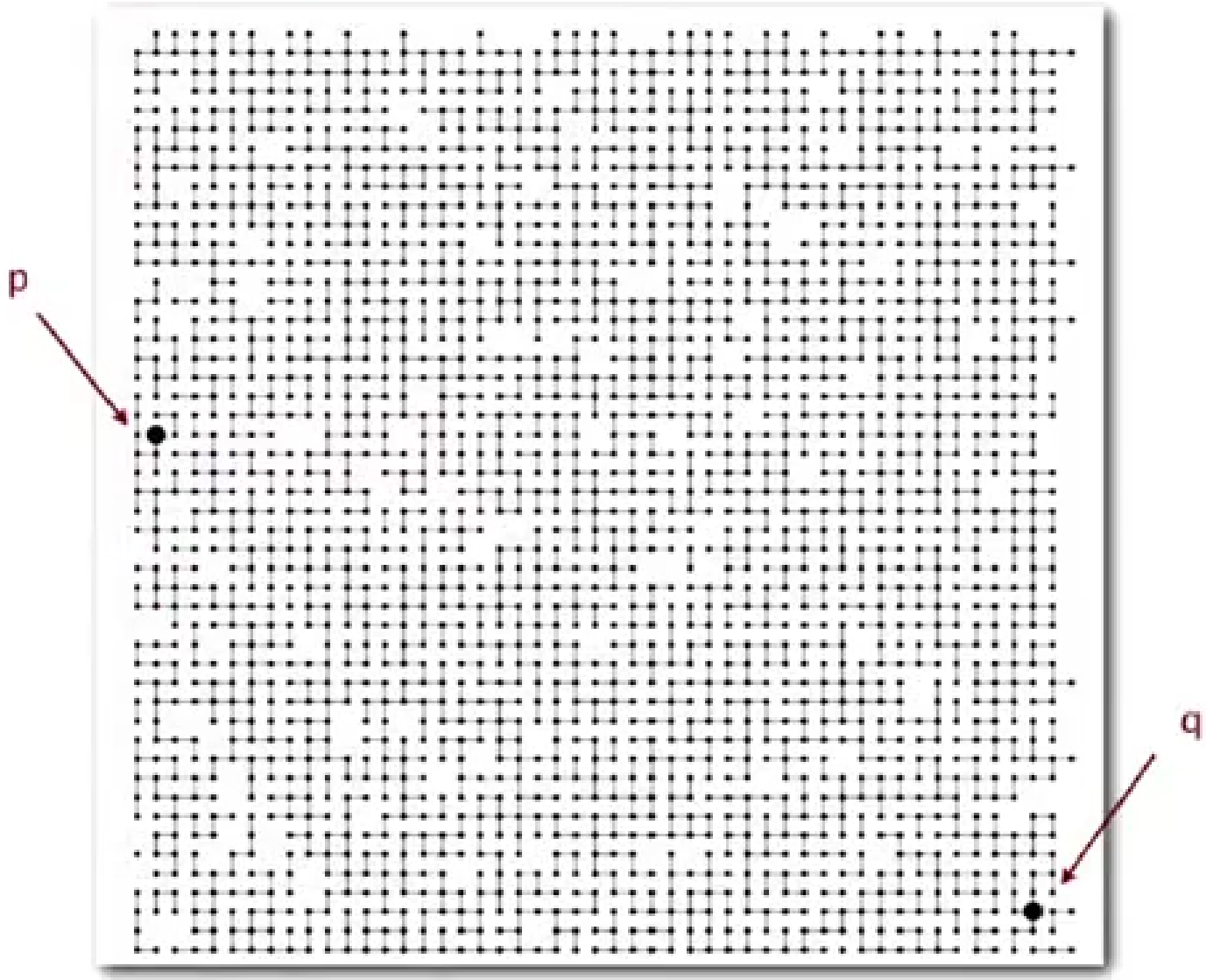
Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 3

Problema: p e q estão ligados?

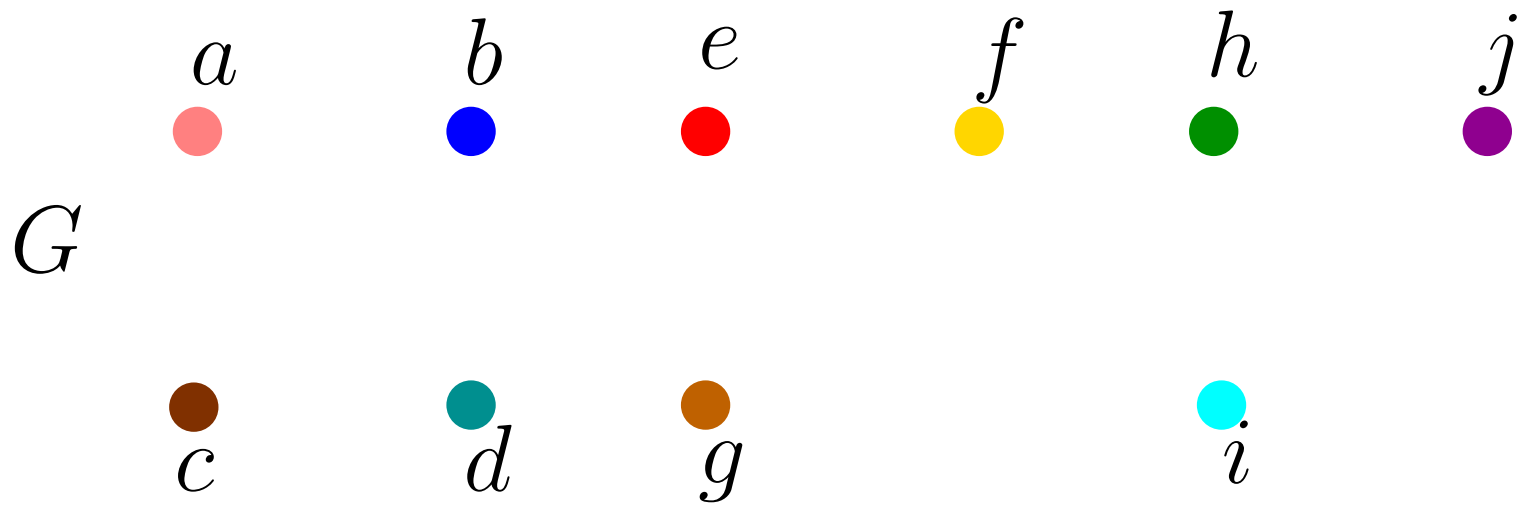


Fonte: algs4

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

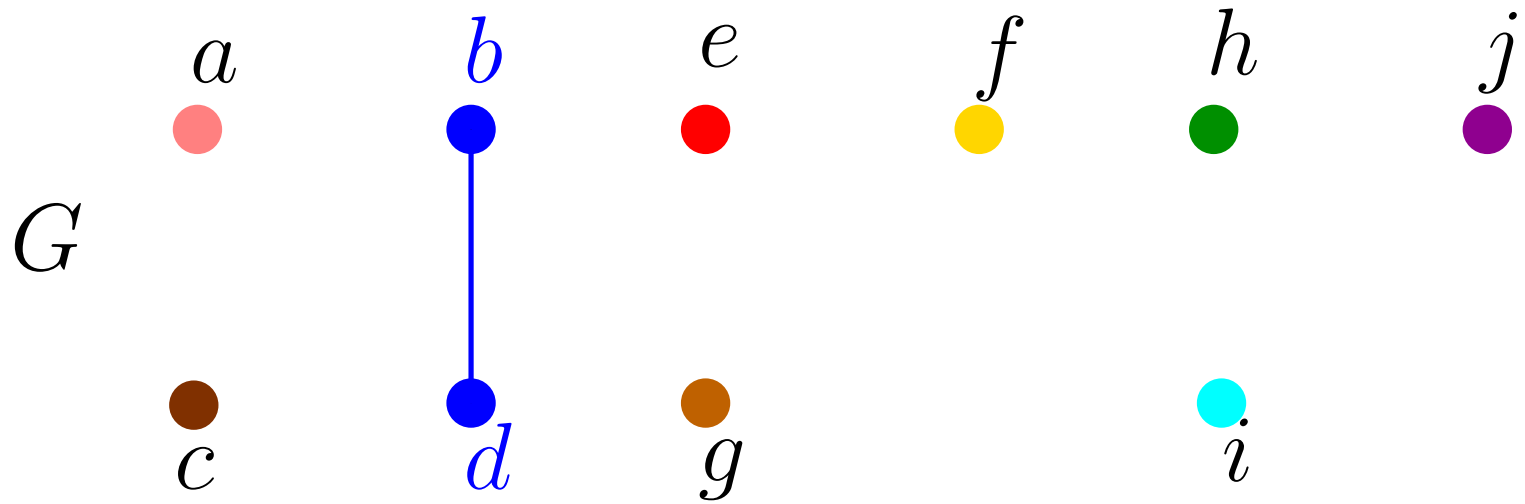


aresta	componentes
	$\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

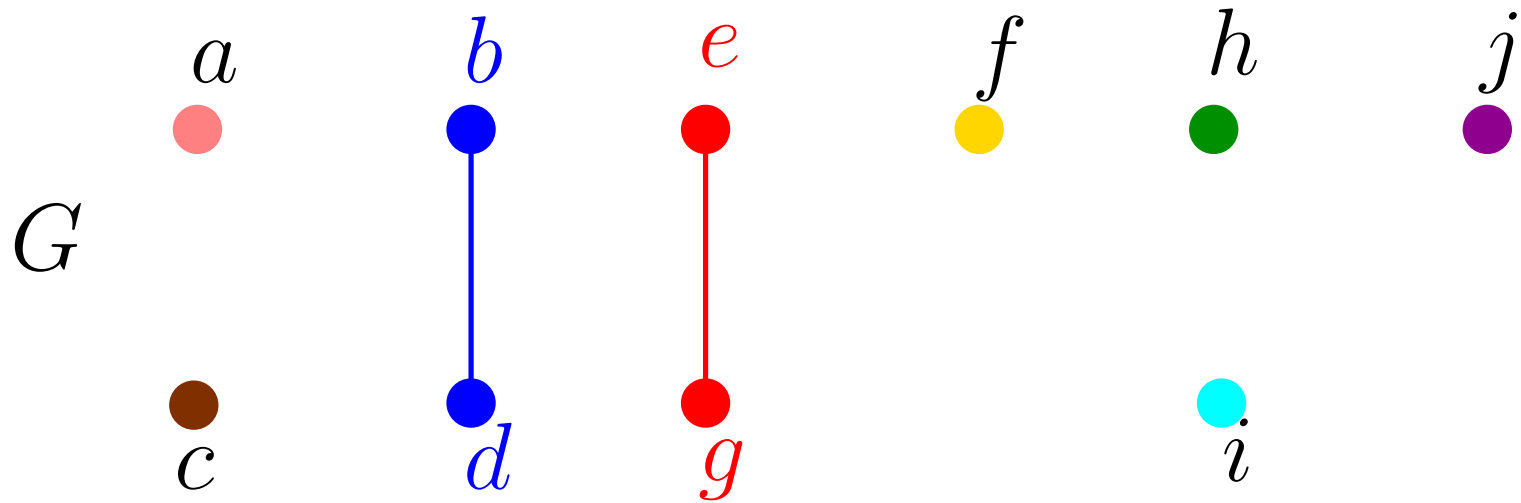


aresta	componentes								
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

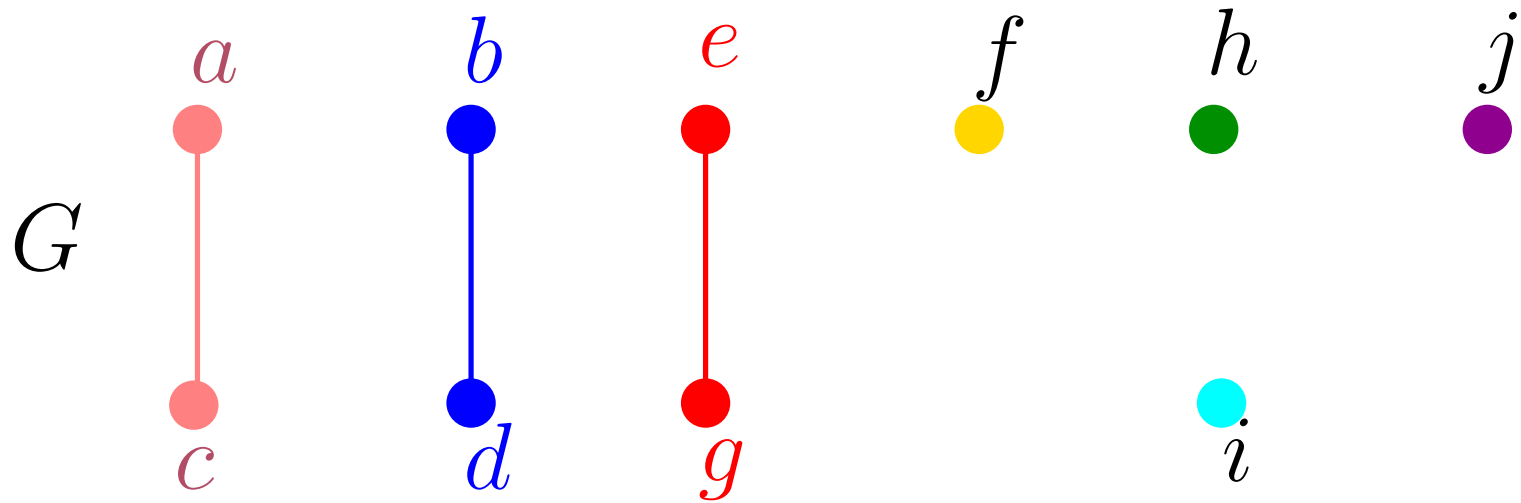


aresta	componentes
(e, g)	$\{a\}$ $\{b, d\}$ $\{c\}$ $\{e, g\}$ $\{f\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

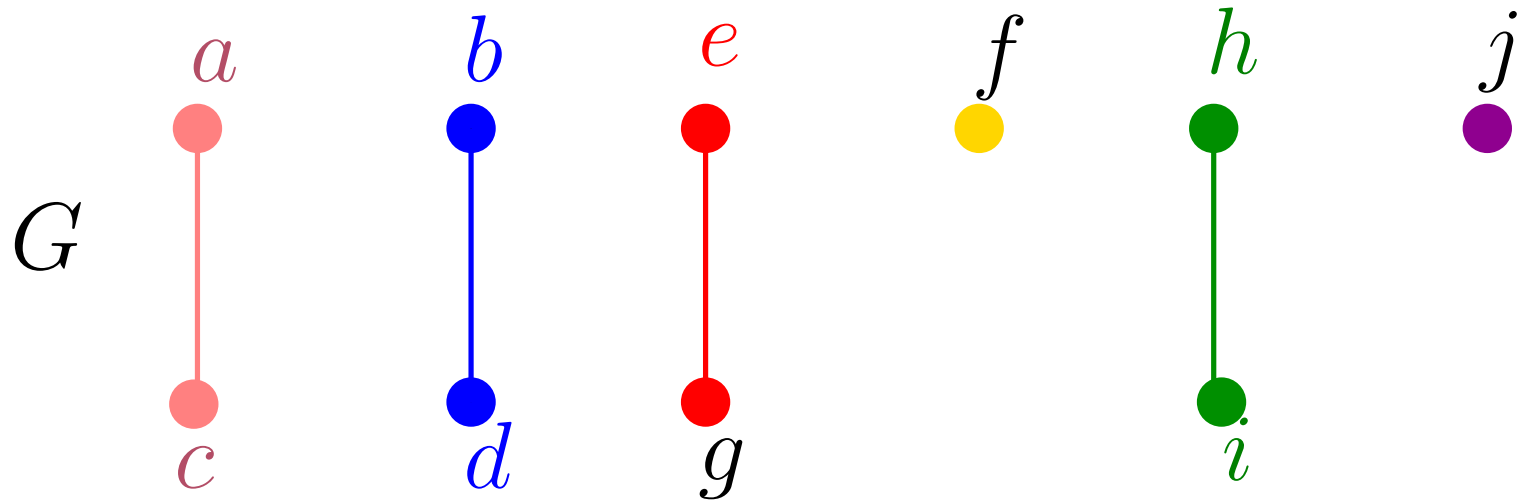


aresta	componentes
(a, c)	$\{a, c\}$ $\{b, d\}$ $\{e, g\}$ $\{f\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

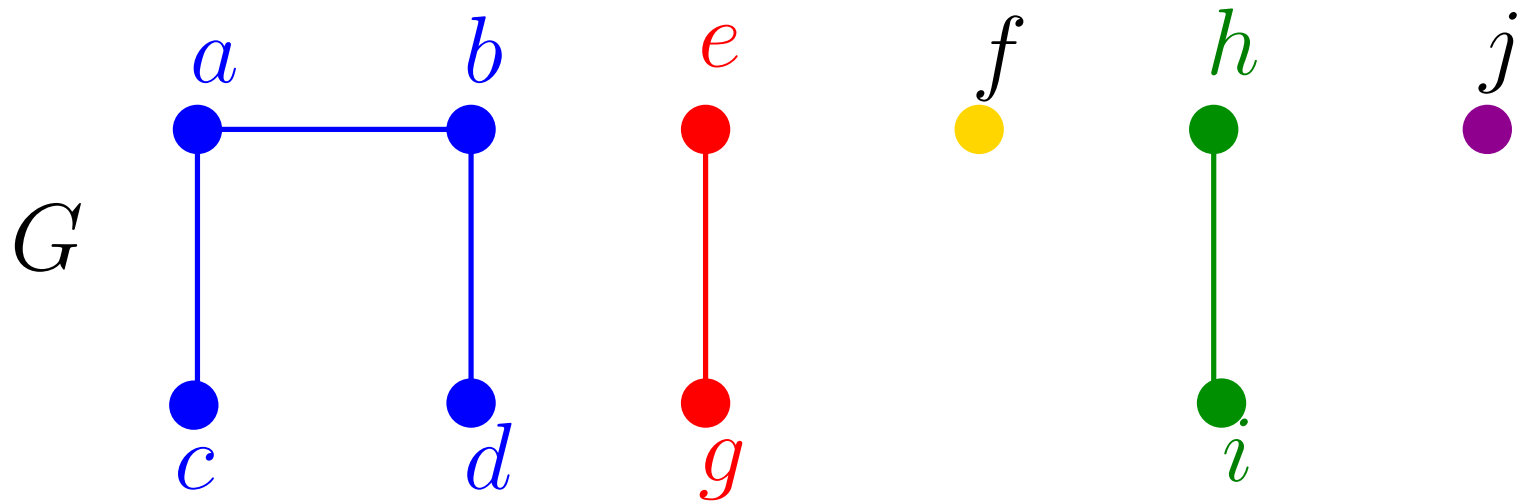


aresta	componentes
(h, i)	$\{a, c\}$ $\{b, d\}$ $\{e, g\}$ $\{f\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

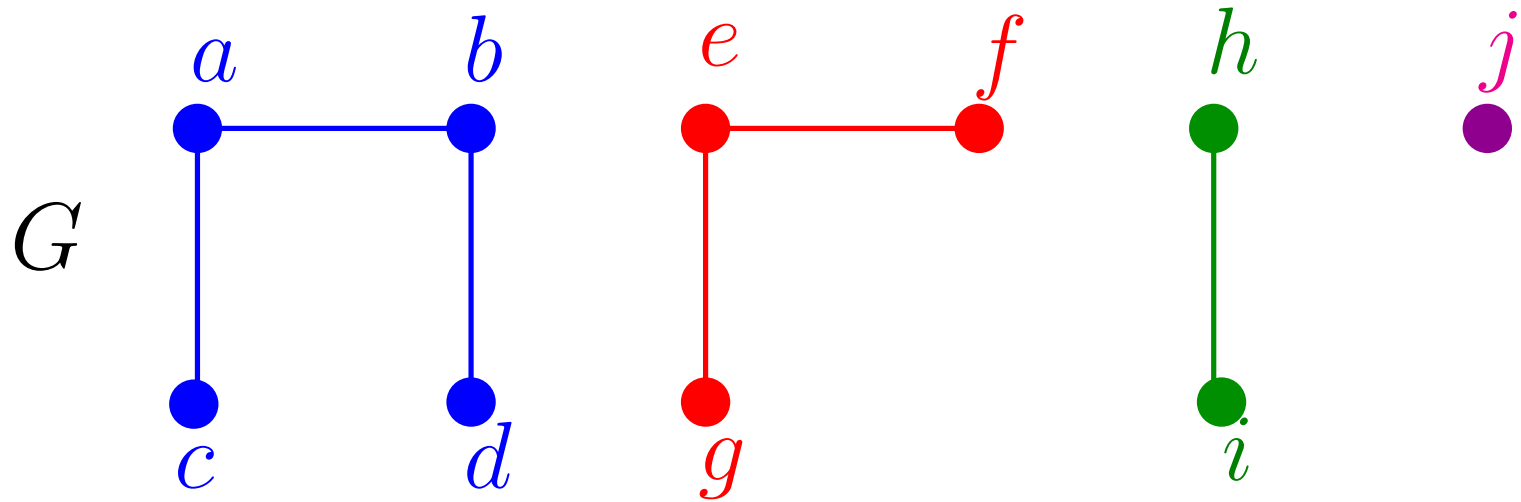


aresta	componentes
(a, b)	$\{a, b, c, d\}$ $\{e, g\}$ $\{f\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

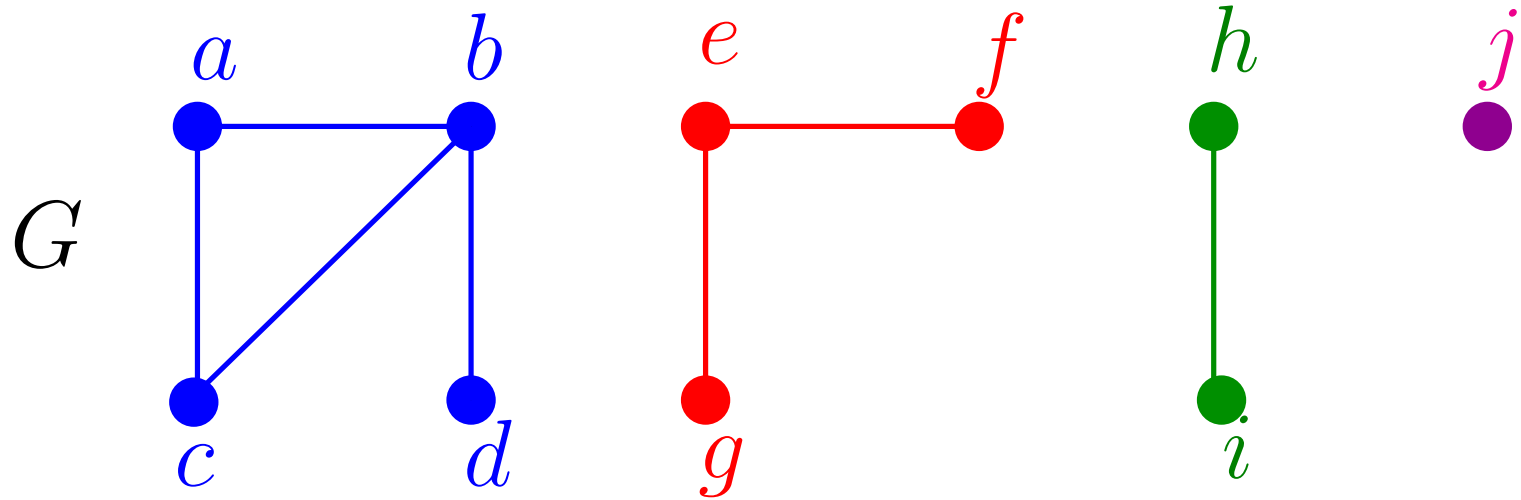


aresta	componentes
(e, f)	$\{a, b, c, d\}$ $\{e, f, g\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico



aresta	componentes
(b, c)	$\{a, b, c, d\}$ $\{e, f, g\}$ $\{h, i\}$ $\{j\}$

Union-find

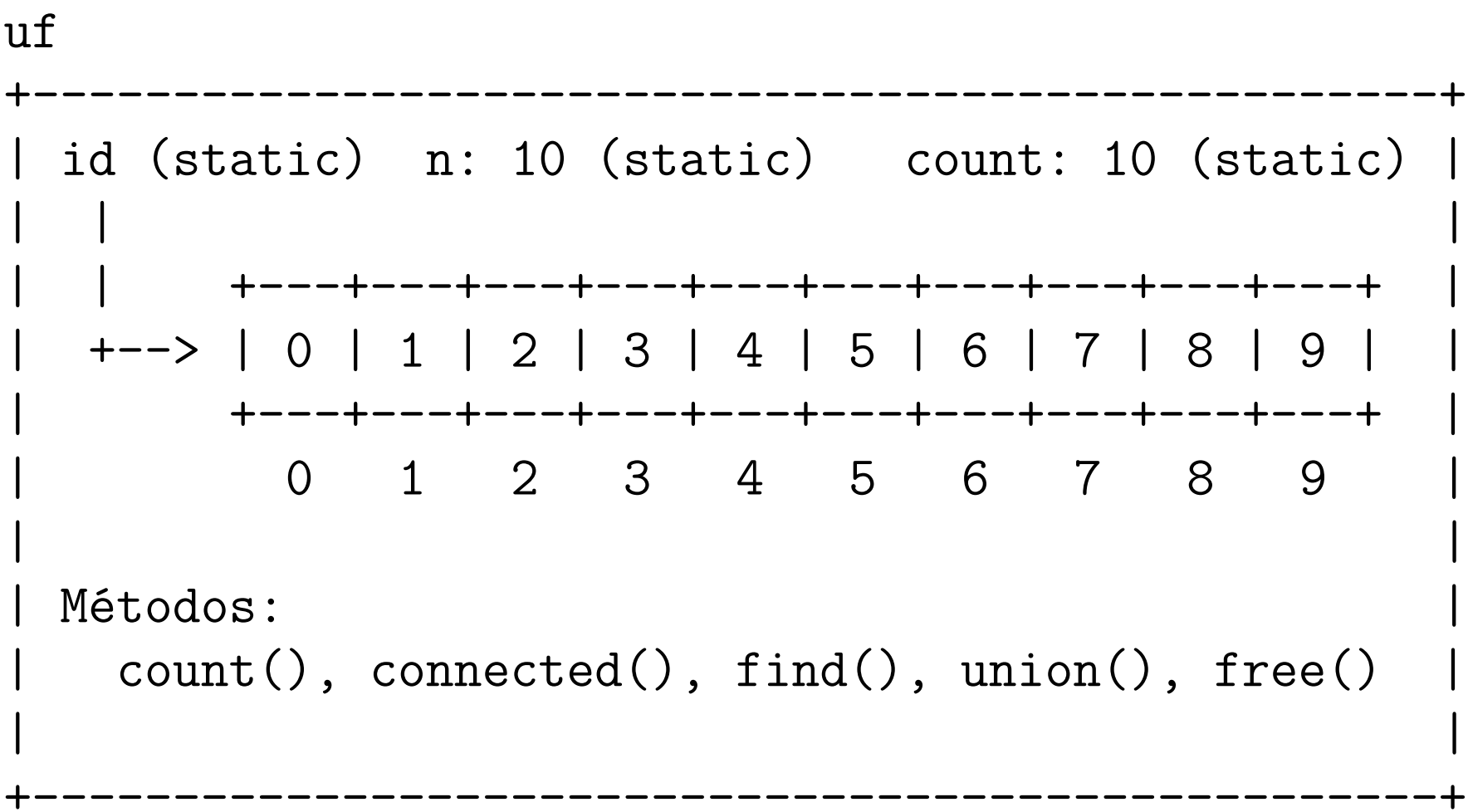
1.5 Case Study: Union-Find

Interface uf.h

Arquivo	uf.h	
`void`	`ufInit(int n)`	inicializa `n` sites com nomes inteiros `0,...,n-1`
`void`	`ufUnion(int p, int q)`	acrescenta ligação entre `p` e `q`
`int`	`ufFind(int p)`	retorna `id` do componente de `p`
`bool`	`ufConnected(int p,int q)`	`true` se `p` e `q` estão no mesmo componente
`int`	`ufCount()`	número de componentes
`void`	`ufFree()`	destroi a ED

QuickFindUF

`ufInit`(10)



Implementação: QuickFindUF.c

```
void ufInit(int n) {
   int i;
   id = mallocSafe(n * sizeof(int));
   for (i = 0; i < n; i++) id[i] = i;
   nUF = count = n;
}
/* retorna o id do componente de p */
int ufFind(int p) {
   return id[p];
}
/* p e q estão no mesmo componente?  */
bool ufConnected(int p, int q) {
   return ufFind(p) == ufFind(q);
}
```

Implementação: QuickFindUF.c

```
void ufUnion(int p, int q) {
   int i, pl = ufFind(p), ql = ufFind(q);
   if (pl == ql) return;
   for (i = 0; i < nUF; i++)
      if (id[i] == pl) id[i] = ql;
   count--;
}
```

```
int ufCount() {
   return count;
}
```

```
void ufFree() {
   free(id);
   id = NULL;
   count = 0;
}
```

Consumo de tempo

`ufInit(n)`	$\Theta(n)$
`ufUnion(p, q)`	$O(n)$
`ufFind(p)`	$\Theta(1)$

Uma sequência de m operações
pode consumir tempo $\Theta(m^2)$ no pior caso.

Consumo de tempo amortizado
de cada operação é $O(m)$.

Hmm. Em `ufUnion()`, seria razoável alterarmos
o menor número possível de posições do vetor `id`.
Para isso precisamos saber qual conjunto
tem o menor número de itens...

Experimentos

```
% time client < tinyUF.txt          (n = 10)
2 components
0.003seg

% time client < mediumUF.txt        (n = 625)
3 components
0.006seg

% time client < largeUF.txt     (n = 1000000)
:-(
```

Fonte: Pinterest

AULA 4

QuickUnionUF

A **ideia** é trocar o indicador `id[]` do componente por um indicador do `pai[]` do elemento.

Por sua vez, se `p` é um elemento,

`pai[pai[p]]` é o avô de `p`
`pai[pai[pai[p]]]` é o bisavô de `p`,
`pai[pai[pai[pai[p]]]]` é o tataravô,
...

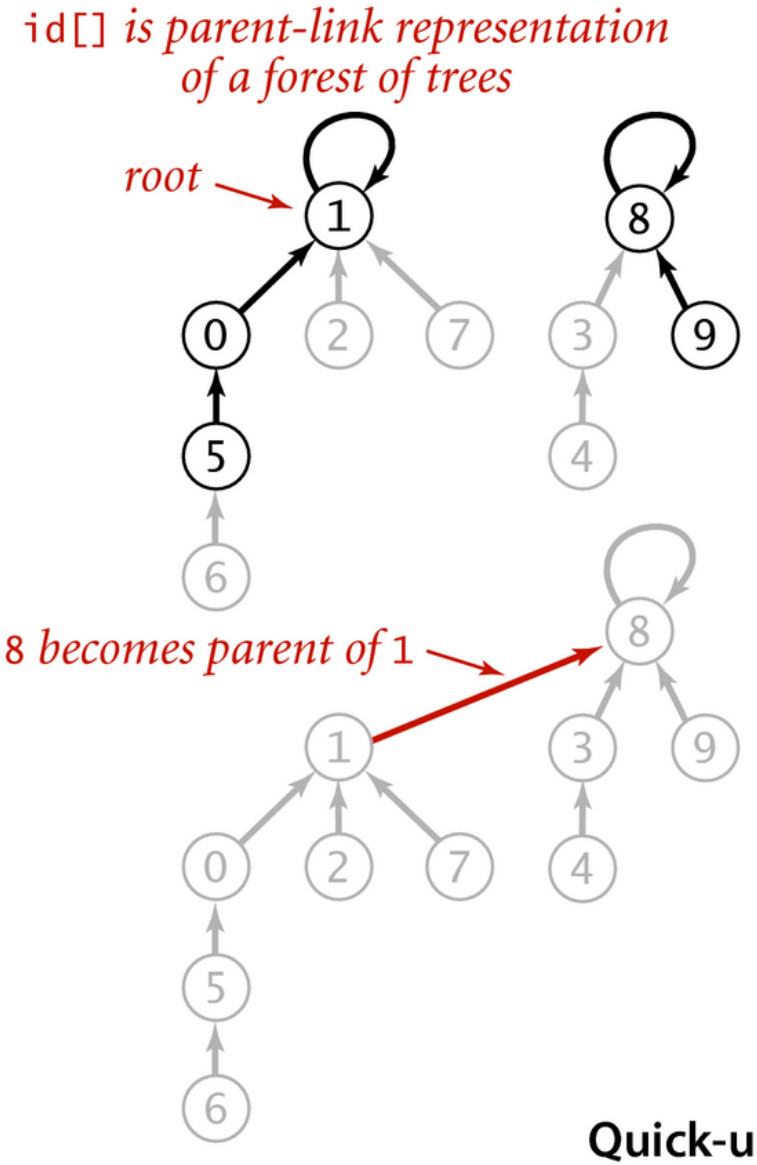
QuickUnionUF

O **representante** ou **nome** de um componente será o elemento que é o pai de si mesmo.

É intuitivo representarmos a estrutura através de um conjunto de árvores disjuntas (= **floresta**) onde as raizes das árvores são os sítios `p` tais que

```
p == pai[p].
```

Estrutura *disjoint-set forest*



find has to follow links to the root

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	3	0	5	1	8	8

`find(5)` *is* `id[id[id[5]]]`

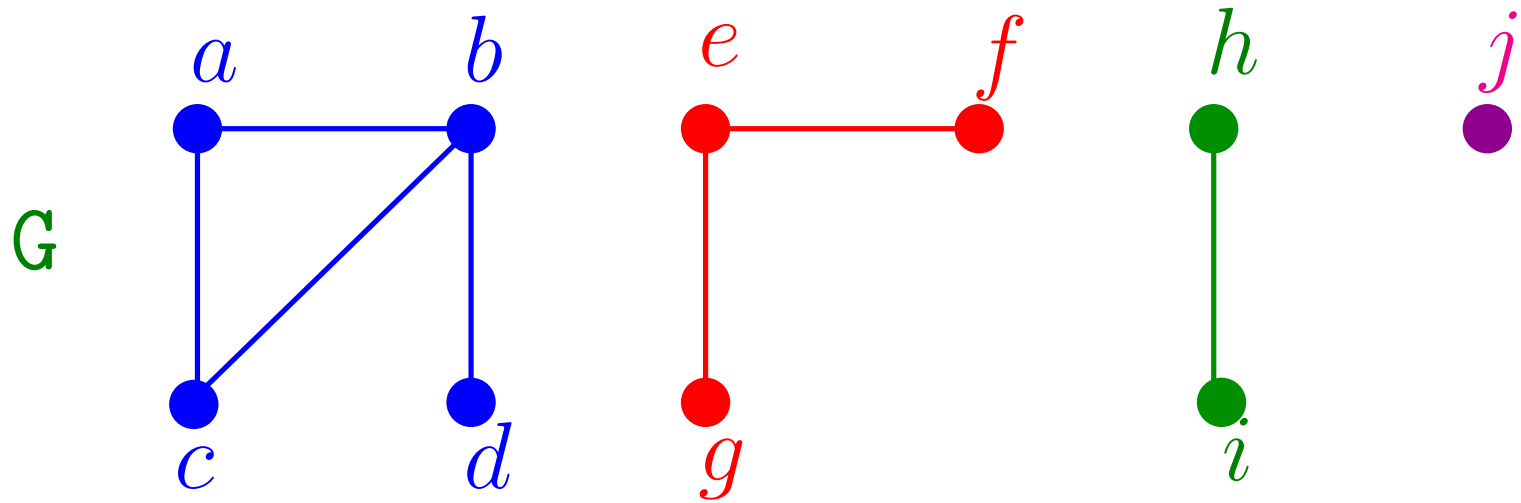
`find(9)` *is* `id[id[9]]`

union changes just one link

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	3	0	5	1	8	8
		1	8	1	8	3	0	5	1	8	8

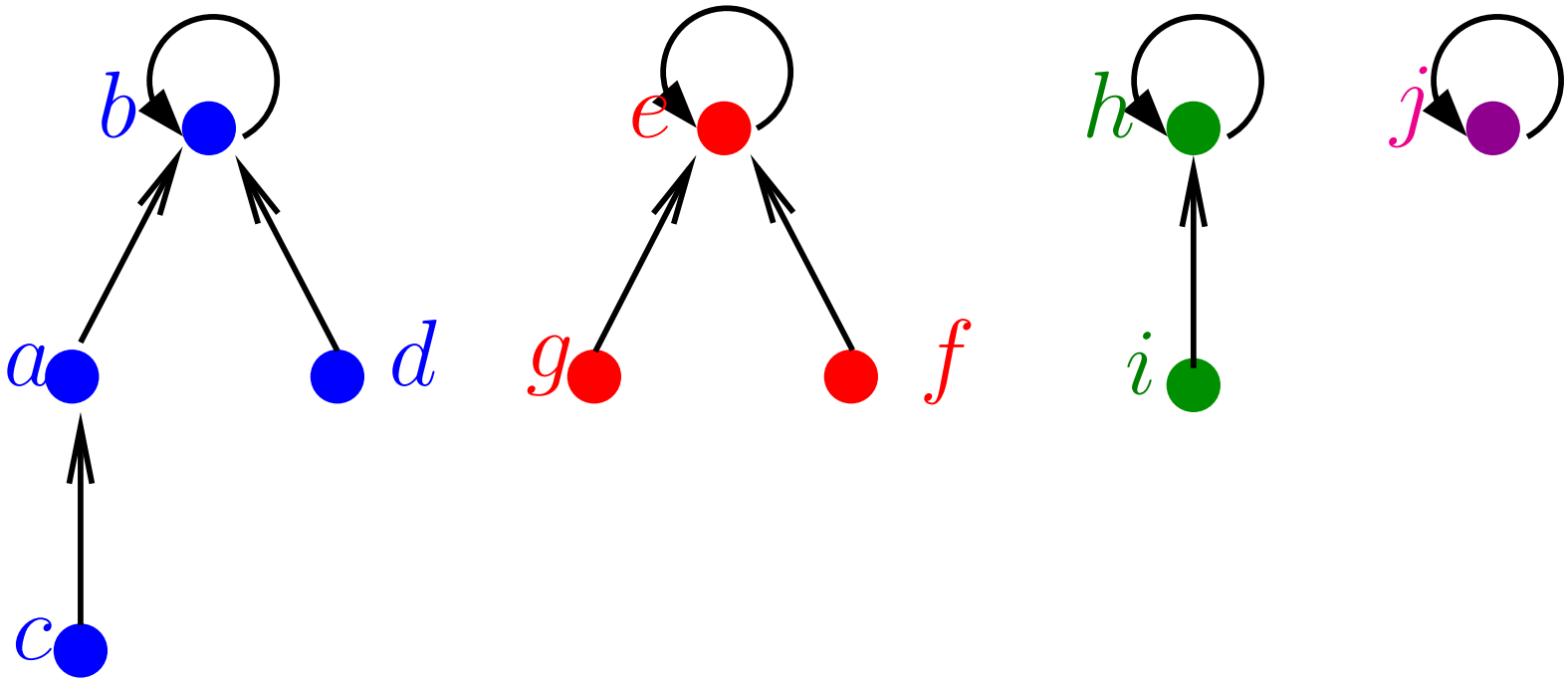
Quick-union overview

Estrutura *disjoint-set forest*



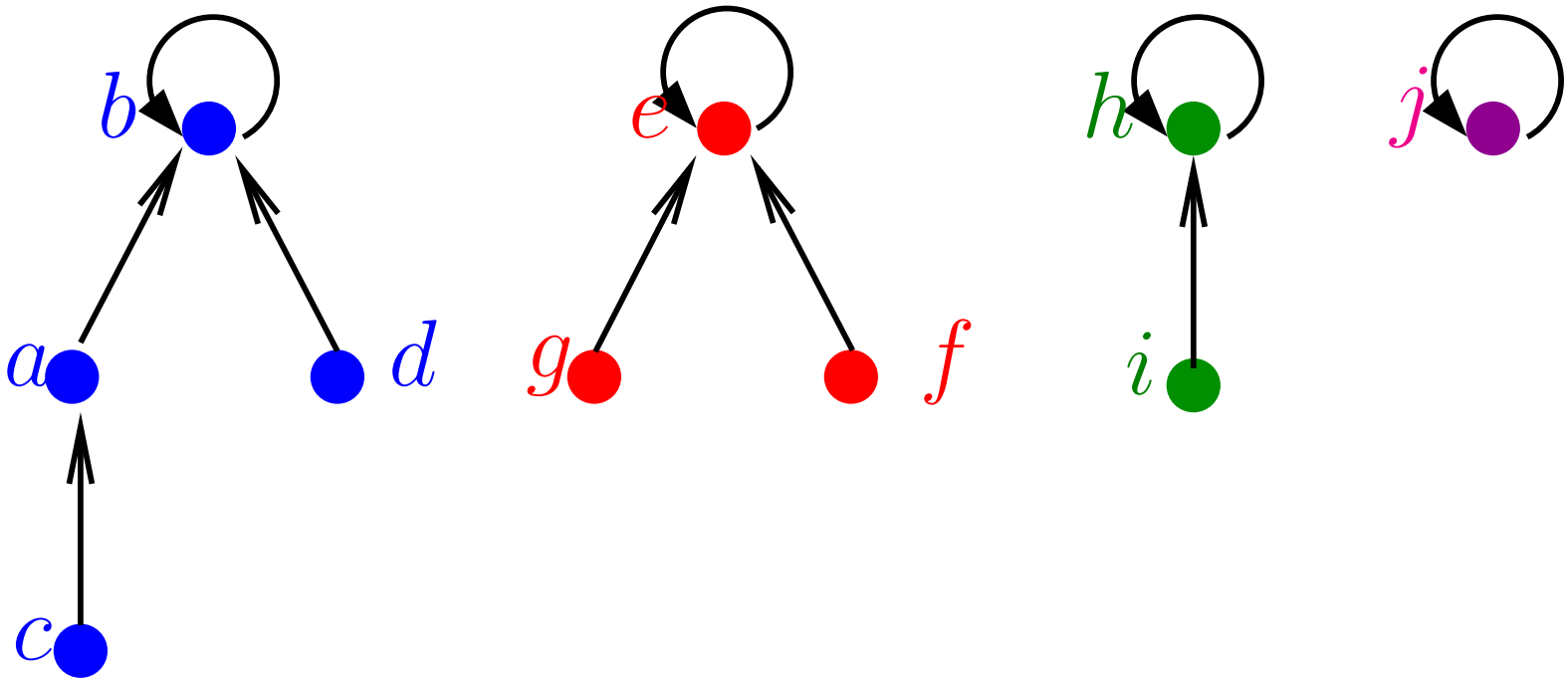
- cada conjunto tem uma `raiz`, que é o seu representante
- cada nó `p` tem um `pai`
- `pai[p]` = `p` se e só se `p` é uma raiz

Estrutura *disjoint-set forest*



- cada conjunto tem uma `raiz`
- cada nó `p` tem um `pai`
- `pai[p]` $=$ `p` se e só se `p` é uma raiz

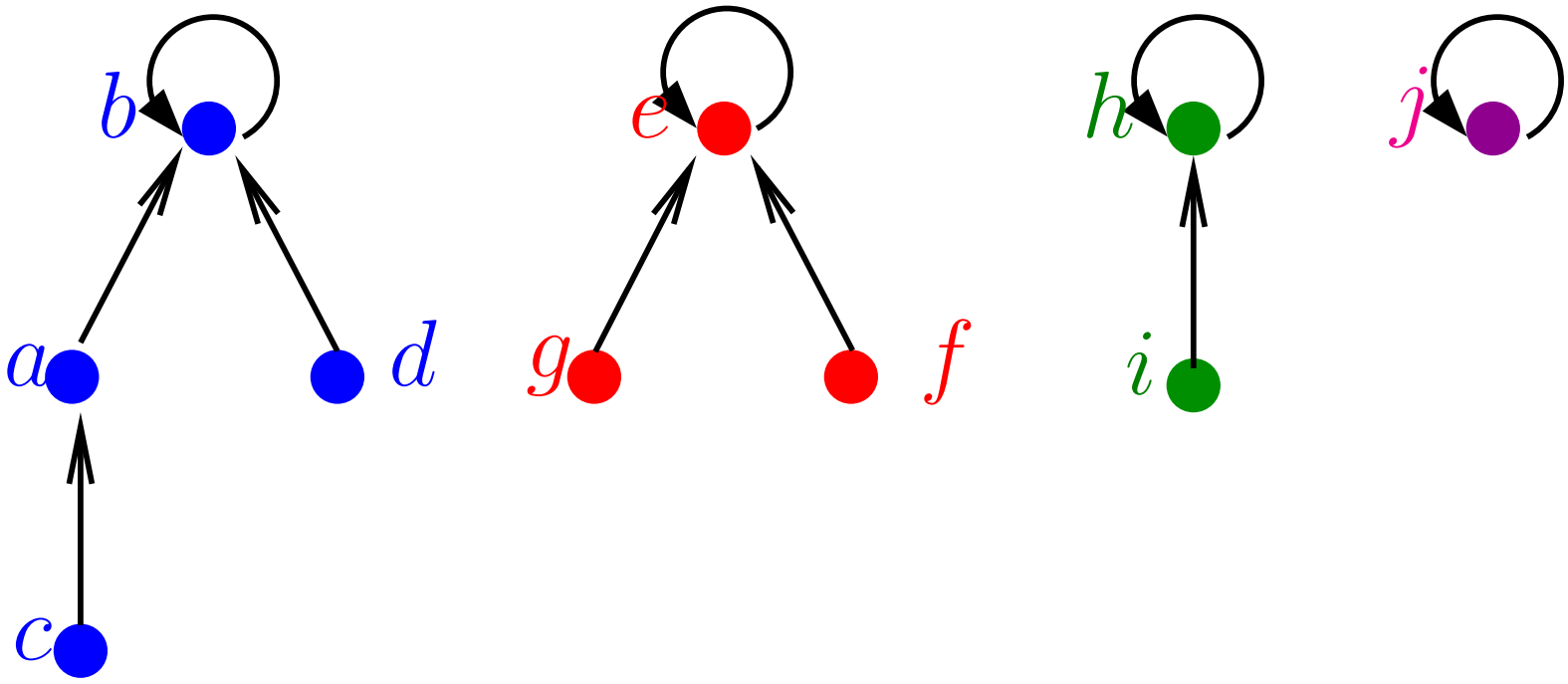
MakeSet e FindSet



MAKESET(p)

1 pai[p] ← p

MakeSet e FindSet



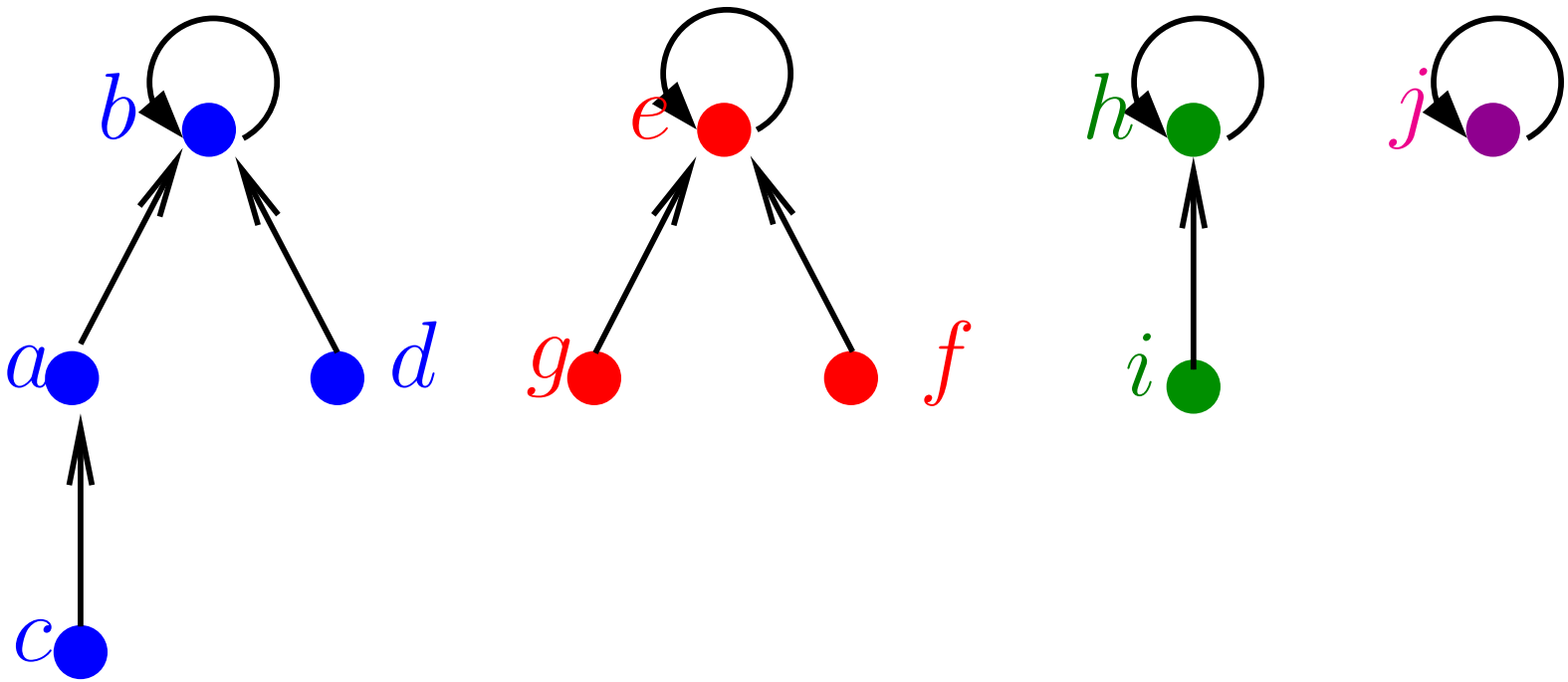
MAKESET(p)

```
1    pai[p] ← p
```

FINDSET(p)

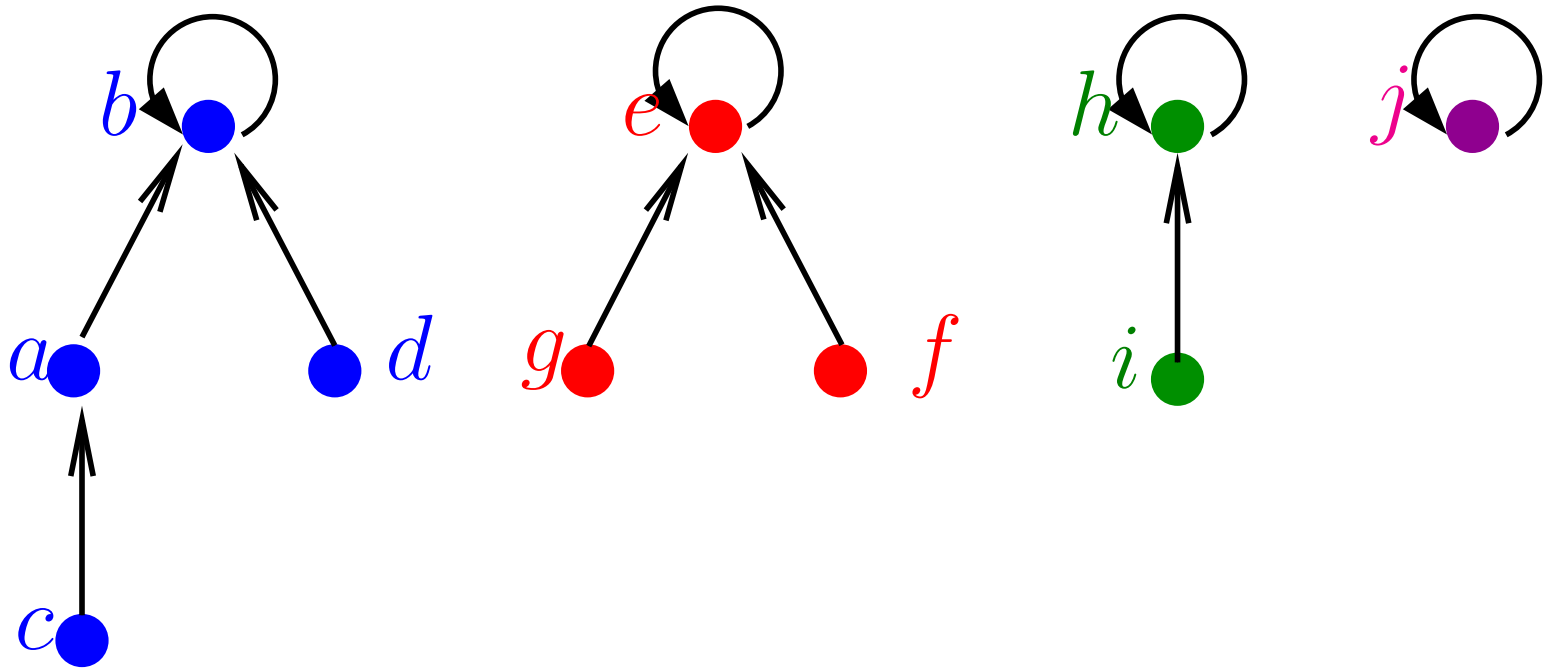
```
1    enquanto pai[p] ≠ p faça
2        p ← pai[p]
3    devolva p
```

FindSet recursivo




```
FindSet(p)
1    se pai[p] = p
2        então devolva p
3        senão devolva FindSet (pai[p])
```

Union



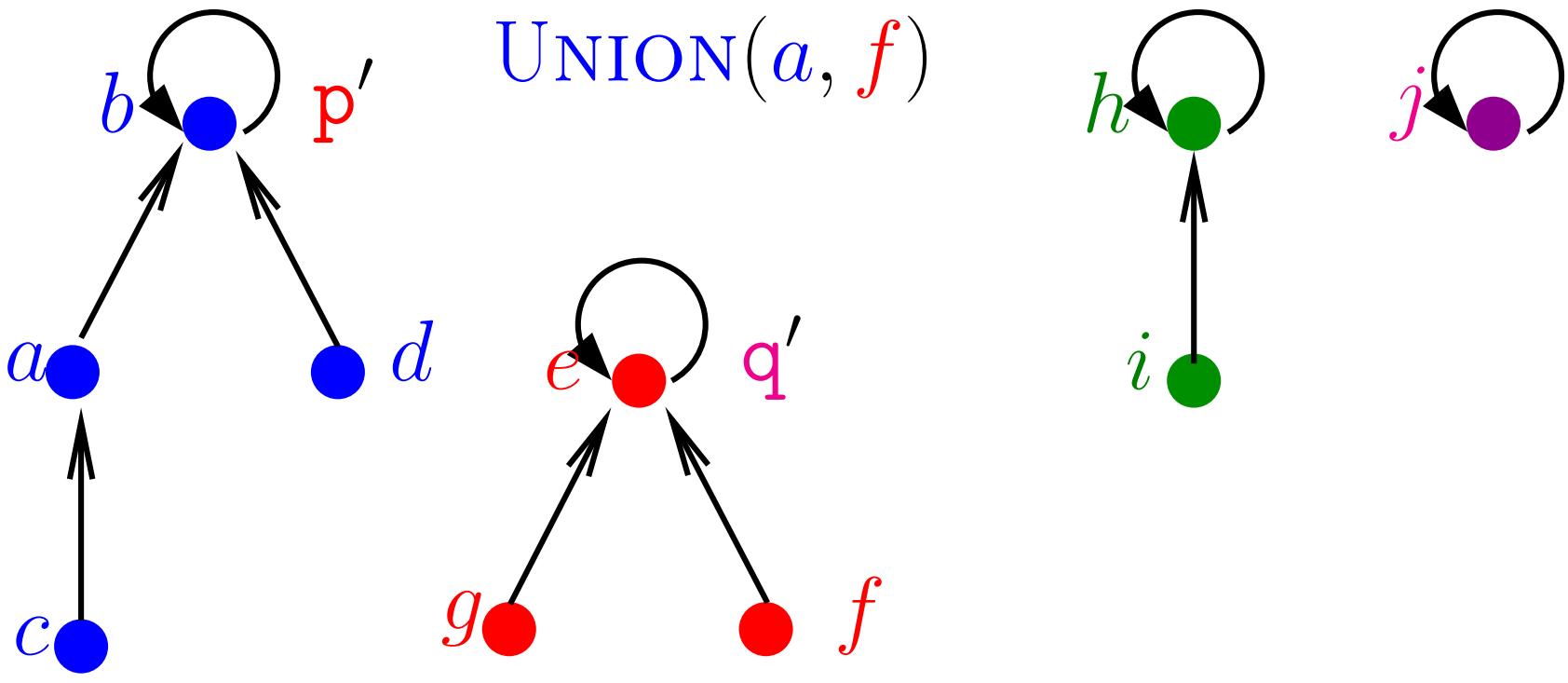
UNION(p, q)

1 $p' \leftarrow$ FINDSET (p)

2 $q' \leftarrow$ FINDSET (q)

3 $pai[q'] \leftarrow p'$

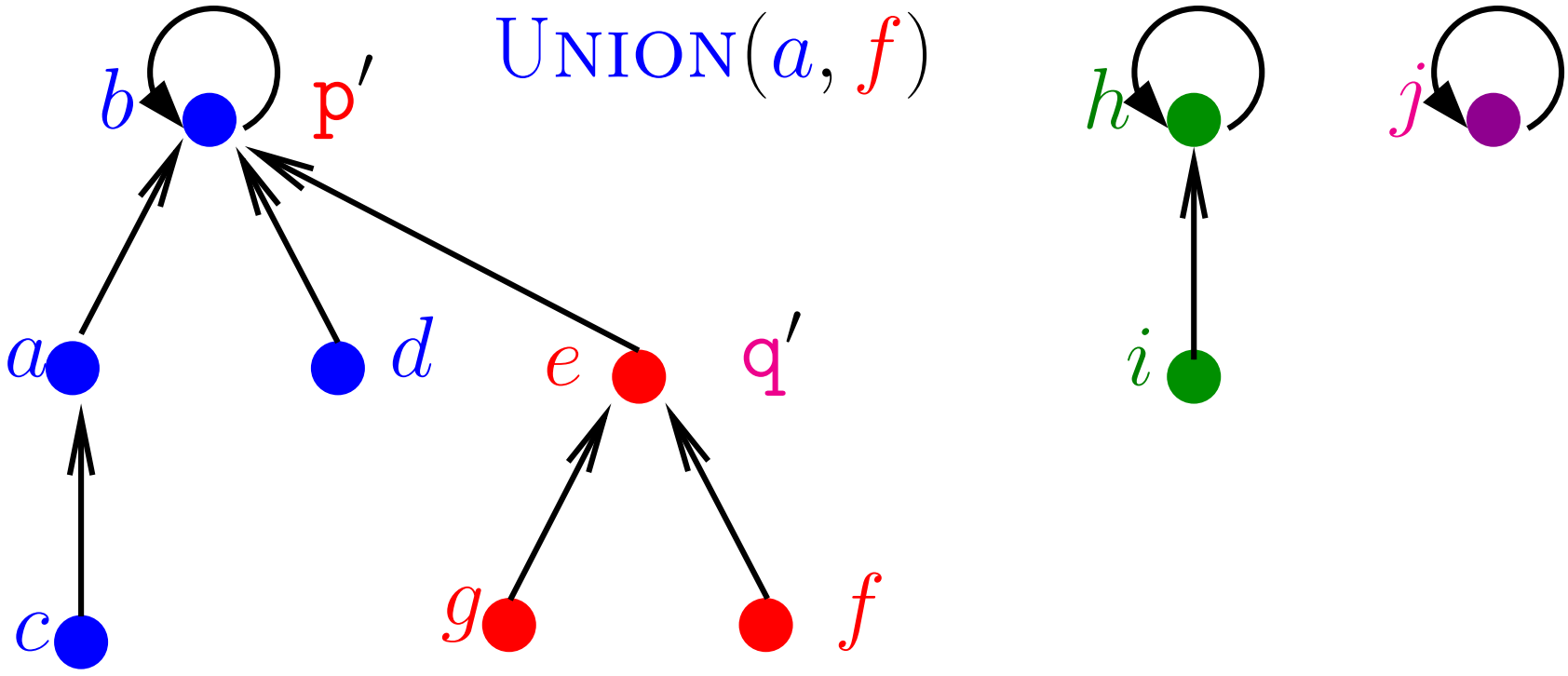
Union



UNION(p, q)

```
1   p′ ← FINDSET (p)
2   q′ ← FINDSET (q)
3   pai[q′] ← p′
```

Union



UNION(p, q)

```
1    p′ ← FINDSET (p)
2    q′ ← FINDSET (q)
3    pai[q′] ← p′
```

MakeSet, Union e FindSet

```
MakeSet(p)
1    pai[p] ← p

Union(p, q)
1    p' ← FindSet (p)
2    q' ← FindSet (q)
3    pai[q'] ← p'

FindSet(p)
1    se pai[p] = p
2        então devolva p
3        senão devolva FindSet (pai[p])
```

QuickUnionUF

`ufInit`(10)


```
uf
+-----------------------------------------------+
| parent (static)                 count: 10 (static) |
|   |                                           |
|   |    +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
|                                                     |
| Métodos:                                            |
|   count(), connected(), find(), union(), free()     |
+-----------------------------------------------+
```

QuickUnionUF

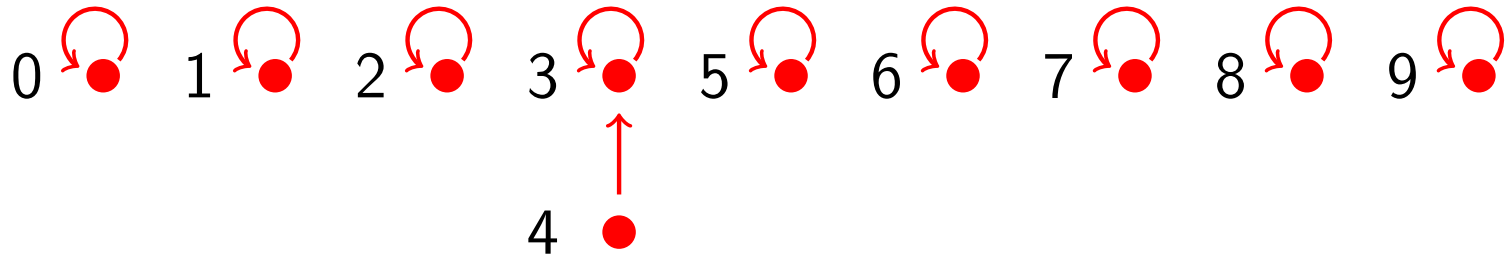
`ufFind`(3) retorna 3

`ufFind`(0) retorna 0


```
uf
+-----------------------------------------------+
| parent (static)                 count: 10 (static) |
|   |                                           |
|   |    +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
|                                                     |
| Métodos:                                            |
|   count(), connected(), find(), union(), free()     |
+-----------------------------------------------+
```

QuickUnionUF

`ufUnion`(4, 3)

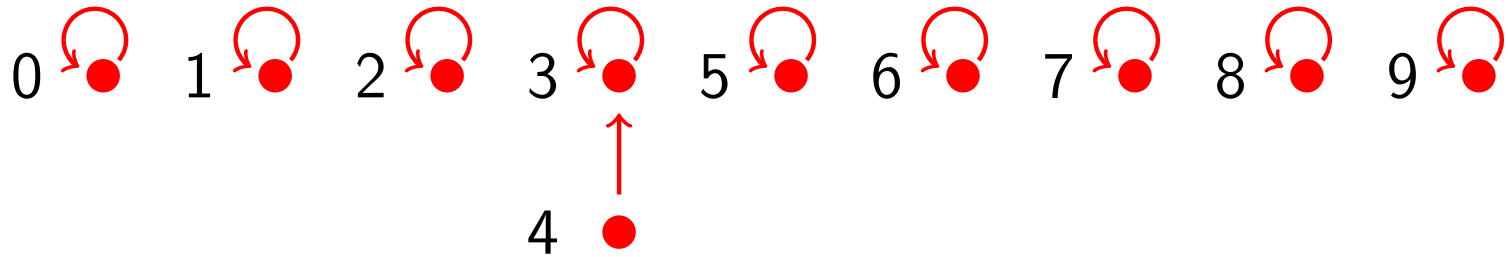



```
uf
+-----------------------------------------------+
| parent (static)                 count:  9 (static) |
|   |     +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                      |
|    count(), connected(), find(), union(), free()   |
+-----------------------------------------------+
```

QuickUnionUF

`ufFind`(3) retorna 3

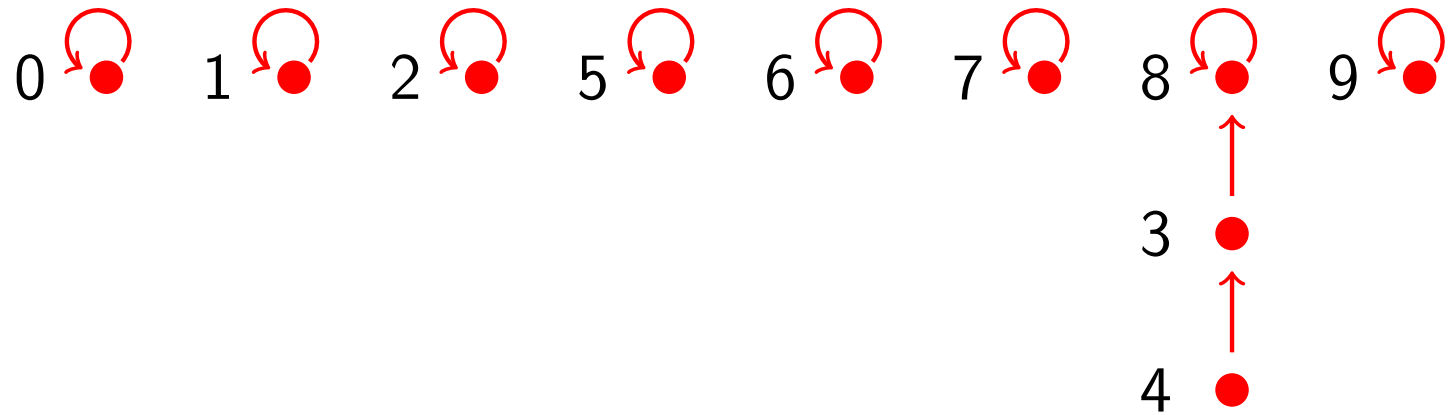
`ufFind`(4) retorna 3




```
uf
+-----------------------------------------+
| parent (static)              count:  9 (static) |
|  |    +---+---+---+---+---+---+---+---+---+---+   |
|  +--> | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 |   |
|       +---+---+---+---+---+---+---+---+---+---+   |
|         0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                          |
|   count(), connected(), find(), union(), free()   |
+-----------------------------------------+
```

QuickUnionUF

`ufUnion`(3, 8)

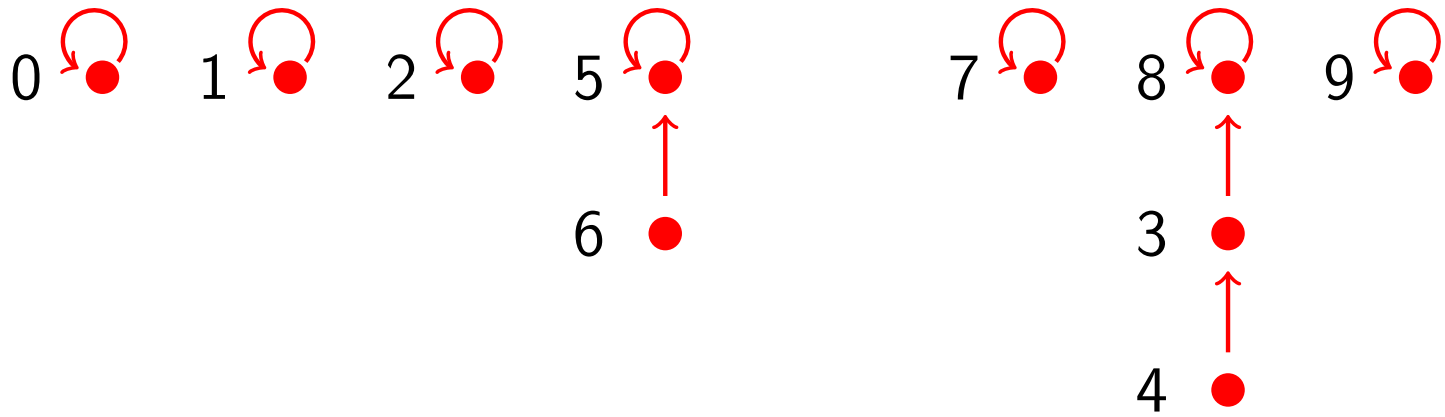


uf

```
+-----------------------------------------------+
| parent (static)                 count:  8 (static) |
|   |     +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 2 | 8 | 3 | 5 | 6 | 7 | 8 | 9 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                            |
|   count(), connected(), find(), union(), free()     |
+-----------------------------------------------+
```

QuickUnionUF

`ufUnion`(6, 5)

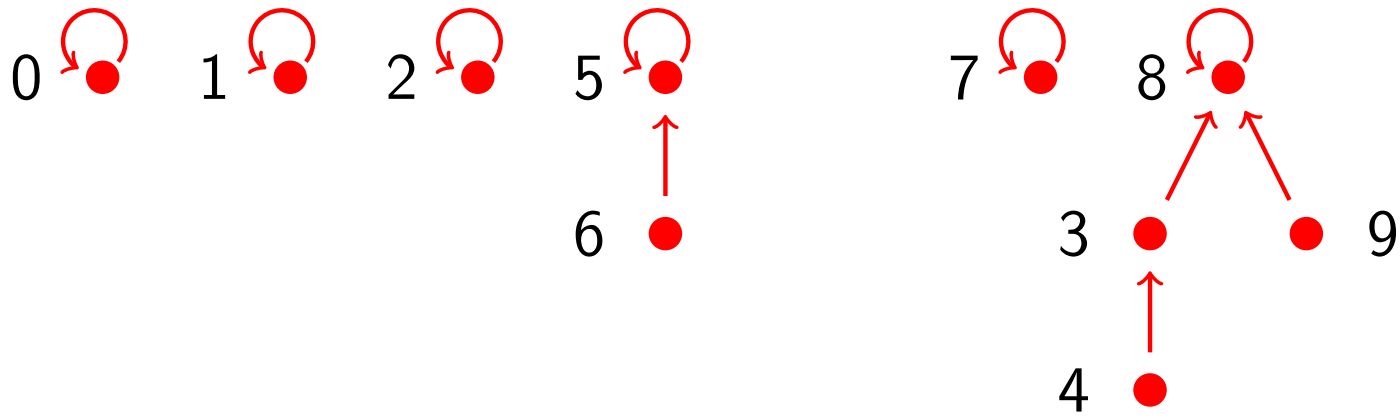


uf

```
+-------------------------------------------------+
| parent (static)                  count:  7 (static) |
|   |     +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 9 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                            |
|   count(), connected(), find(), union(), free()     |
+-----------------------------------------------------+
```

QuickUnionUF

`ufUnion`(9, 4)

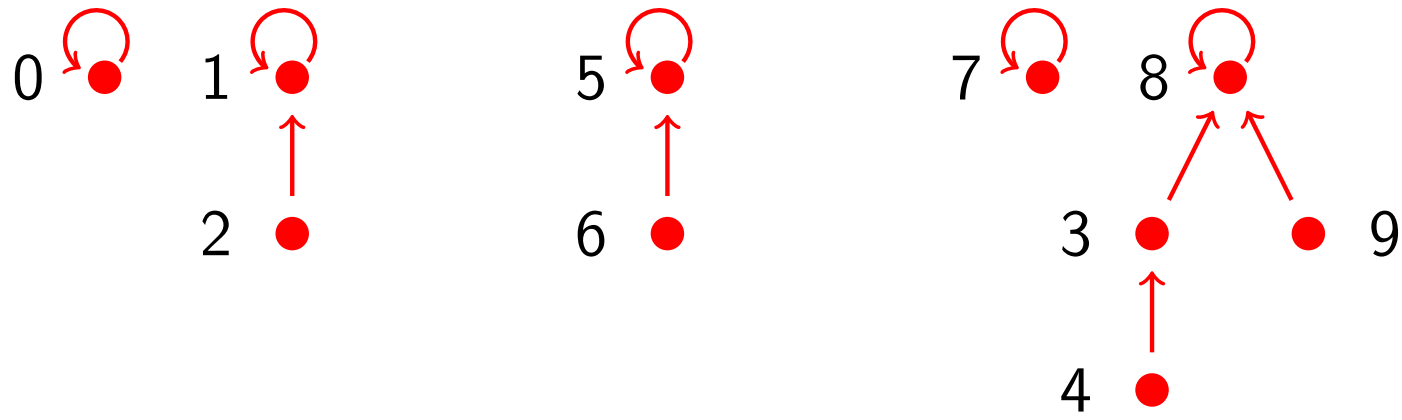


uf

```
+-----------------------------------------------+
| parent (static)                 count:  6 (static) |
|   |     +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                      |
|   count(), connected(), find(), union(), free()   |
+-----------------------------------------------+
```

QuickUnionUF

`ufUnion`(2, 1)

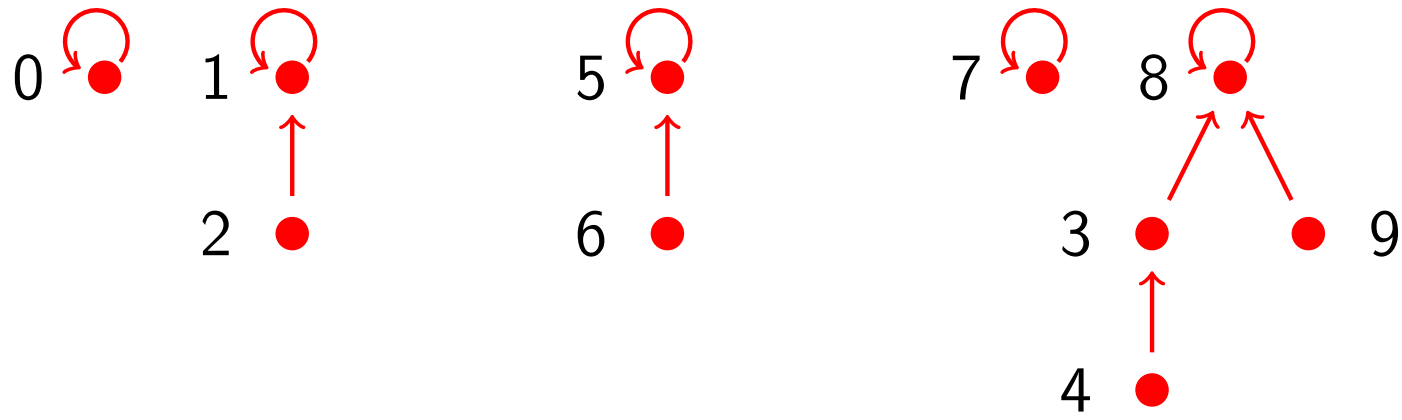


uf

```
+-----------------------------------------------+
| parent (static)                 count:  5 (static) |
|   |     +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                            |
|   count(), connected(), find(), union(), free()     |
+-----------------------------------------------+
```

QuickUnionUF

`ufUnion`(8, 9)



uf

```
+-----------------------------------------------+
| parent (static)                 count:  5 (static) |
|   |     +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                            |
|   count(), connected(), find(), union(), free()     |
+-----------------------------------------------+
```

QuickUnionUF

`ufUnion`(5, 0)

uf

```
+-----------------------------------------------+
| parent (static)                 count:  4 (static) |
|   |     +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 1 | 8 | 3 | 0 | 5 | 7 | 8 | 8 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                            |
|   count(), connected(), find(), union(), free()     |
+-----------------------------------------------+
```

QuickUnionUF

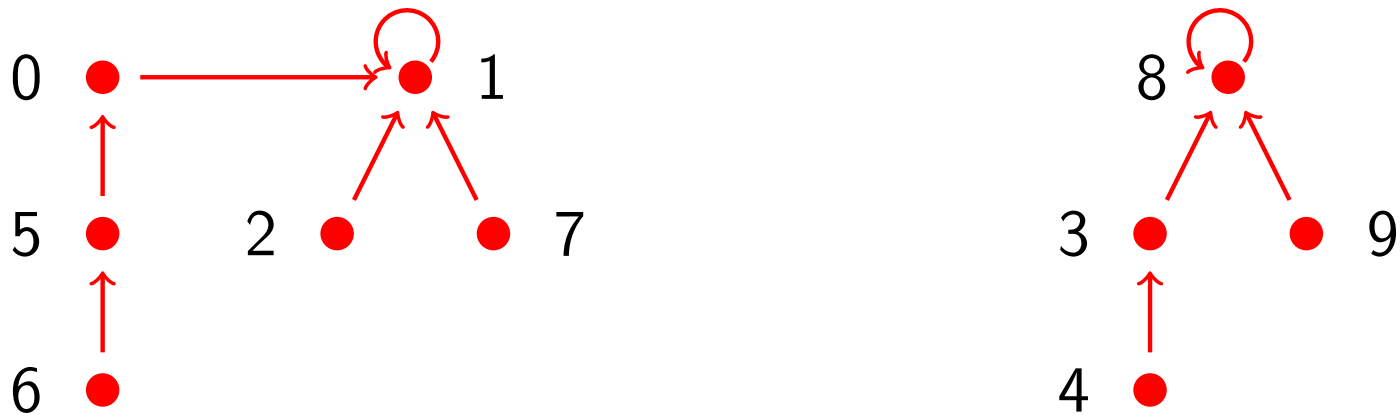
`ufUnion`(7, 2)

uf

```
+-----------------------------------------------+
| parent (static)                 count:  3 (static) |
|   |     +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 0 | 1 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                            |
|   count(), connected(), find(), union(), free()     |
+-----------------------------------------------+
```

QuickUnionUF

`ufUnion`(6, 1)



uf

```
+-----------------------------------------------+
| parent (static)                 count:  2 (static) |
|   |     +---+---+---+---+---+---+---+---+---+---+   |
|   +--> | 1 | 1 | 1 | 8 | 3 | 0 | 1 | 1 | 8 | 8 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
| Métodos:                                            |
|   count(), connected(), find(), union(), free()     |
+-----------------------------------------------+
```

Arquivo QuickFindUF.c: esqueleto

```
#include "uf.h"

static int *pai;
static int count; /* no. compomentes */

void ufInit(int n) {...}
void ufUnion(int p, int q) {...}
int  ufFind(int p) {...}
bool ufConnected(int p, int q) {...}
int  ufCount() {...}
void ufFree() {...}
```

Implementação: QuickUnionUF.c

```
void ufInit(int n) {
   int i;
   pai = mallocSafe(n * sizeof(int));
   for (i = 0; i < n; i++) pai[i] = i;
   count = n;
}

/* retorna o id do componente de p */
int ufFind(int p) {
   while (pai[p] != p) p = pai[p];
   return p;
}
```

Implementação: QuickUnionUF.c

```
bool ufConnected(int p, int q) {
   return ufFind(p) == ufFind(q);
}
void ufUnion(int p, int q) {
   int pl = ufFind(p), ql = ufFind(q);
   if (pl != ql) return;
   pai[ql] = pl;
   count--;
}
```

Implementação: QuickUnionUF.c

```
bool ufConnected(int p, int q) {
   return ufFind(p) == ufFind(q);
}

void ufUnion(int p, int q) {
   int pl = ufFind(p), ql = ufFind(q);
   if (pl != ql) return;
   pai[ql] = pl;
   count--;
}

int ufCount() {
   return count;
}
```

```
void ufFree() {
   free(pai);
   pai = NULL;
   count = 0;
}
```

Consumo de tempo

`ufInit(n)`	$\Theta(n)$
`ufFind(p)`	$O(n)$
`ufUnion(p, q)`	$O(n)$

M M M U F U U F U F F F U F

$\underbrace{}_{n}$

$\underbrace{}_{m}$

Custo total da sequência:

$$n\,\Theta(1) + m\,O(n) + n\,O(n) = O(mn)$$

Experimentos

```
% time client < tinyUF.txt
```
2 components
0.003seg

```
% time client < mediumUF.txt
```
3 components
0.005seg

```
% time client < largeUF.txt
```
:-(

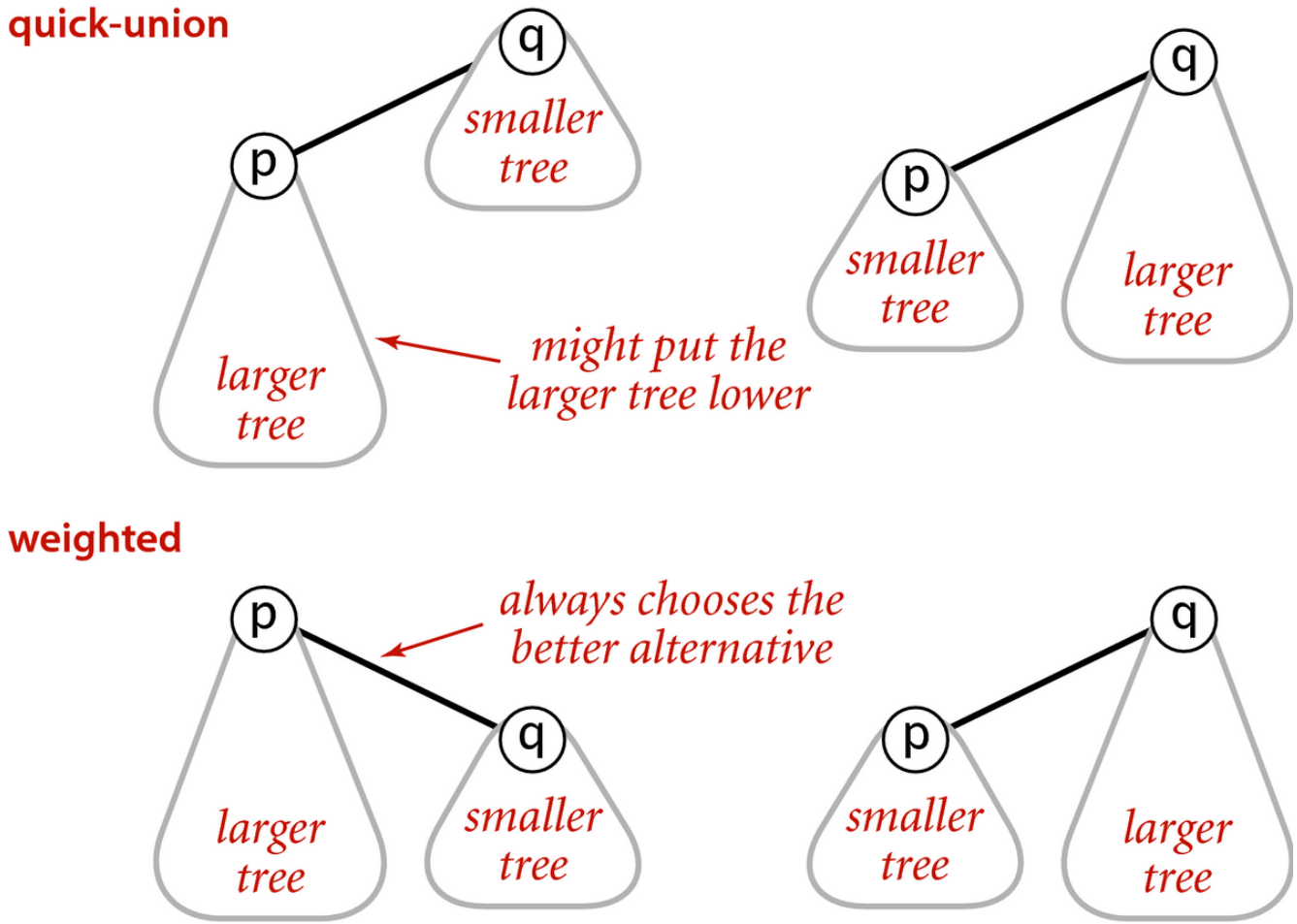
WeightedQuickUnionUF

Ideia:

ligar a raiz da árvore com menos elementos
na raiz da árvore com mais elementos.

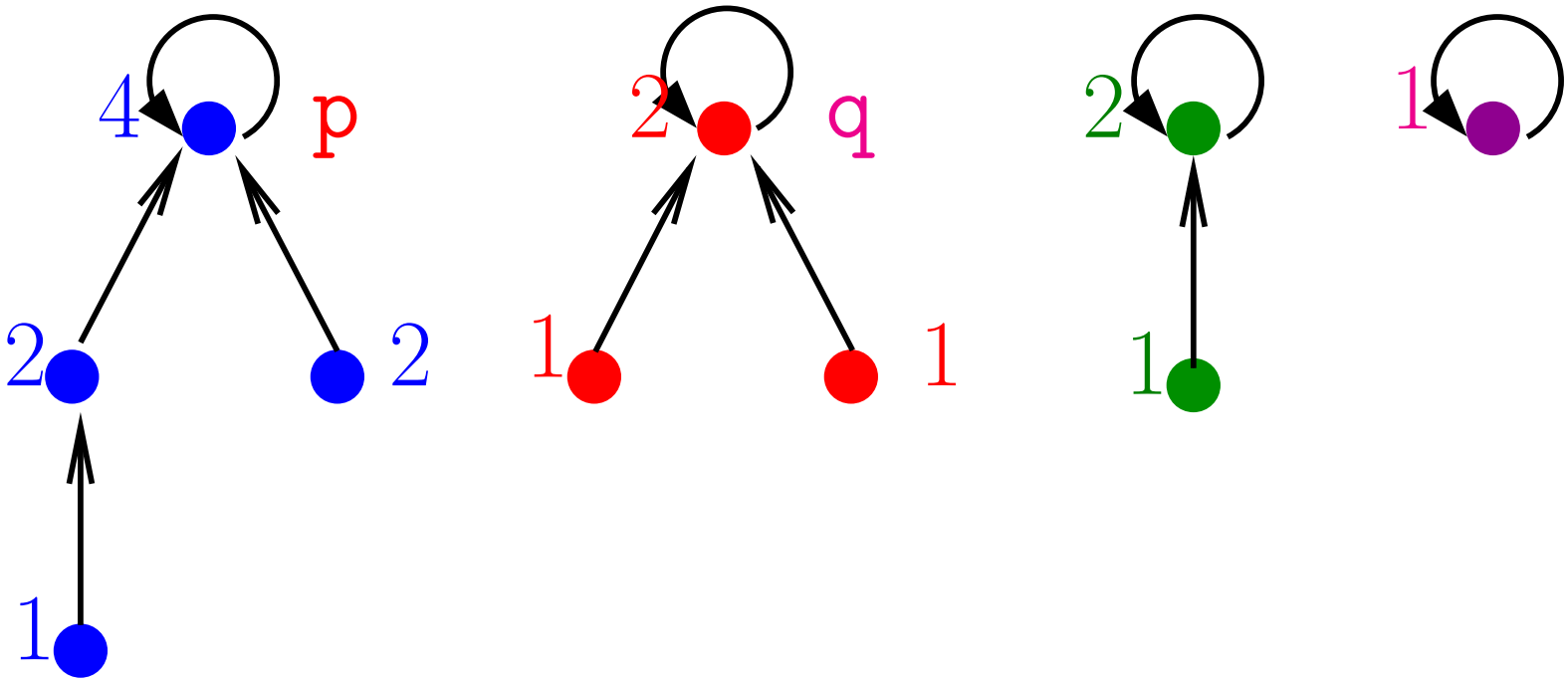
Isso seria a política natural para
tornarmos o quick-find mais eficiente.

WeightedQuickUnionUF



Weighted quick-union

union by rank

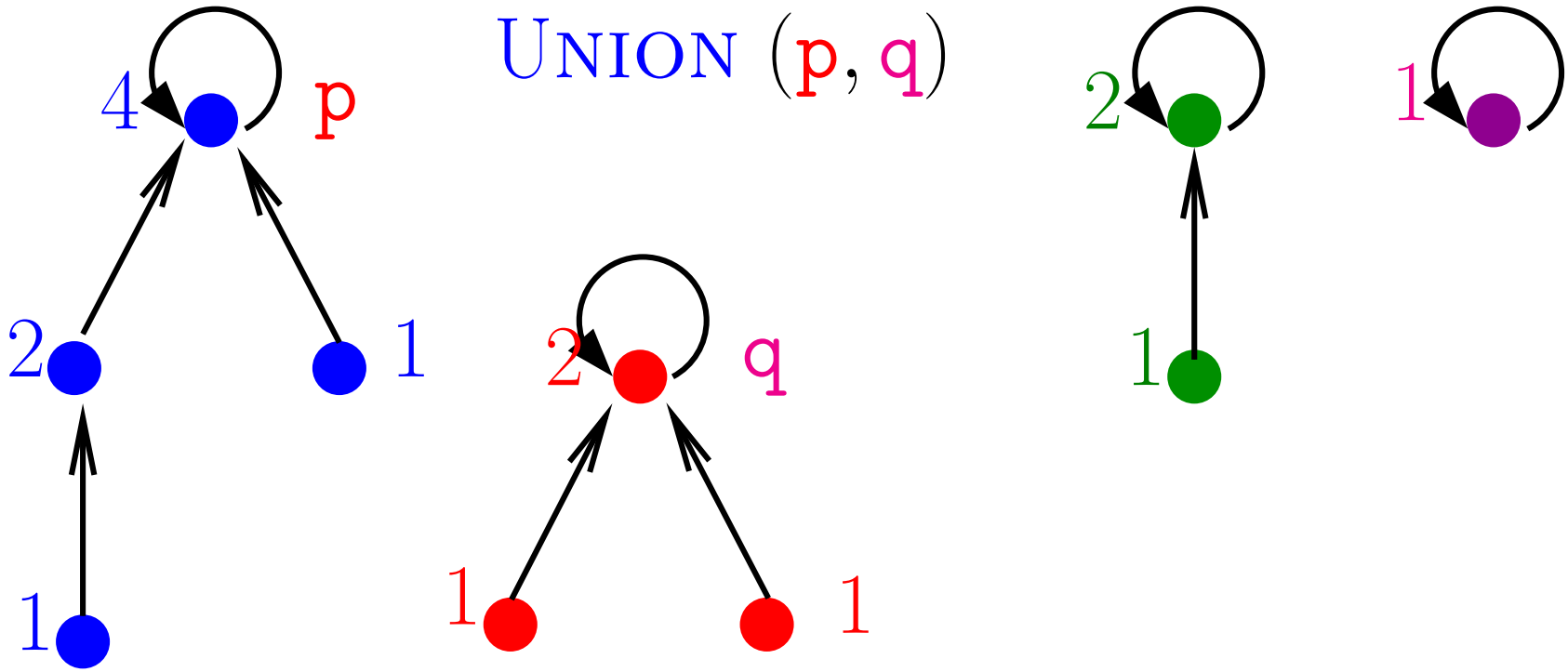


`rank`[`p`] = posto do nó `p`

MAKESET (`p`)

```
1   pai[p] ← p
2   rank[p] ← 0
```

union by rank

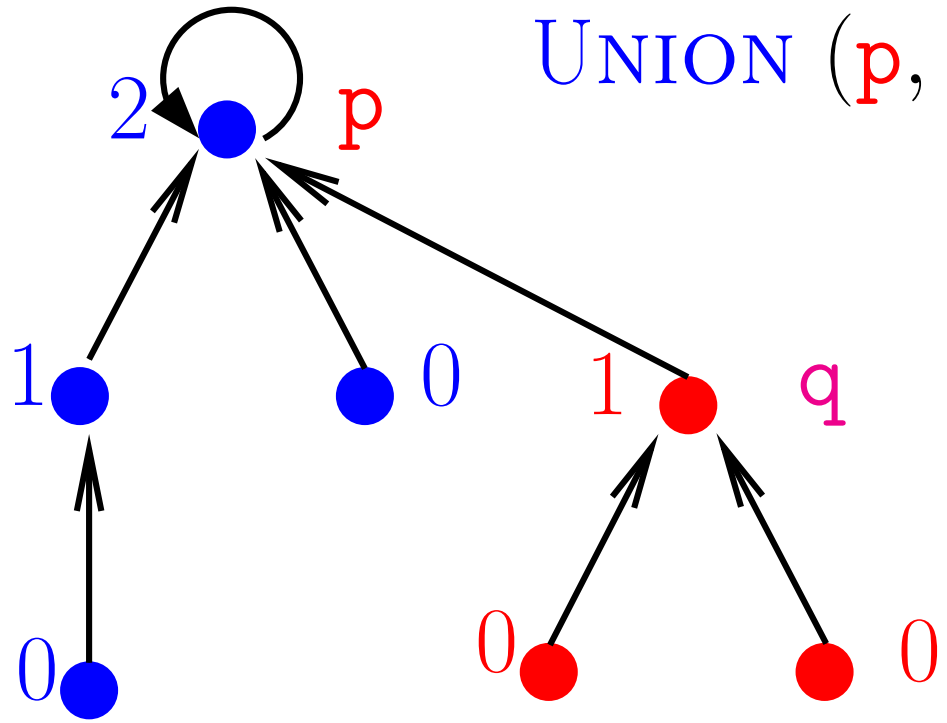


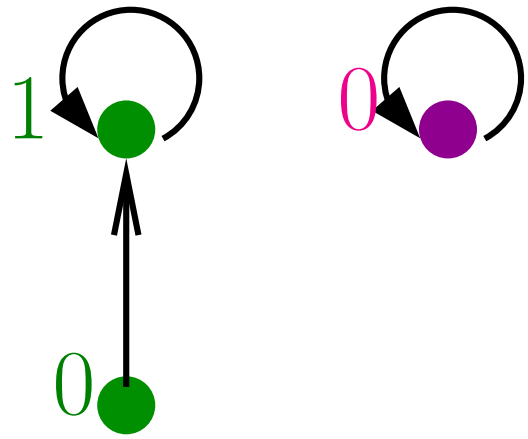
`rank`[p] = posto do nó p

MAKESET (p)

```
1   pai[p] ← p
2   rank[p] ← 0
```

union by rank



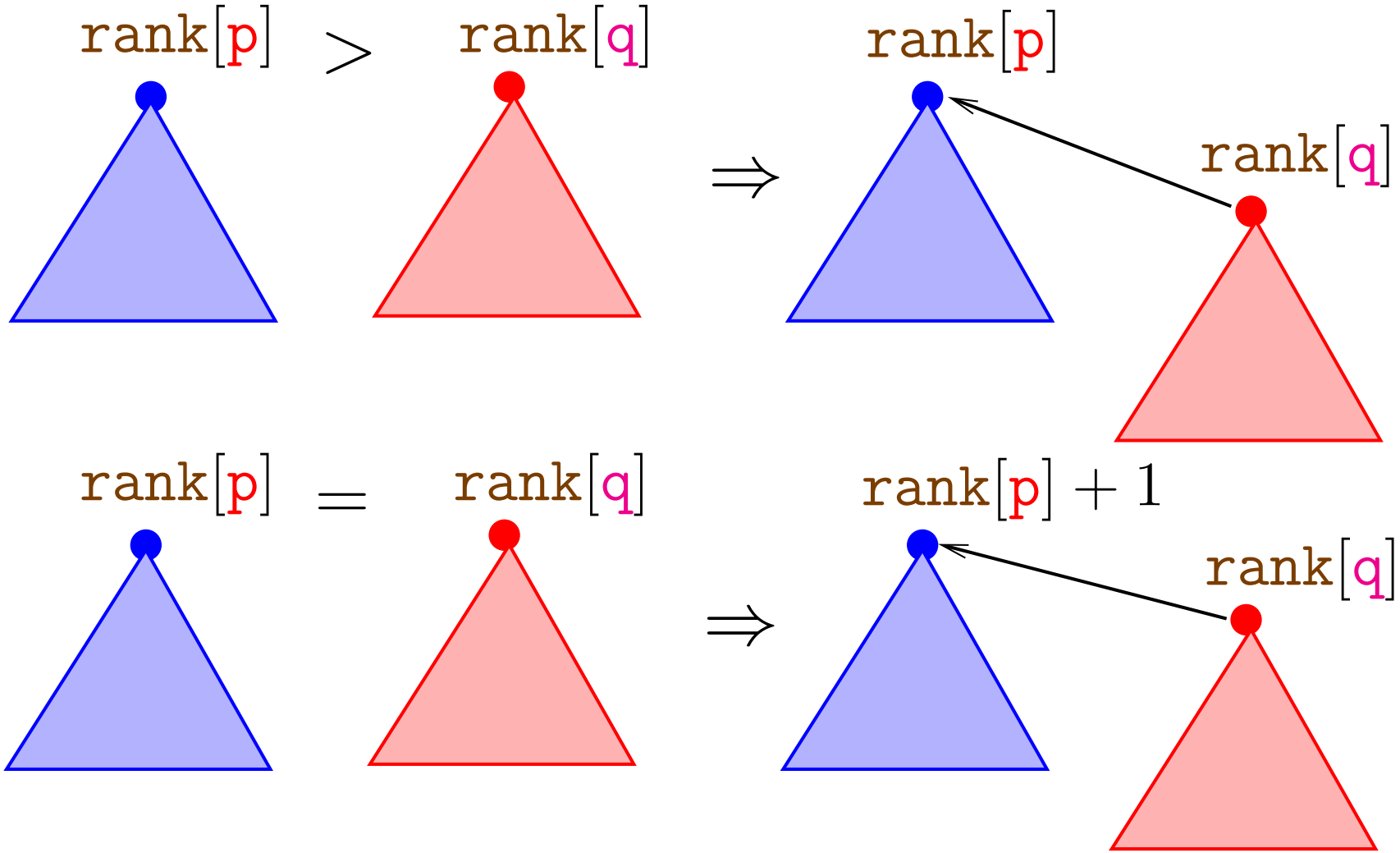


`rank`[p] = posto do nó p

MAKESET (p)

```
1    pai[p] ← p
2    rank[p] ← 0
```

union by rank



union by rank

```
UNION (p, q)   ▷ com "union by rank"
1   p' ← FINDSET (p)
2   q' ← FINDSET (q)     ▷ supõe que p' ≠ q'
3   se rank[p'] > rank[q']
4      então pai[q'] ← p'
5      senão pai[p'] ← q'
6            se rank[p'] = rank[q']
7               então rank[q'] ← rank[q'] + 1
```

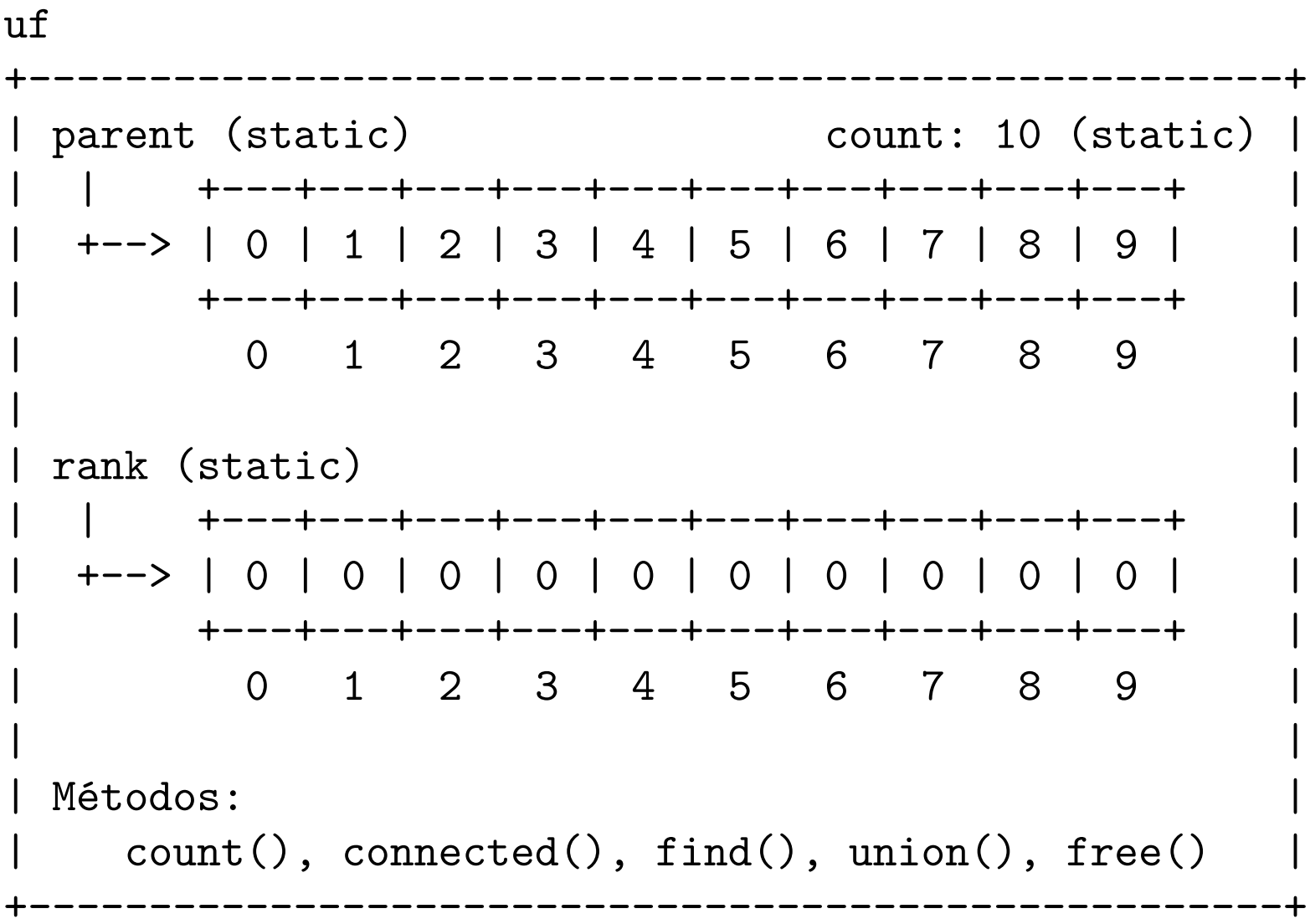
Variante: *union by size*

`size[p]`: número de nós na árvore enraizada em `p`

```
UNION (p, q)  ▷ com "union by size"
1   p' ← FINDSET (p)
2   q' ← FINDSET (q)    ▷ supõe que p' ≠ q'
3   se size[p'] > size[q']
4       então pai[q'] ← p'
5             size[p'] ← size[p'] + size[q']
6       senão pai[p'] ← q'
7             size[q'] ← size[q'] + size[p']
```

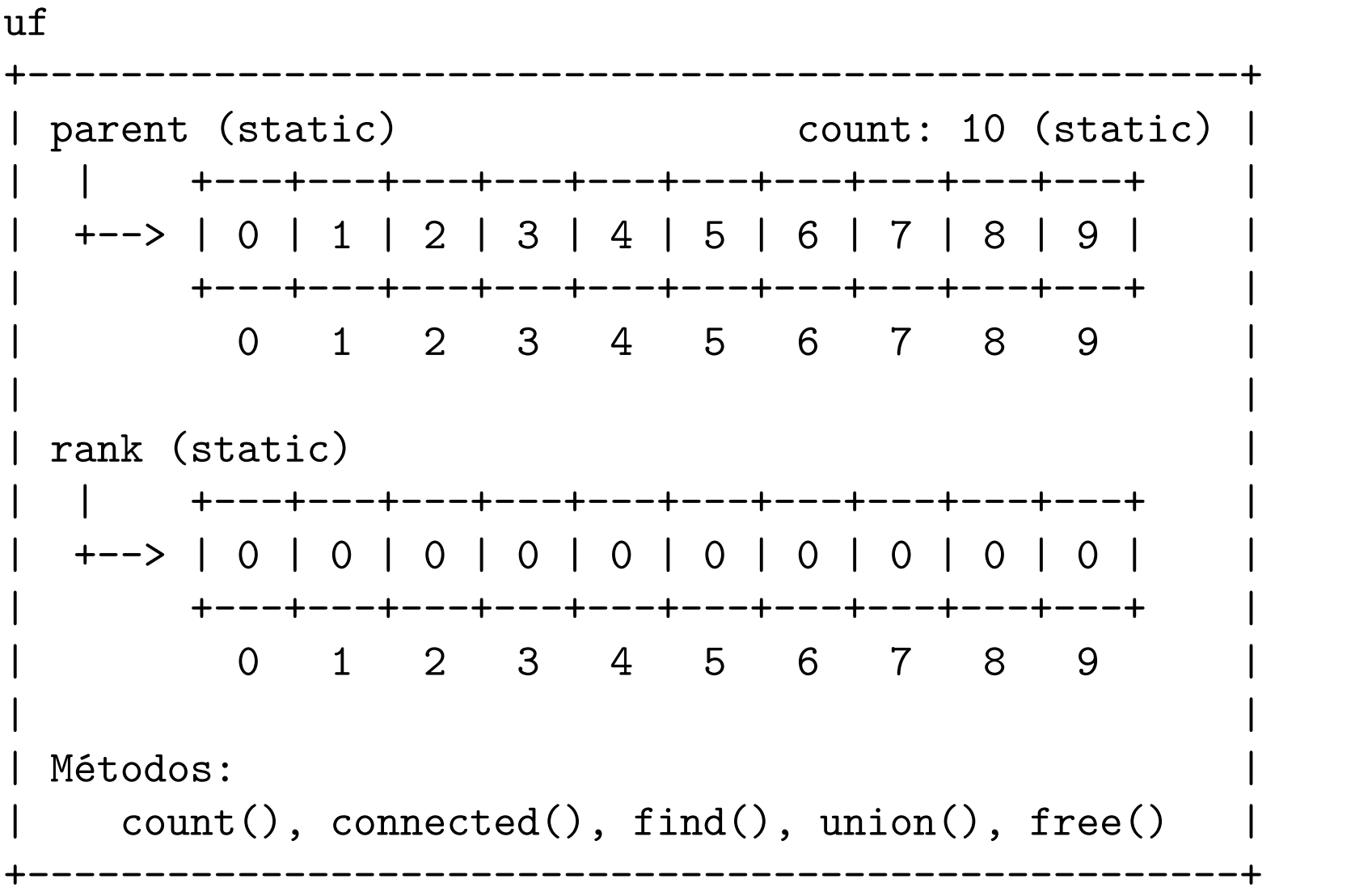
WeightedQuickUnionUF

`ufInit`(10)



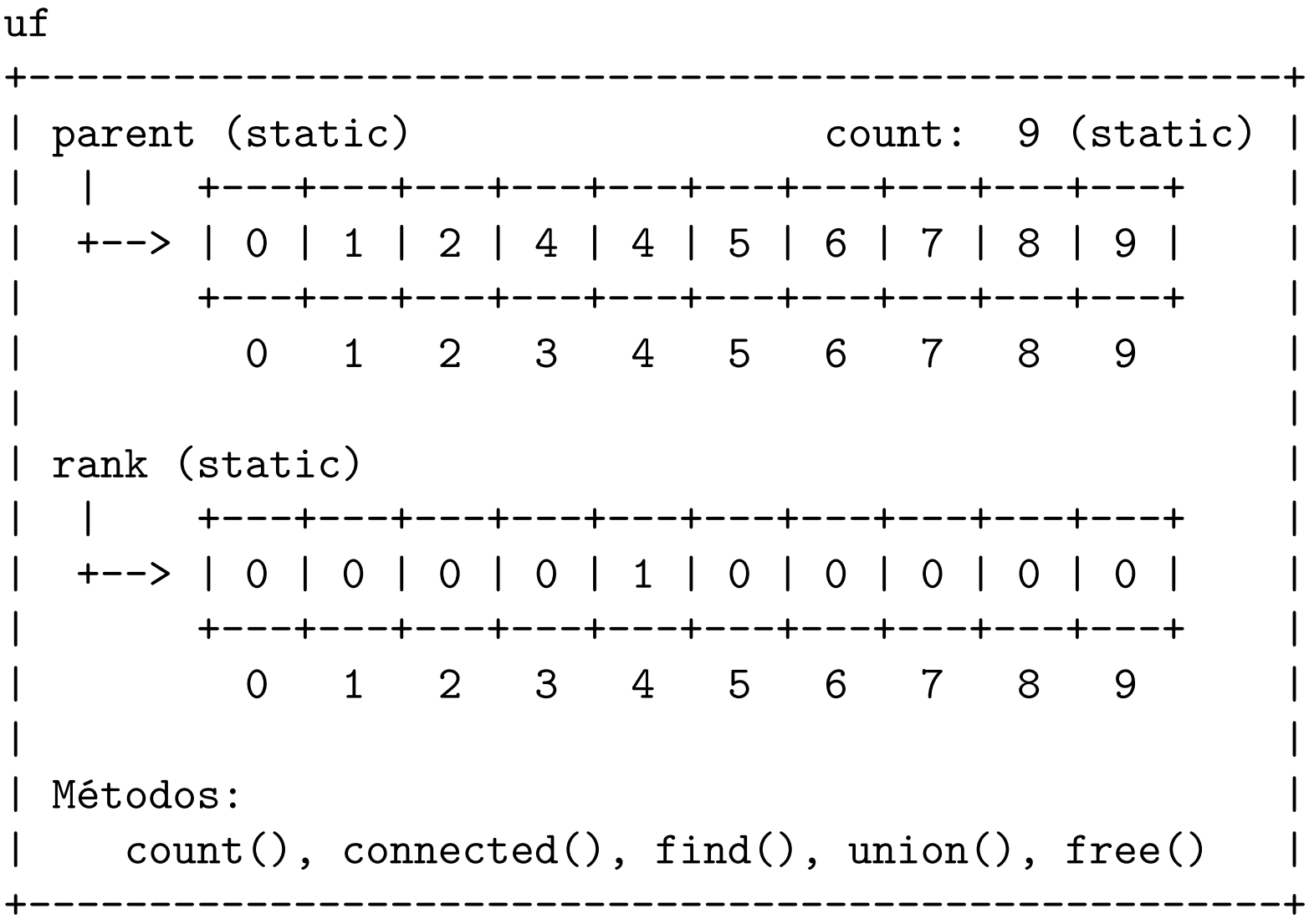
WeightedQuickUnionUF

`ufFind`(3) retorna 3 `ufFind`(0) retorna 0



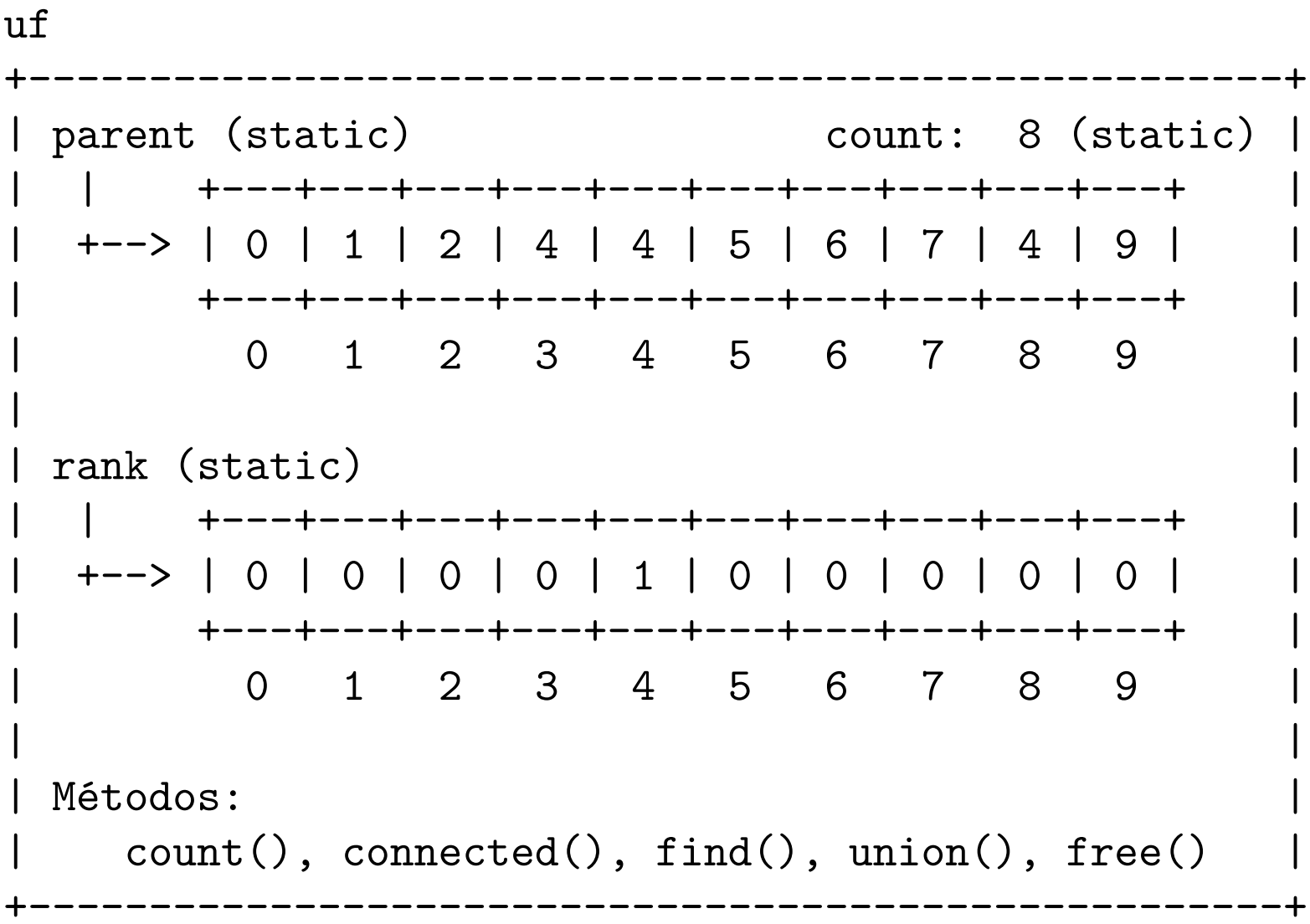
WeightedQuickUnionUF

ufUnion(4, 3)



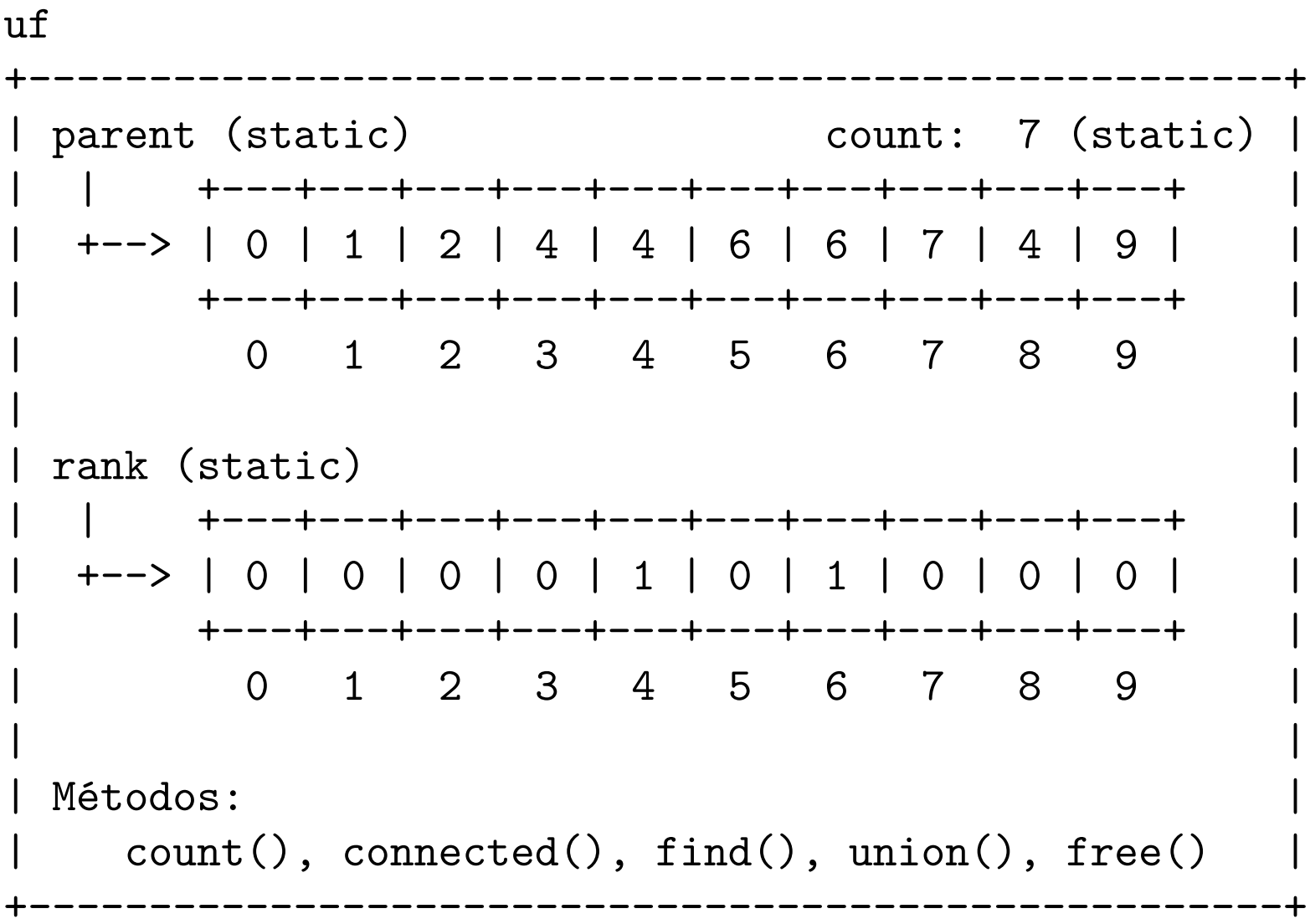
WeightedQuickUnionUF

ufUnion(3, 8)



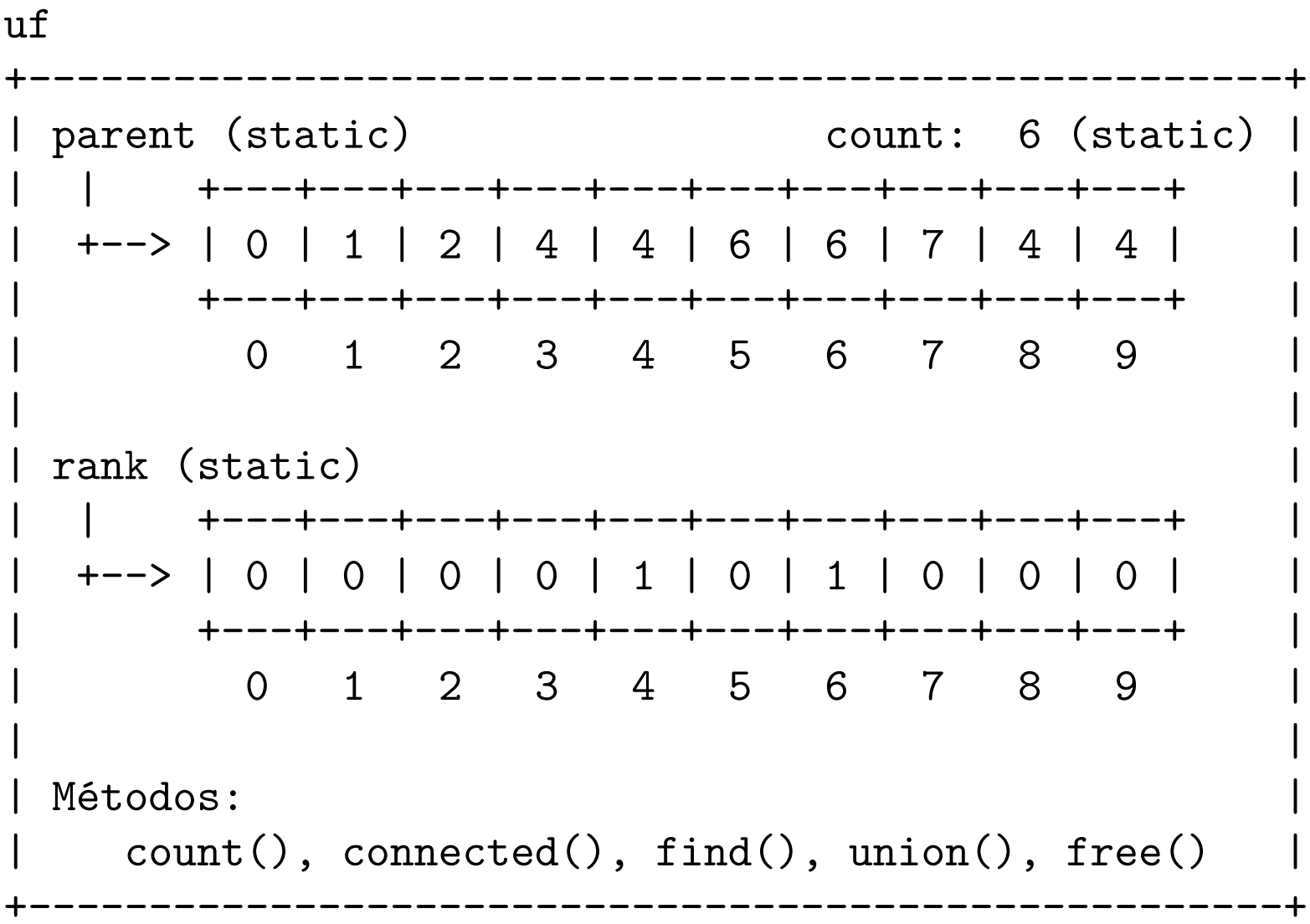
WeightedQuickUnionUF

`ufUnion`(6, 5)



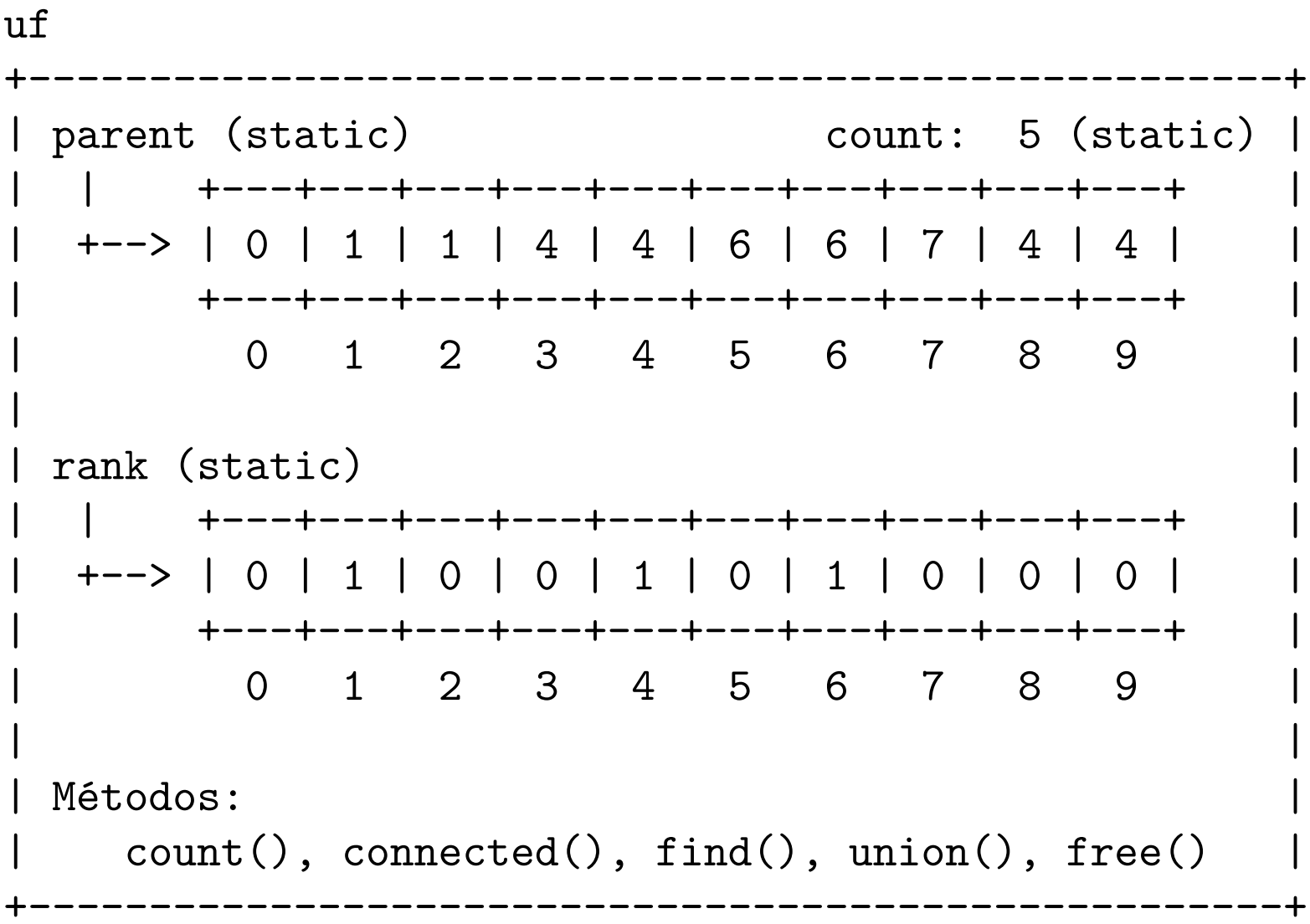
WeightedQuickUnionUF

ufUnion(9, 4)



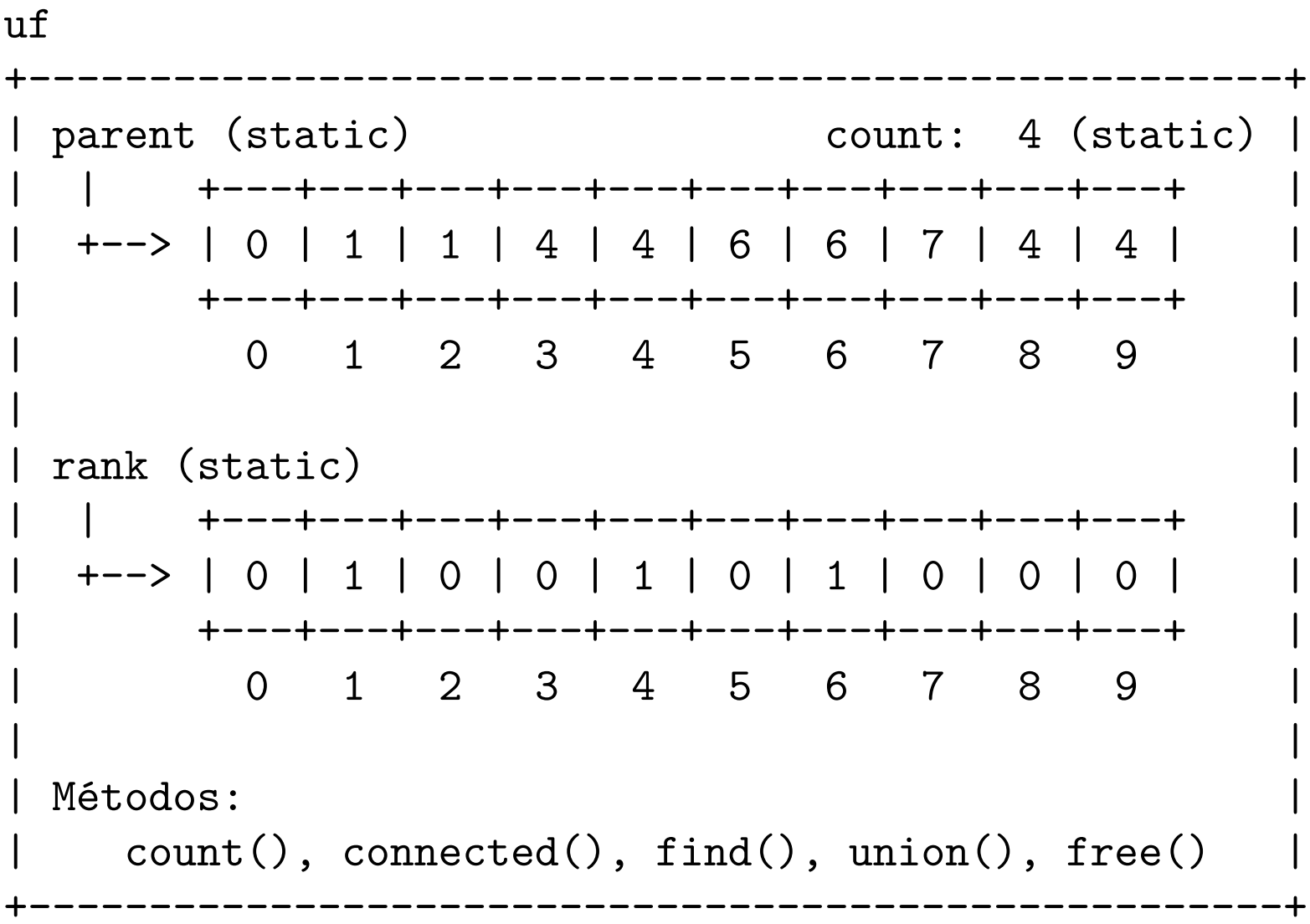
WeightedQuickUnionUF

`ufUnion`(2, 1)



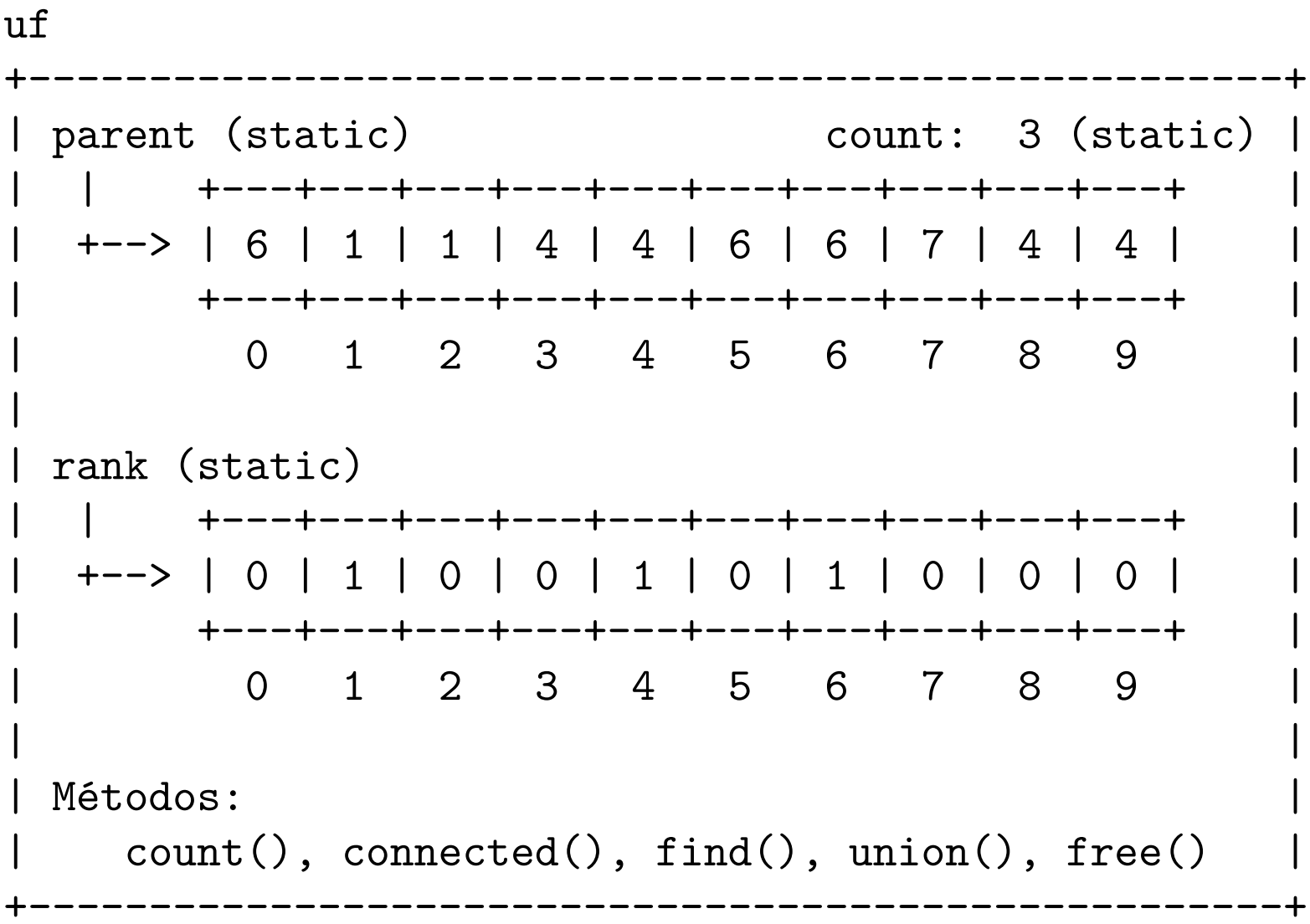
WeightedQuickUnionUF

ufUnion(8, 9)



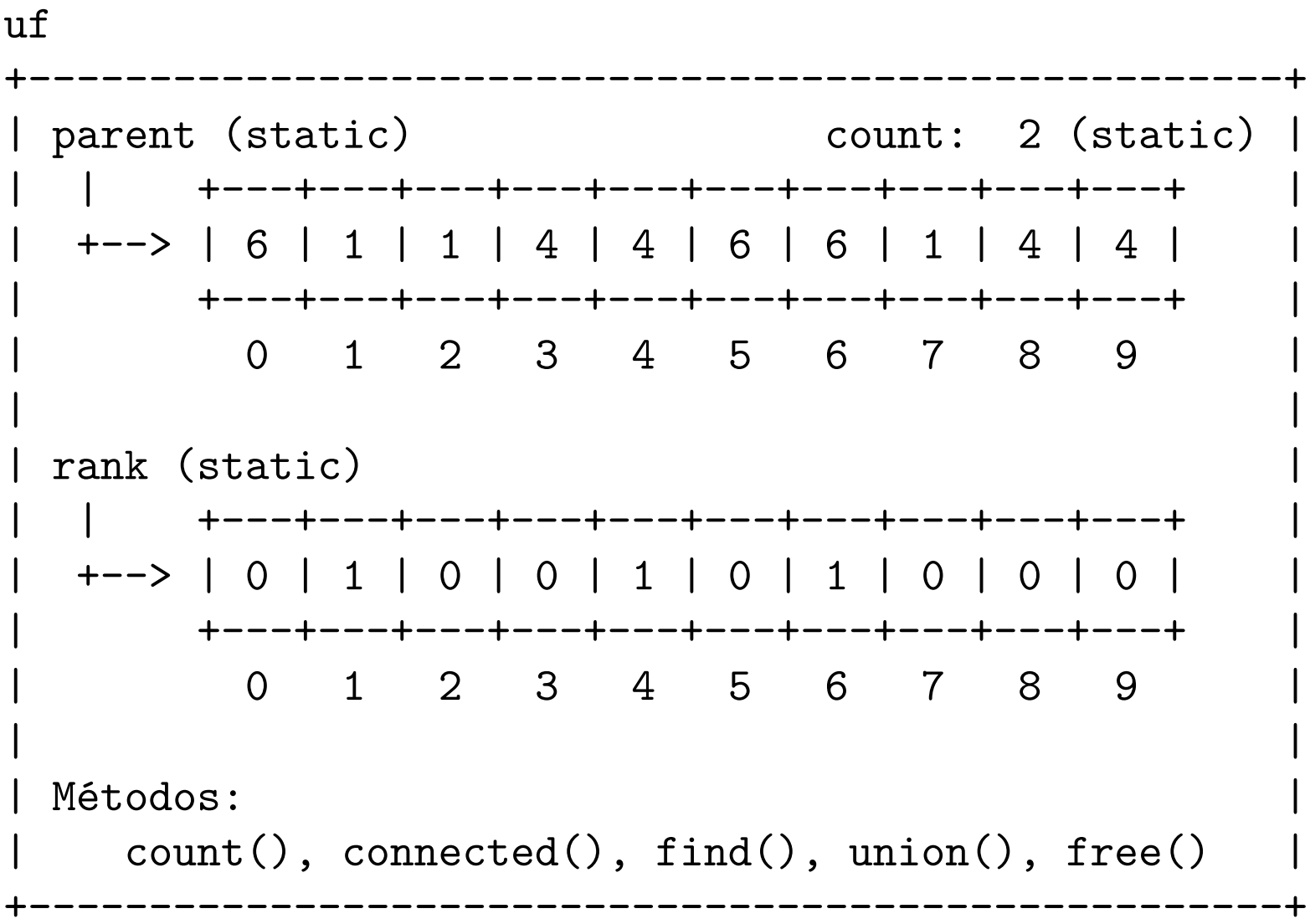
WeightedQuickUnionUF

`ufUnion`(5, 0)



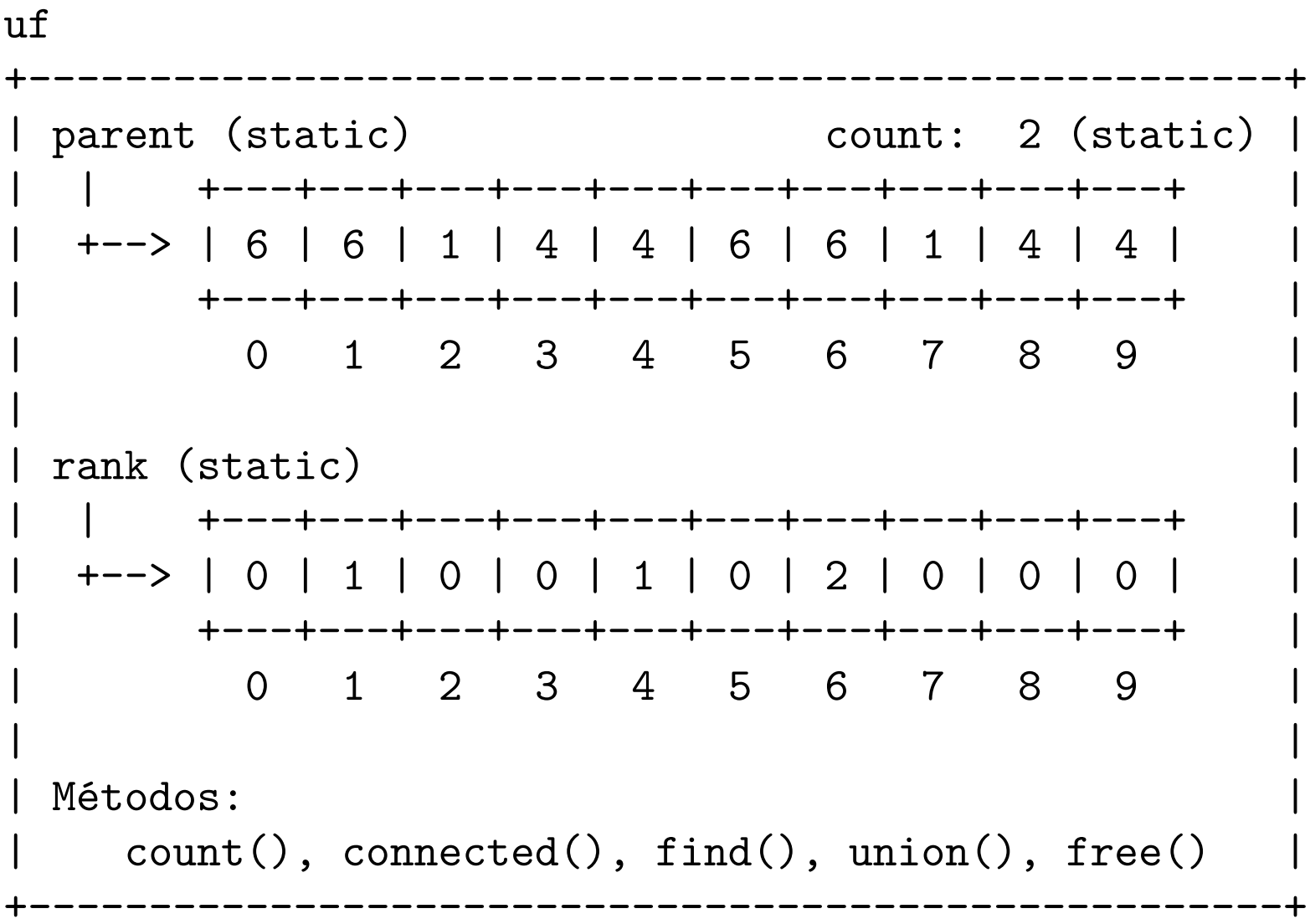
WeightedQuickUnionUF

ufUnion(7, 2)



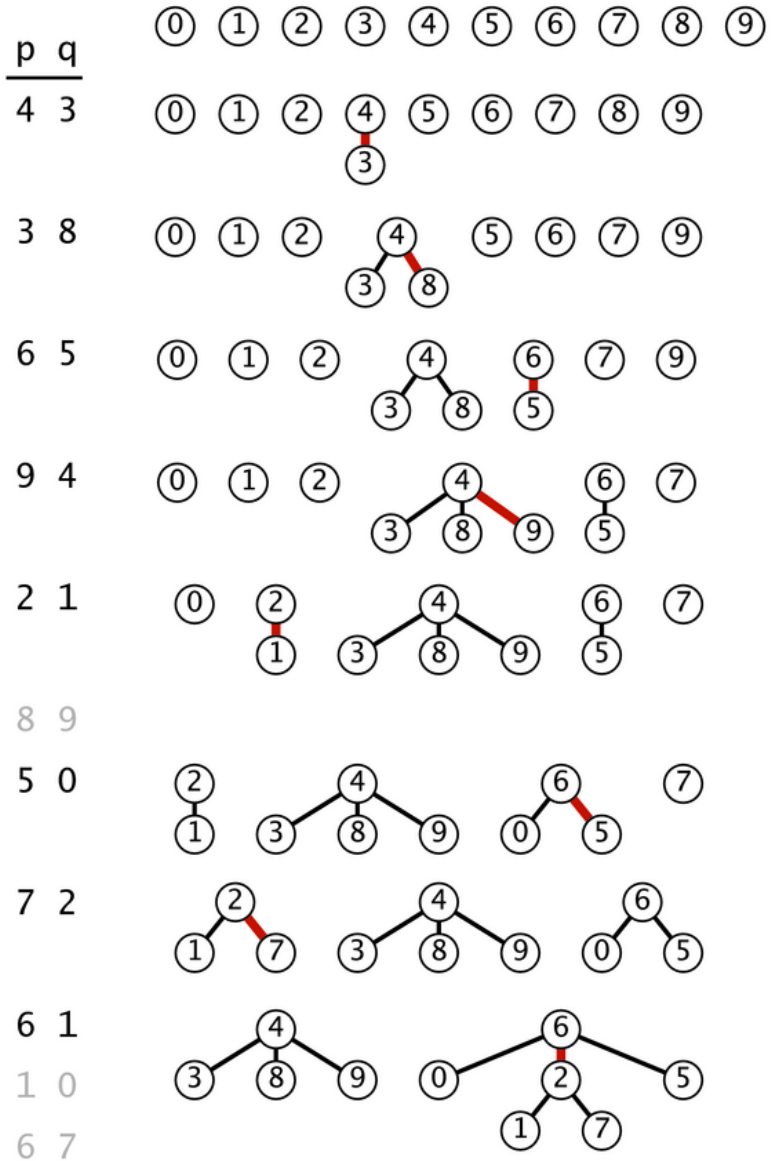
WeightedQuickUnionUF

ufUnion(6, 1)

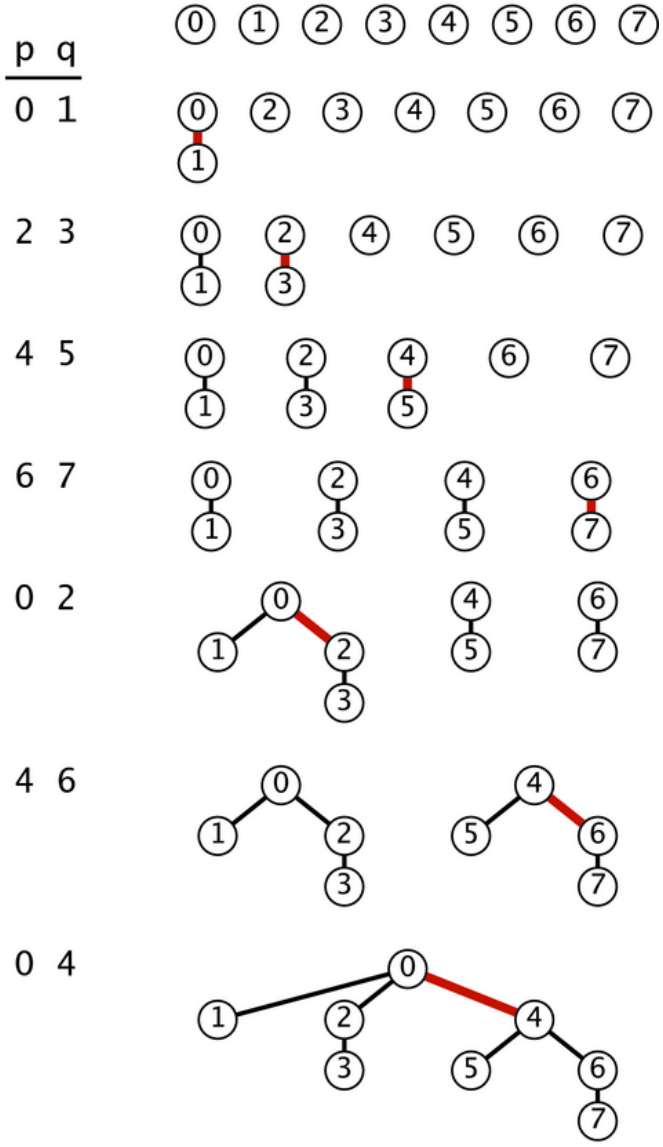


Simulação

reference input



worst-case input



Weighted quick-union traces (forests of trees)

Implementação: WeightedQuickUnionUF.c

```
#include "uf.h"
static int *pai;
static int *rank;
static int count;
```

Implementação: WeightedQuickUnionUF.c

```
#include "uf.h"
static int *pai;
static int *rank;
static int count;

void ufInit(int n) {
    int i;
    pai = mallocSafe(n * sizeof(int));
    rank = mallocSafe(n * sizeof(int));
    for (i = 0; i < n; i++) {
        pai[i] = i;
        rank[i] = 0;
    }
    count = n;
}
```

Implementação: WeightedQuickUnionUF.c

```
void ufUnion(int p, int q) {
   int pl = ufFind(p), ql = ufFind(q);
   if (rank[pl] > rank[ql])
      pai[ql] = pl;
   else {
      pai[pl] = ql;
      if (rank[pl] == rank[ql])
         rank[ql]++;
   }
   count--;
}
```

Implementação: WeightedQuickUnionUF.c

```
/* retorna o id do componente de p */
int ufFind(int p) {
    while (pai[p] != p) p = pai[p];
    return p;
}
```

Implementação: WeightedQuickUnionUF.c

```
/* retorna o id do componente de p */
int ufFind(int p) {
    while (pai[p] != p) p = pai[p];
    return p;
}

/* p e q estão no mesmo componente? */
bool ufConnected(int p, int q) {
    return ufFind(p) == ufFind(q);
}

int ufCount() {
    return count;
}
```

```
void ufFree() {
    free(pai);
    free(rank);
    pai = rank = NULL;
    count = 0;
}
```

Estrutura *disjoint-set forest*

Para verificar que o consumo de tempo de `ufUnion()` e `ufFind()` é não superior a $\lg n$, basta demonstrar que

> *Na floresta de árvores disjuntas produzida durante uma sequência de operações* `ufUnion()`, *toda árvore com altura* h *tem pelo menos* 2^h *nós*.

A demonstração é por indução no número de operações `ufUnion()` realizadas.

Estrutura *disjoint-set forest*

Base: Inicialmente nenhuma operação `ufUnion`() foi realizada e toda árvore tem altura zero e possui um nó. Logo vale a afirmação.

Passo: Sejam `p` e `q` elementos e considere a operação `ufUnion`(`p`, `q`).

Se `p` e `q` estão em uma mesma árvore, então não há o que demonstrar.

Portanto, podemos supor que a árvore T_p que contém `p` e a árvore T_q que contém `q` são distintas.

Estrutura *disjoint-set forest*

Sejam

- h_p e n_p a altura e número de nós de T_p e
- h_q e n_q a altura e número de nós de T_q.

Pela hipótese de indução, $n_p \geq 2^{h_p}$ e $n_q \geq 2^{h_q}$.

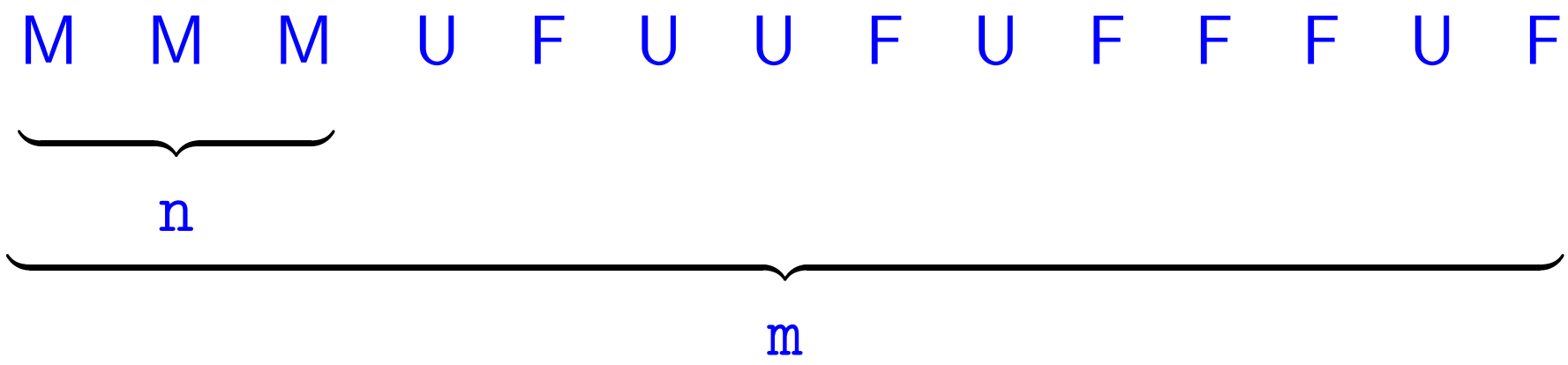
Seja T a árvore de altura h resultante da operação `ufUnion`(p,q). Se $h \leq \max\{h_p, h_q\}$, não há o que demonstrar. Assim, podemos supor que, digamos, $n_p \geq n_q$ e $h = h_q + 1$. Logo,

$$n = n_p + n_q \geq n_q + n_q \geq 2^{h_q} + 2^{h_q} = 2^{h_q+1} = 2^h,$$

o que encerra este rascunho de demonstração.

Consumo de tempo

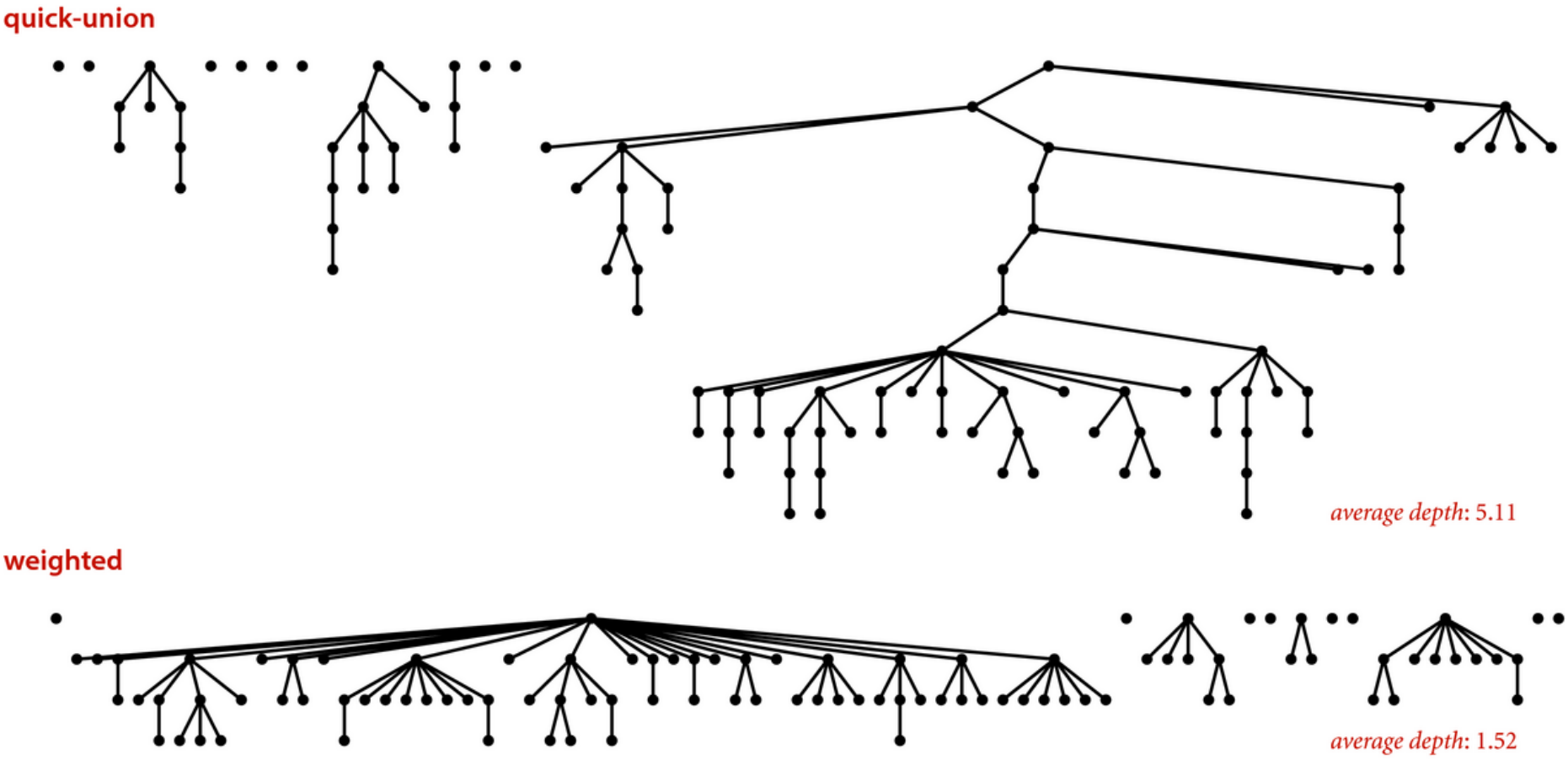
`ufInit(n)`	$\Theta(n)$
`ufFind(p)`	$O(\lg n)$
`ufUnion(p, q)`	$O(\lg n)$



Custo total da sequência:

$$\Theta(n) + m\,O(\lg n) + n\,O(\lg n) = O(m \lg n)$$

Ilustração



Quick-union and weighted quick-union (100 sites, 88 `union()` operations)

Experimentos

```
% time client < tinyUF.txt
```
2 components
0.0003seg

```
% time client < mediumUF.txt
```
3 components
0.005seg

```
% time client < largeUF.txt
```
6 components
3.726seg

Encurtamento de caminhos

Acrescentando uma linha a `ufFind()`
encurtamos o comprimento do caminho à metade.

```
int ufFind(int p) {
    while (p != pai[p]) {
        /* encurta caminho à metade */
        pai[p] = pai[pai[p]];
        p = pai[p];
    }
    return p;
}
```

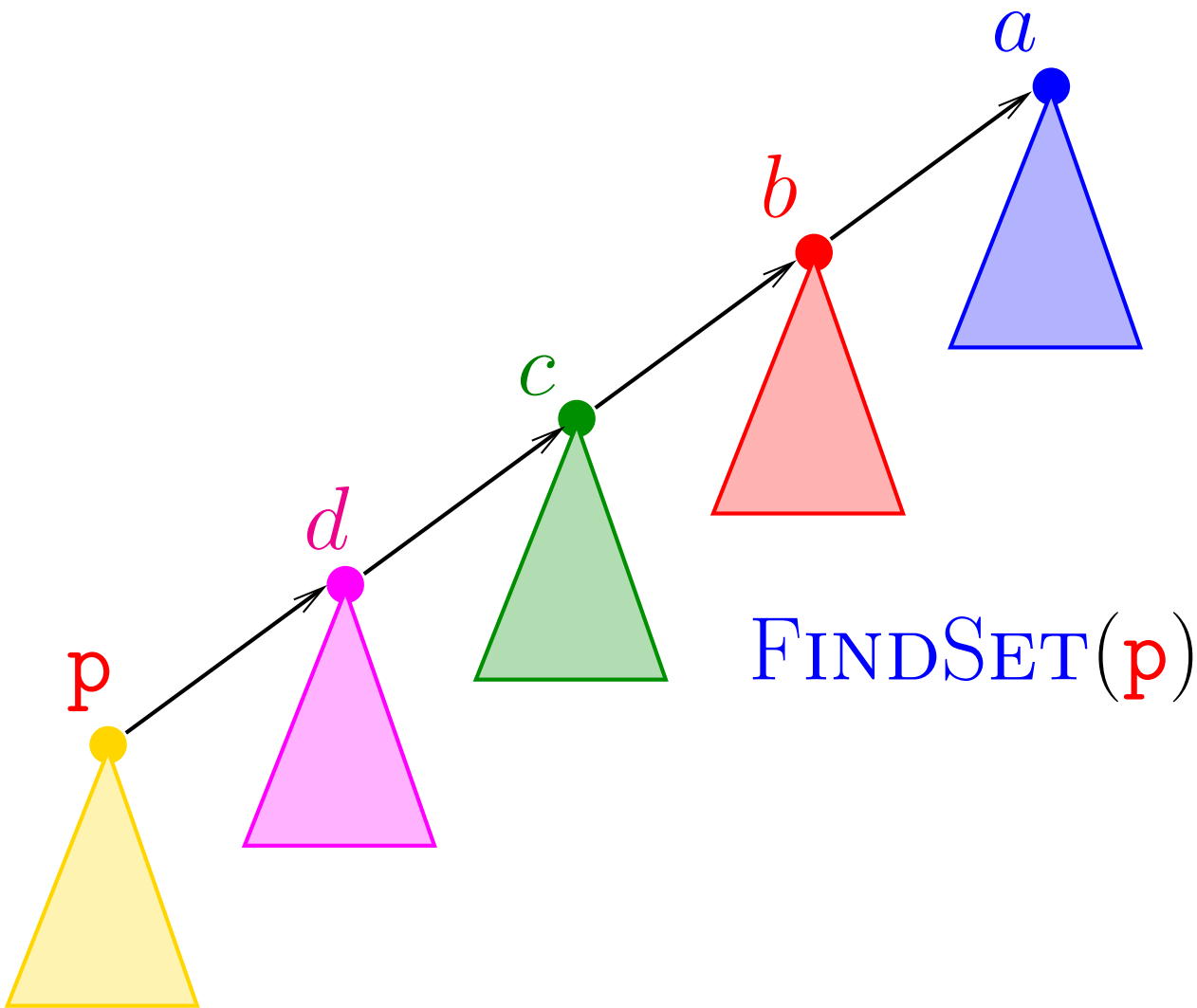
Mais experimentos

```
% time client < tinyUF.txt
2 components
0.0003seg

% time client < mediumUF.txt
3 components
0.004seg

% time client < largeUF.txt
6 components
3.511seg
```

Path compression

Ideia:

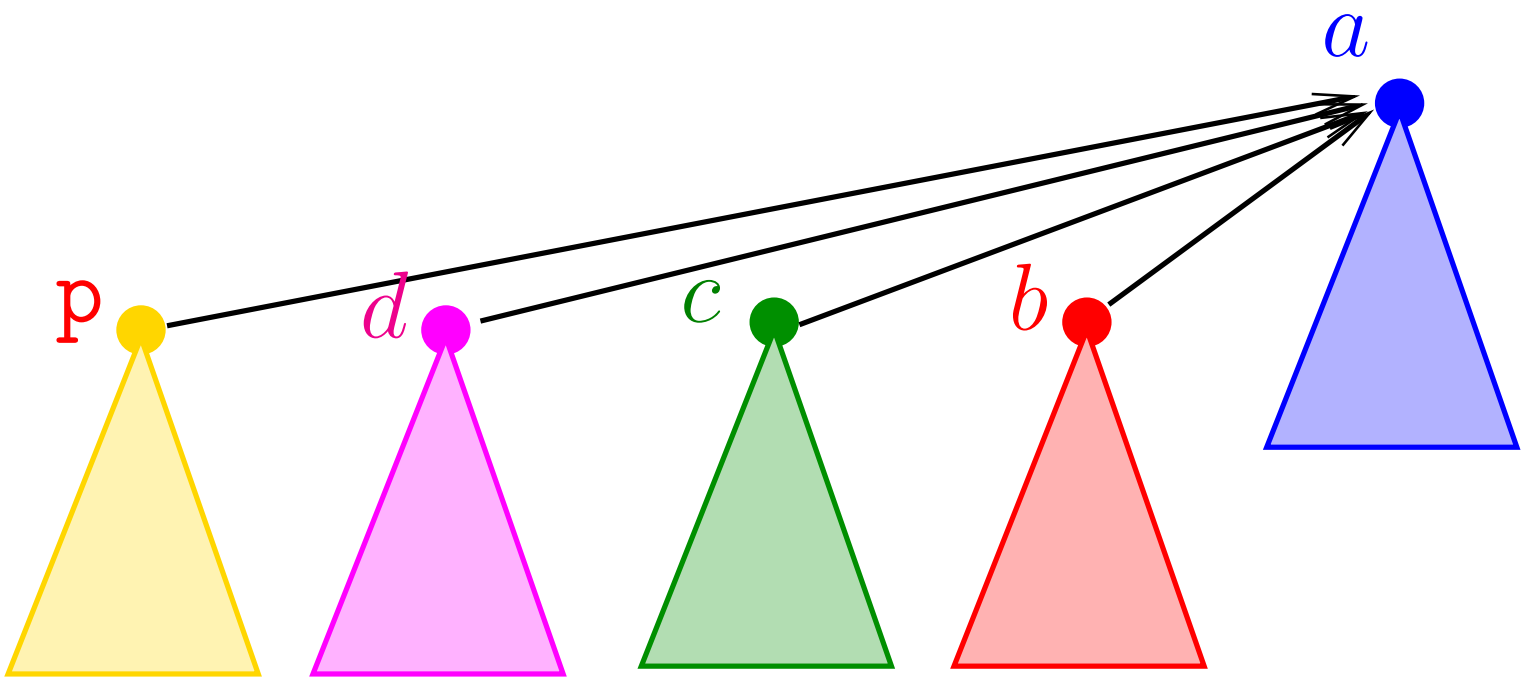
encurtar os caminhos durante cada `ufFind()`.



Path compression

Ideia:

encurtar os caminhos durante cada `ufFind()`.



FindSet(p)