MAC0323 Algoritmos e Estruturas de Dados II

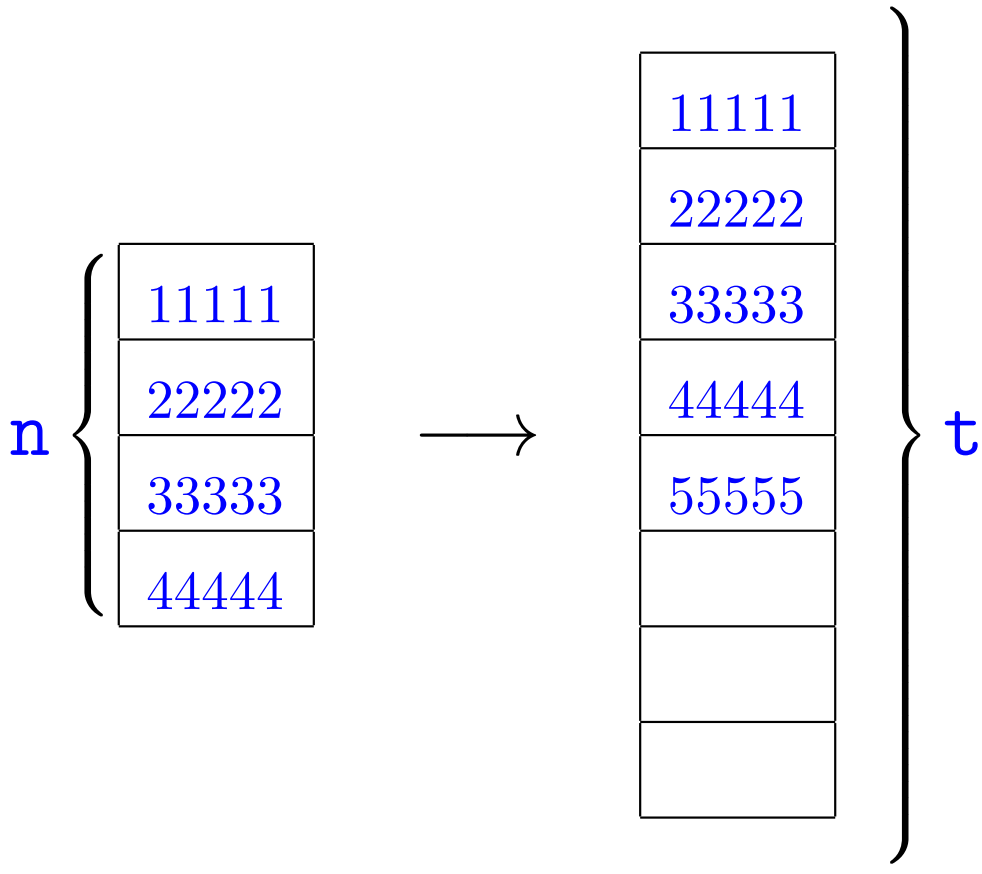
Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 2

Tabelas dinâmicas



$\texttt{n}[\texttt{T}]=$ número de itens $\qquad$ $\texttt{t}[\texttt{T}]=$ tamanho de $\texttt{T}$

Inicialmente $\texttt{n}[\texttt{T}] = \texttt{t}[\texttt{T}] = 0$

Sequência de $\mathtt{m}$ TABLE-INSERTs

$$\mathtt{T}_0 \xrightarrow{1^{\mathrm{a}}\ \mathrm{op}} \mathtt{T}_1 \xrightarrow{2^{\mathrm{a}}\ \mathrm{op}} \mathtt{T}_2 \longrightarrow \cdots \xrightarrow{\mathtt{m}^{\mathrm{a}}\ \mathrm{op}} \mathtt{T}_{\mathtt{m}}$$

$\mathtt{T}_i$ = estado de $\mathtt{T}$ depois da i^{a} operação.

Custo real da i^{a} operação:

$$c_i = \begin{cases} 1 & \text{se há espaço} \\ \mathtt{n}_i & \text{se tabela cheia,} \end{cases}$$

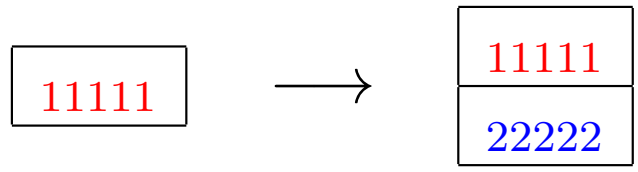
onde $\mathtt{n}_i$ = valor de $\mathtt{n}[\mathtt{T}]$ depois da i^{a} operação
$= i$.

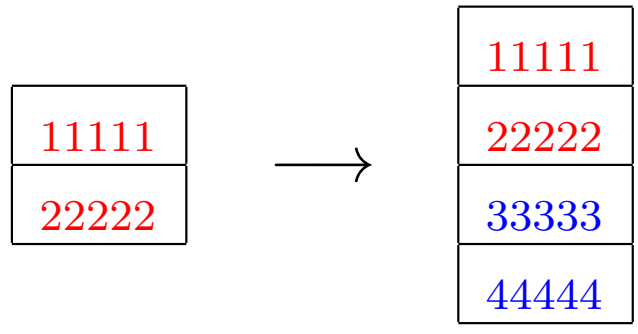
Custo de uma operação $= \mathrm{O}(\mathtt{m})$.

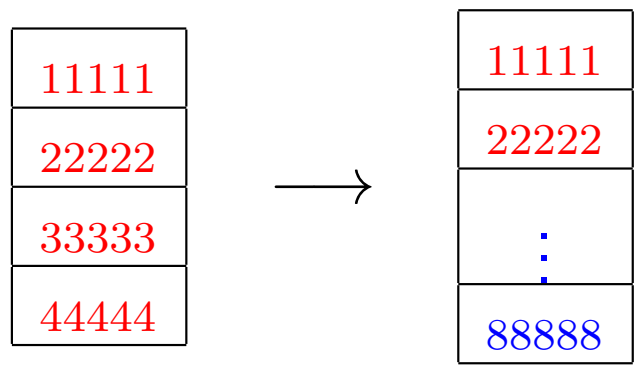
Custo das $\mathtt{m}$ operações $= \mathrm{O}(\mathtt{m}^2)$. Exagero!

Exemplo

n[T] (operação)	t[T]	custo
1	1	1
2	2	1+1
3	4	1+2
4	4	1
5	8	1+4
6	8	1
7	8	1
8	8	1
9	16	1+8
10	16	1
16	16	1
17	32	1+16
33	64	1+32







Custo amortizado

Custo total: Para $k = \lfloor \lg(\mathtt{m} - 1) \rfloor$,

$$\sum_{i=1}^{\mathtt{m}} c_i = \mathtt{m} + \sum_{i=0}^{k} 2^i = \mathtt{m} + 2^{k+1} - 1 < \mathtt{m} + 2\mathtt{m} - 1 < 3\mathtt{m}.$$

Custo amortizado:

$$\frac{3\mathtt{m}}{\mathtt{m}} = 3 = \Theta(1)$$

AULA 3

Pilhas redimensionáveis

Fonte: https://br.pinterest.com/

Pilha (= stack) e sua API (PF)

1.3 Bags, Queues, and Stacks (SW)

Pilhas redimensionáveis

Considere a implementação de saco (`Bag`) em vetor com redimensionamento.

O custo amortizado da operação `add`() é muito baixo, pois cada ocorrência de uma execução lenta de `add`() é precedida por muitas ocorrências de execuções rápidas.

Pilhas redimensionáveis

Arquivo	`stack.h`	
`void`	`stackInit()`	cria uma pilha de itens vazia
`void`	`stackPush(Item item)`	empilha o item
`Item`	`stackPop()`	desempilha um item
`bool`	`stackEmpty()`	a pilha está vazia?
`int`	`stackSize()`	número de itens na pilha
`void`	`stackFree()`	destroi a pilha

Interface stack.h

```
/* stack.h */

#include "Item.h"

#define bool int

void stackInit();
void stackPush(Item);
Item stackPop();
bool stackEmpty();
int  stackSize();
void stackFree();
```

Arquivo stack.c: esqueleto

```
#include "stack.h"

static Item *a; /* vetor para a pilha */
static int s;   /* tamanho atual do vetor a */
static int n;   /* número de itens na pilha */

void stackInit() {...}
void stackPush(Item item) {...}
Item stackPop() {...}
bool stackEmpty() {...}
int  stackSize() {...}
void stackFree() {...}
```

Cliente

```
int main(int argc, char *argv[]) {
   int i;
   stackInit();
   for (i = 1; i < argc; i++)
      if (strcmp(argv[i], "-") != 0)
         stackPush(argv[i]);
```

Cliente

```
int main(int argc, char *argv[]) {
   int i;
   stackInit();
   for (i = 1; i < argc; i++)
      if (strcmp(argv[i], "-") != 0)
         stackPush(argv[i]);
      else if (!stackEmpty())
         printf("%s + ", stackPop());
```

Cliente

```
int main(int argc, char *argv[]) {
   int i;
   stackInit();
   for (i = 1; i < argc; i++)
      if (strcmp(argv[i], "-") != 0)
         stackPush(argv[i]);
      else if (!stackEmpty())
         printf("%s + ", stackPop());
   printf("(%d left on stack)\n",
                  stackSize());
   stackFree();
   return 0;
}
```

Implementação: stack.c

```
static Item *a;

static int s;

static int n;

static void *mallocSafe(int);

void stackInit() {
   a = mallocSafe(sizeof(Item));
   s = 1;
   n = 0;
}
```

stackEmpty(), stackSize() e stackFree()

```
int stackEmpty() {
   return n == 0;
}

int stackSize() {
   return n;
}

void stackFree() {
   free(a);
   a = NULL;    /* evita loitering */
}
```

stackPush() e stackPop()

```
static void stackResize(int capacity);

void stackPush(Item item) {
   if (n == s) stackResize(2*s);
   a[n++] = item;
}
```

stackPush() e stackPop()

```
static void stackResize(int capacity);

void stackPush(Item item) {
   if (n == s) stackResize(2*s);
   a[n++] = item;
}

Item stackPop() {
   Item item = a[n-1];
   n--;
```

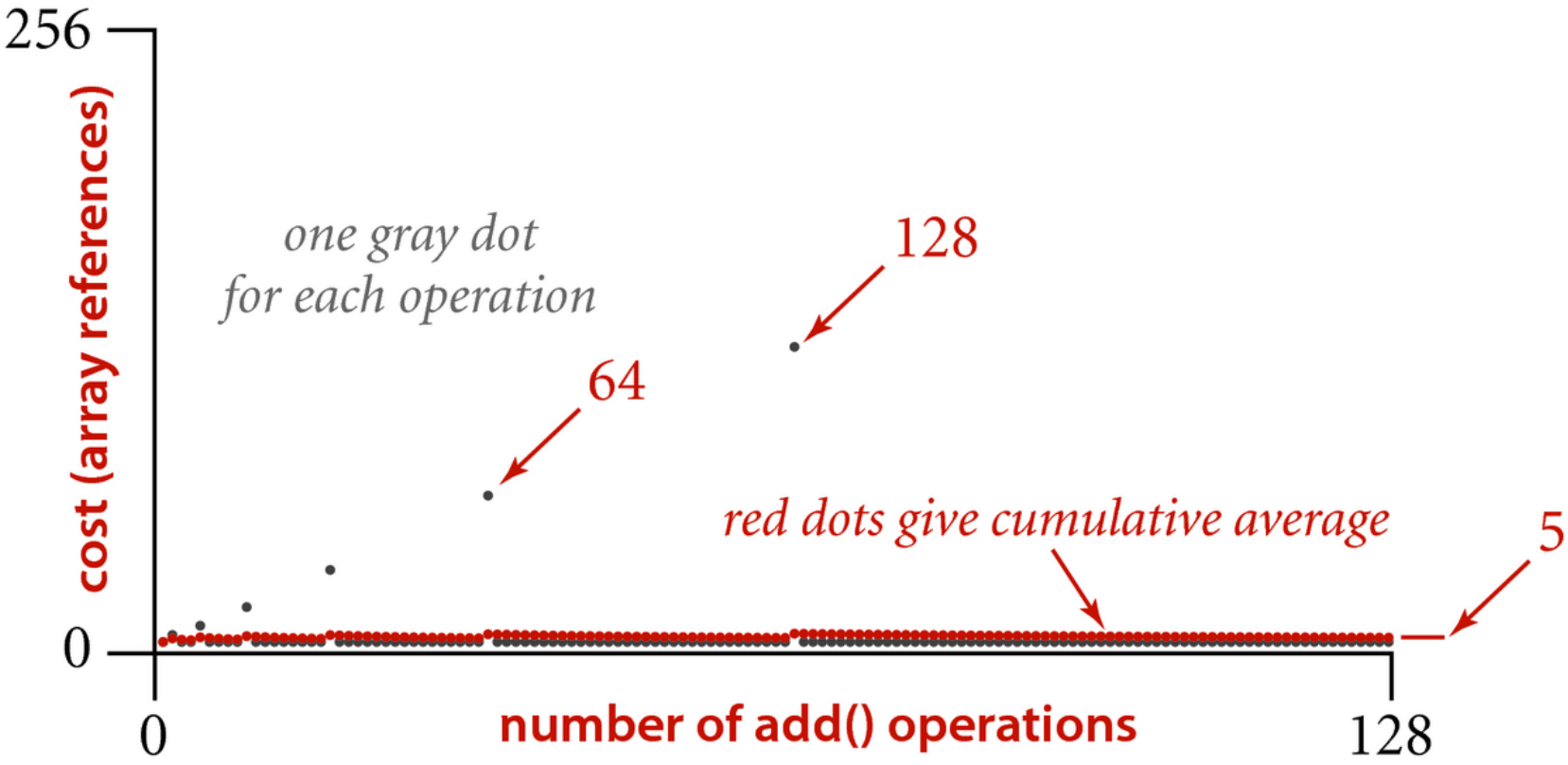
stackPush() e stackPop()

```
static void stackResize(int capacity);

void stackPush(Item item) {
   if (n == s) stackResize(2*s);
   a[n++] = item;
}

Item stackPop() {
   Item item = a[n-1];
   n--;

   /* shrink size of array if necessary */
   if (n > 0 && n == s/4)
      stackResize(s/2);
   return item;
}
```

stackResize()

```
static void stackResize(int capacity) {
   int i;
   Item *tmp, *b;

   b = mallocSafe(capacity*sizeof(Item));
   for(i = 0; i < n; i++)
      b[i] = a[i];
   s = capacity;
   tmp = a;
   a = b;
   free(tmp);
}
```

Bags redimensionáveis



Amortized cost of adding to a `RandomBag`

Conectividade dinâmica

Fonte: Wifi humor

1.5 Case Study: Union-Find

Leitura, vídeos, ...

Leitura: Case Study: Union-Find, S&W

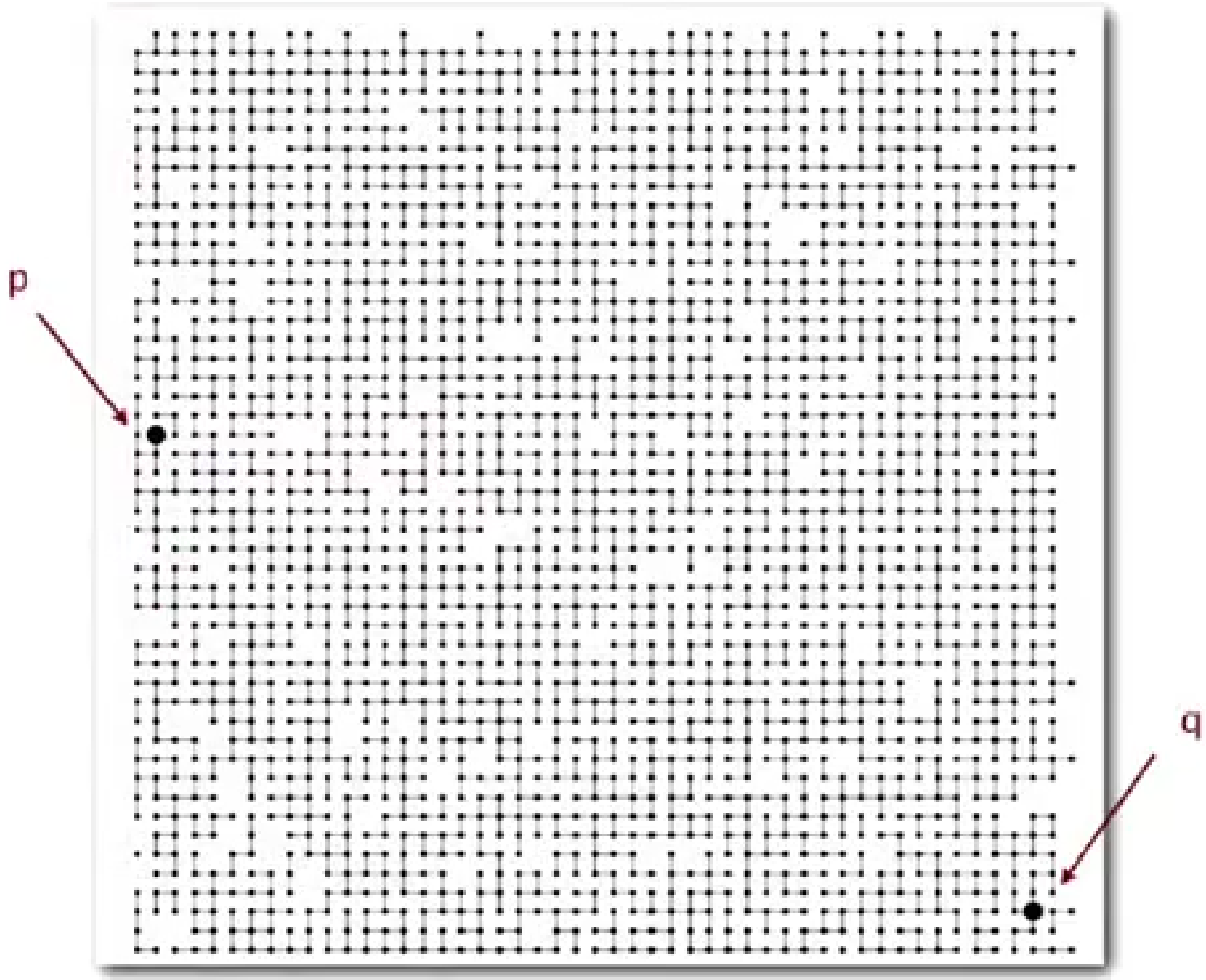
Vídeos:
Union-find e Kruskal, Gabriel Russo, canal BCC e
Union-find, Robert Sedgewick.

Considere uma coleção de conjuntos disjuntos.

Conjuntos são modificados ao longo do tempo.

Terminologia utiliza metáfora de redes:
sítios/sites, conexão, ...

Problema: p e q estão ligados?



Fonte: algs4

Conjuntos disjuntos

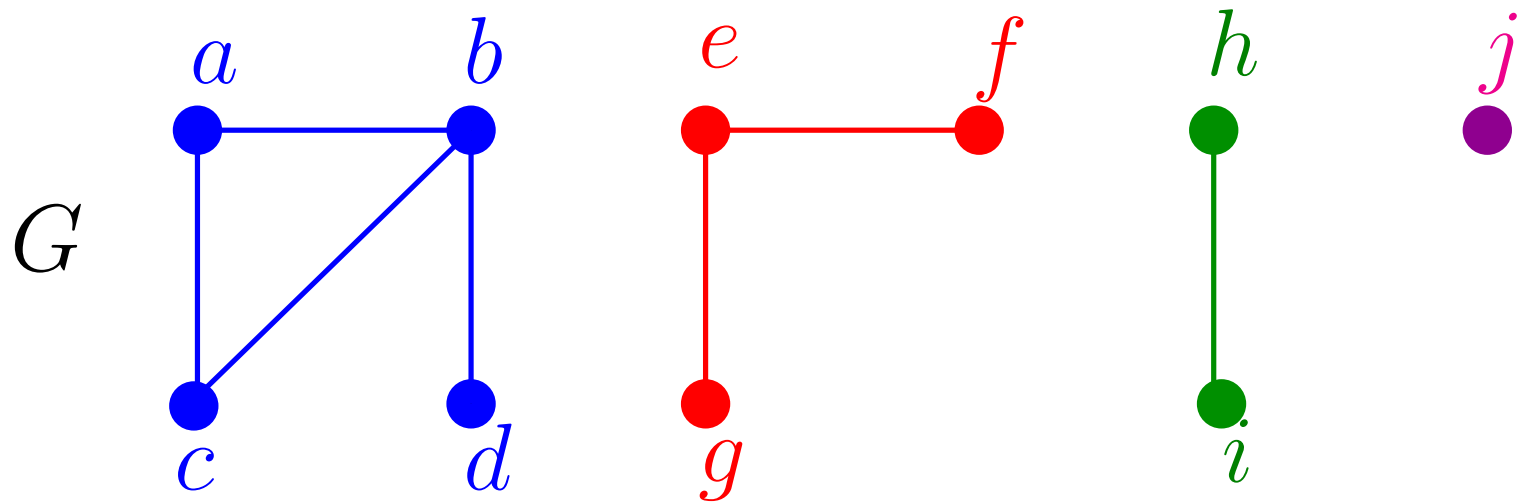
Seja $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$
uma coleção de conjuntos disjuntos, ou seja,

$$S_i \cap S_j = \emptyset$$

para todo $i \neq j$.

Conjuntos disjuntos

Exemplo de coleção disjunta de conjuntos:
componentes conexos de um grafo



componentes formam conjuntos disjuntos de vértices

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

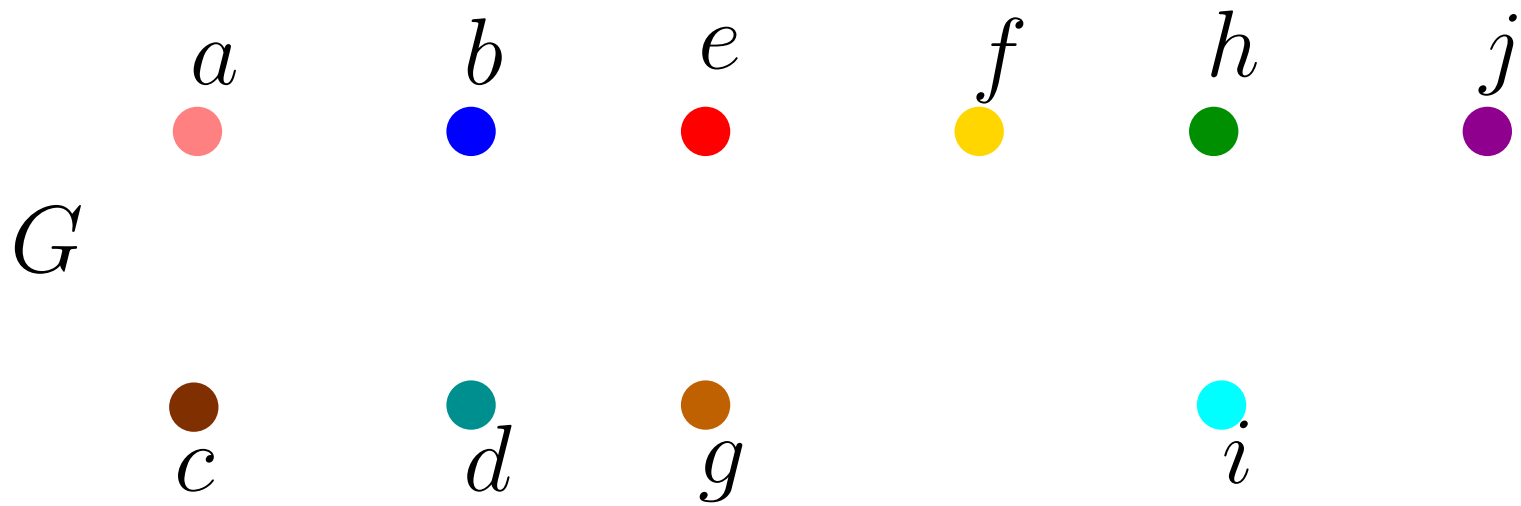
Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

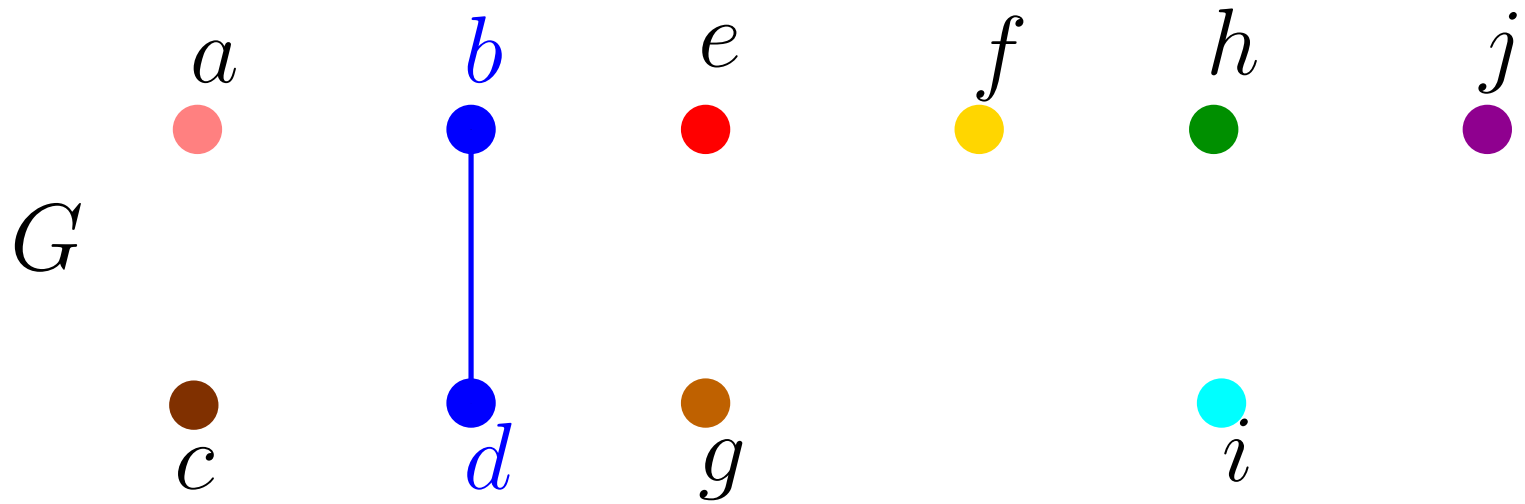


aresta	componentes
	$\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

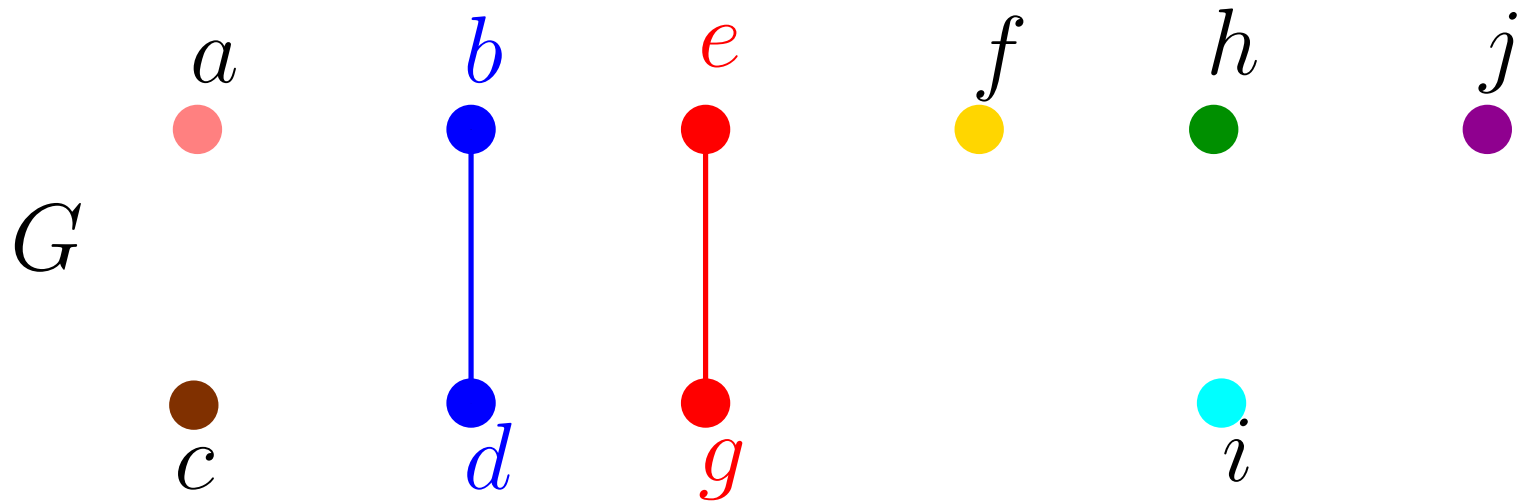


aresta	componentes								
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

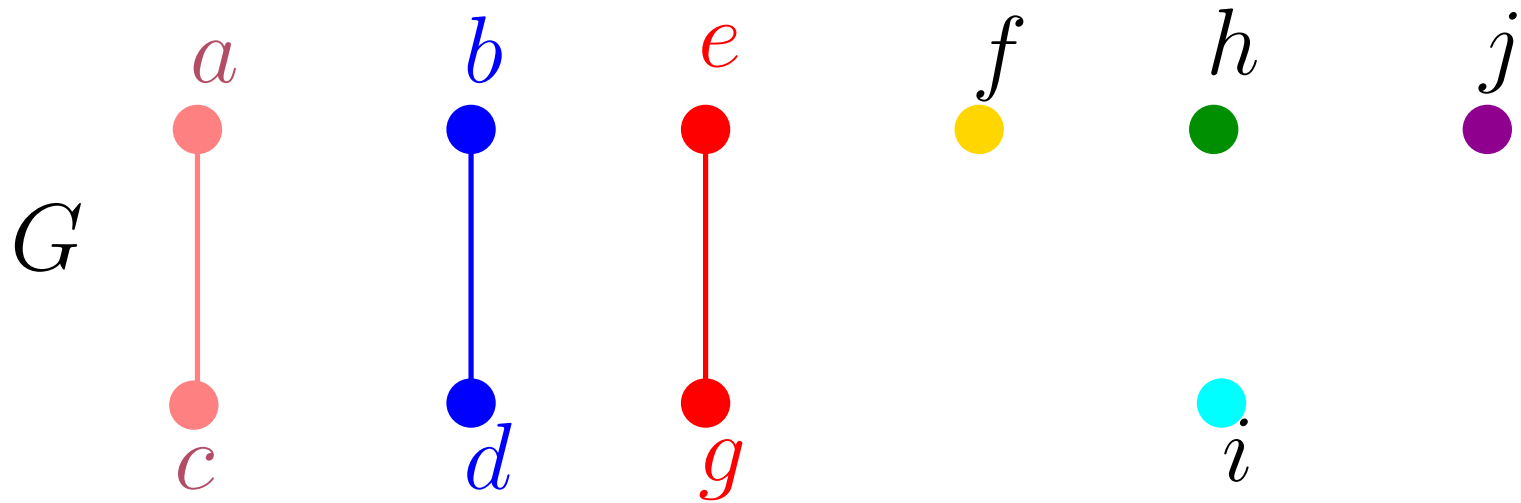


aresta	componentes
(e, g)	$\{a\}$ $\{b, d\}$ $\{c\}$ $\{e, g\}$ $\{f\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

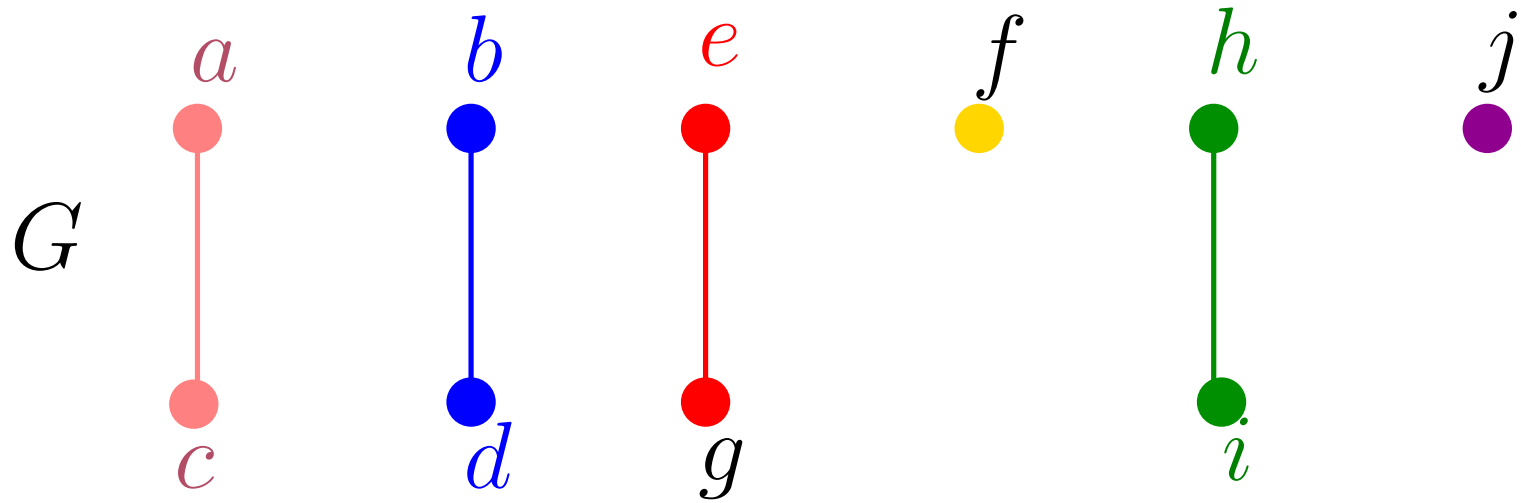


aresta	componentes
(a, c)	$\{a, c\}$ $\{b, d\}$ $\{e, g\}$ $\{f\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

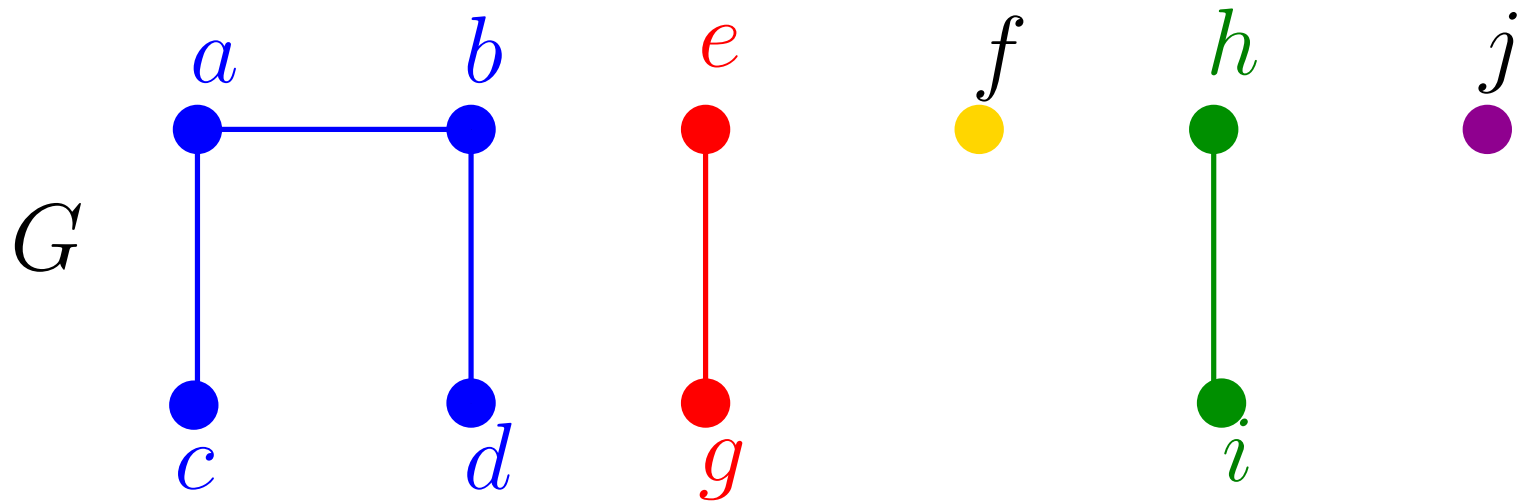


aresta	componentes
(h, i)	$\{a, c\}$ $\{b, d\}$ $\{e, g\}$ $\{f\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

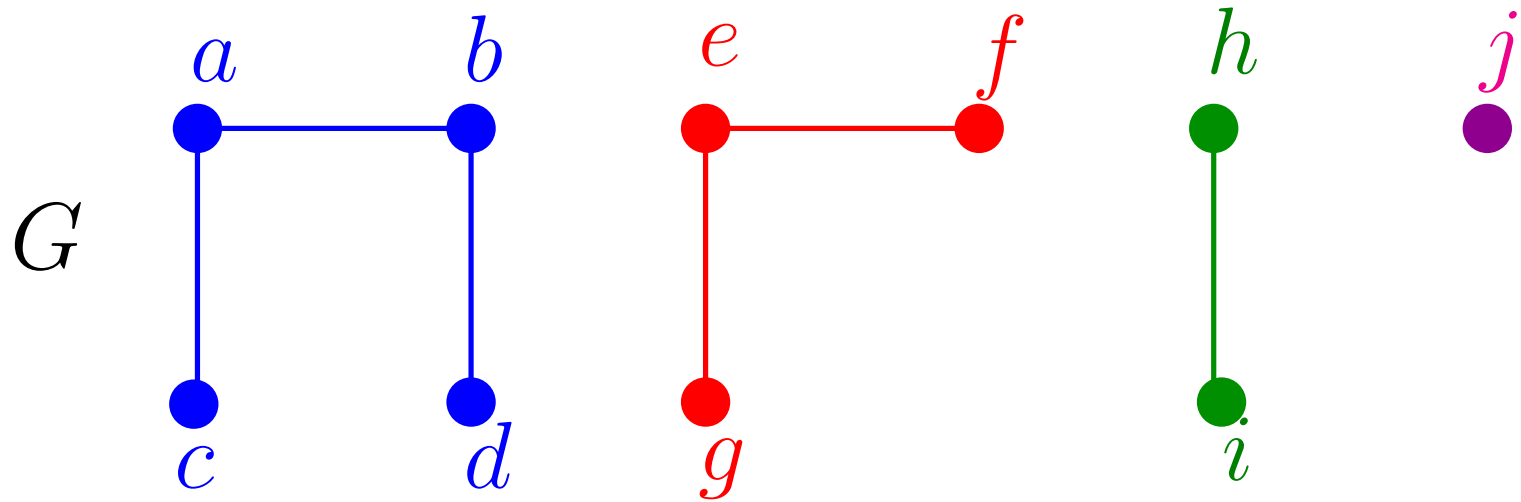


aresta	componentes
(a, b)	$\{a, b, c, d\}$ $\{e, g\}$ $\{f\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

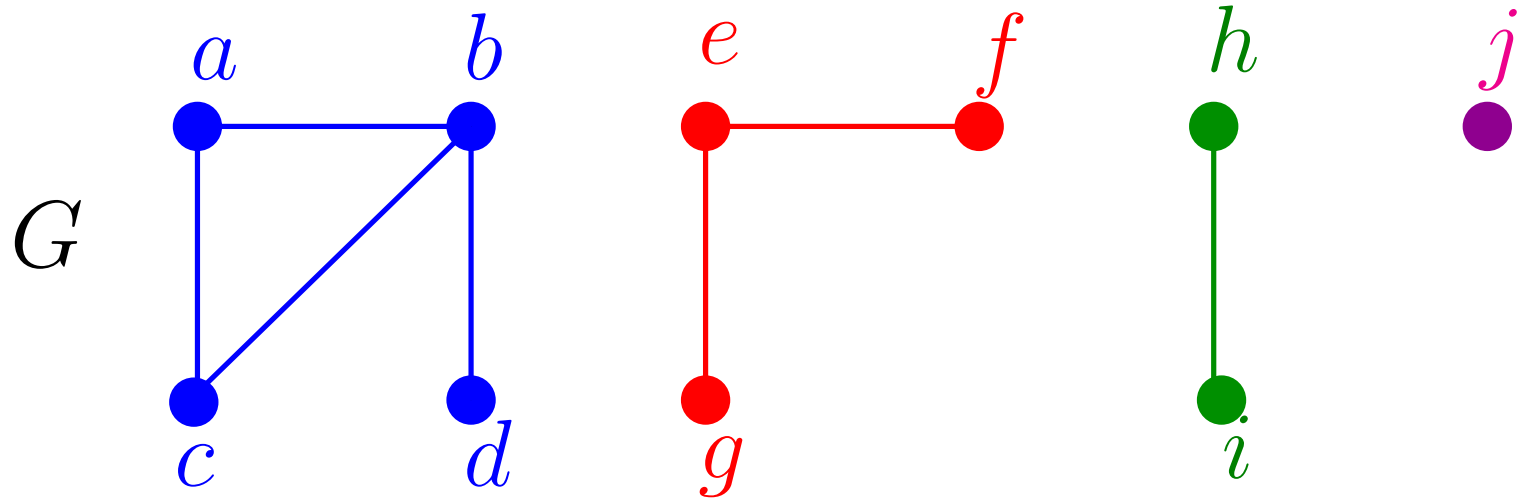


aresta	componentes
(e, f)	$\{a, b, c, d\}$ $\{e, f, g\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico



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

Operações básicas

$\mathcal{S}$ coleção de conjuntos disjuntos.

Cada conjunto tem um **representante**.

MakeSet (x): x é elemento novo
$\mathcal{S} \leftarrow \mathcal{S} \cup \{\{x\}\}$

Operações básicas

$\mathcal{S}$ coleção de conjuntos disjuntos.

Cada conjunto tem um **representante**.

MakeSet (x): x é elemento novo
$\mathcal{S} \leftarrow \mathcal{S} \cup \{\{x\}\}$

Union (x, y): x e y em conjuntos diferentes
$\mathcal{S} \leftarrow \mathcal{S} - \{S_x, S_y\} \cup \{S_x \cup S_y\}$
x está em S_x e y está em S_y

Operações básicas

$\mathcal{S}$ coleção de conjuntos disjuntos.

Cada conjunto tem um **representante**.

MakeSet (x): x é elemento novo
$\mathcal{S} \leftarrow \mathcal{S} \cup \{\{x\}\}$

Union (x, y): x e y em conjuntos diferentes
$\mathcal{S} \leftarrow \mathcal{S} - \{S_x, S_y\} \cup \{S_x \cup S_y\}$
x está em S_x e y está em S_y

FindSet (x): devolve representante do conjunto que contém x

Connected-Components

Recebe um grafo G e contrói uma representação dos componentes conexos de G.

```
CONNECTED-COMPONENTS (G)
1  para cada vértice v de G faça
2      MAKESET (v)
3  para cada aresta (u, v) de G faça
4      se FINDSET (u) ≠ FINDSET (v)
5          então UNION (u, v)
```

Detalhes de implementação:
objeto representando vértice u aponta para a representação de u como conjunto, e vice-versa.

Consumo de tempo

`n` := número de vértices do grafo

`m` := número de arestas do grafo

linha	consumo de todas as execuções da linha

Consumo de tempo

$n :=$ número de vértices do grafo

$m :=$ número de arestas do grafo

linha	consumo de todas as execuções da linha
1	$= \Theta(n)$
2	$= n \times$ consumo de tempo MakeSet
3	$= \Theta(m)$
4	$= 2m \times$ consumo de tempo FindSet
5	$< n \times$ consumo de tempo Union

$$\begin{aligned} \text{total} \leq \Theta(n+m) &+ n \times \text{consumo de tempo MakeSet} \\ &+ 2m \times \text{consumo de tempo FindSet} \\ &+ n \times \text{consumo de tempo Union} \end{aligned}$$

Same-Component

Decide se u e v estão no mesmo componente:

```
SAME-COMPONENT (u, v)
1   se FINDSET (u) = FINDSET (v)
2       então devolva SIM
3       senão devolva NÃO
```

Algoritmo de Kruskal

Encontra **árvore geradora mínima** (CLRS 23).

MST-KRUSKAL (G, w) $\quad \triangleright$ G conexo

```
0   coloque arestas em ordem crescente de w
1   A ← ∅
2   para cada vértice v faça
3       MAKESET (v)
4   para cada aresta uv em ordem crescente de w faça
5       se NÃO SAME-COMPONENT (u, v)
6           então UNION (u, v)
7                 A ← A ∪ {uv}
8   devolva A
```

"Avô" de todos os algoritmos gulosos.

Conjuntos disjuntos dinâmicos

Sequência de operações

MakeSet, Union, FindSet

$$\underbrace{\underbrace{M \quad M \quad M}_{n} \quad U \quad F \quad U \quad U \quad F \quad U \quad F \quad F \quad F \quad U \quad F}_{m}$$

Que estrutura de dados usar?

Compromissos (*trade-off*s).

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

Cliente

```
#include "uf.h"
int main() {
    int n, p, q;
    scanf("%d", &n);
    ufInit(n);
    while (!feof(stdin)) {
        scanf("%d %d", &p, &q);
        if (ufConnected(p, q)) continue;
        ufUnion(p, q);
        printf("%d + %d\n", p, q);
    }
    printf("%d componentes\n", ufCount());
    ufFree();
    return 0;
}
```

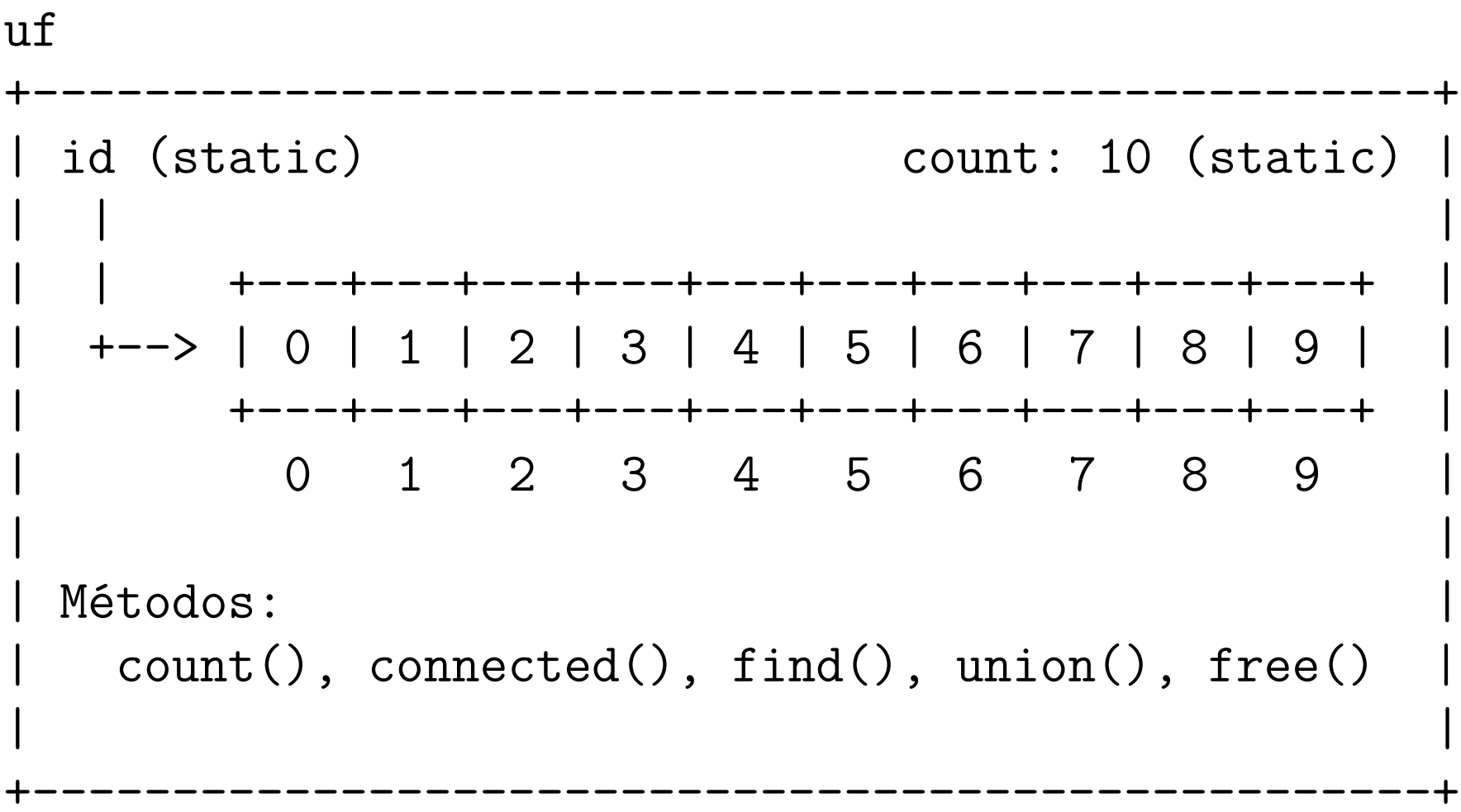
Quick-find

Fonte: Youtube

1.5 Case Study: Union-Find

QuickFindUF

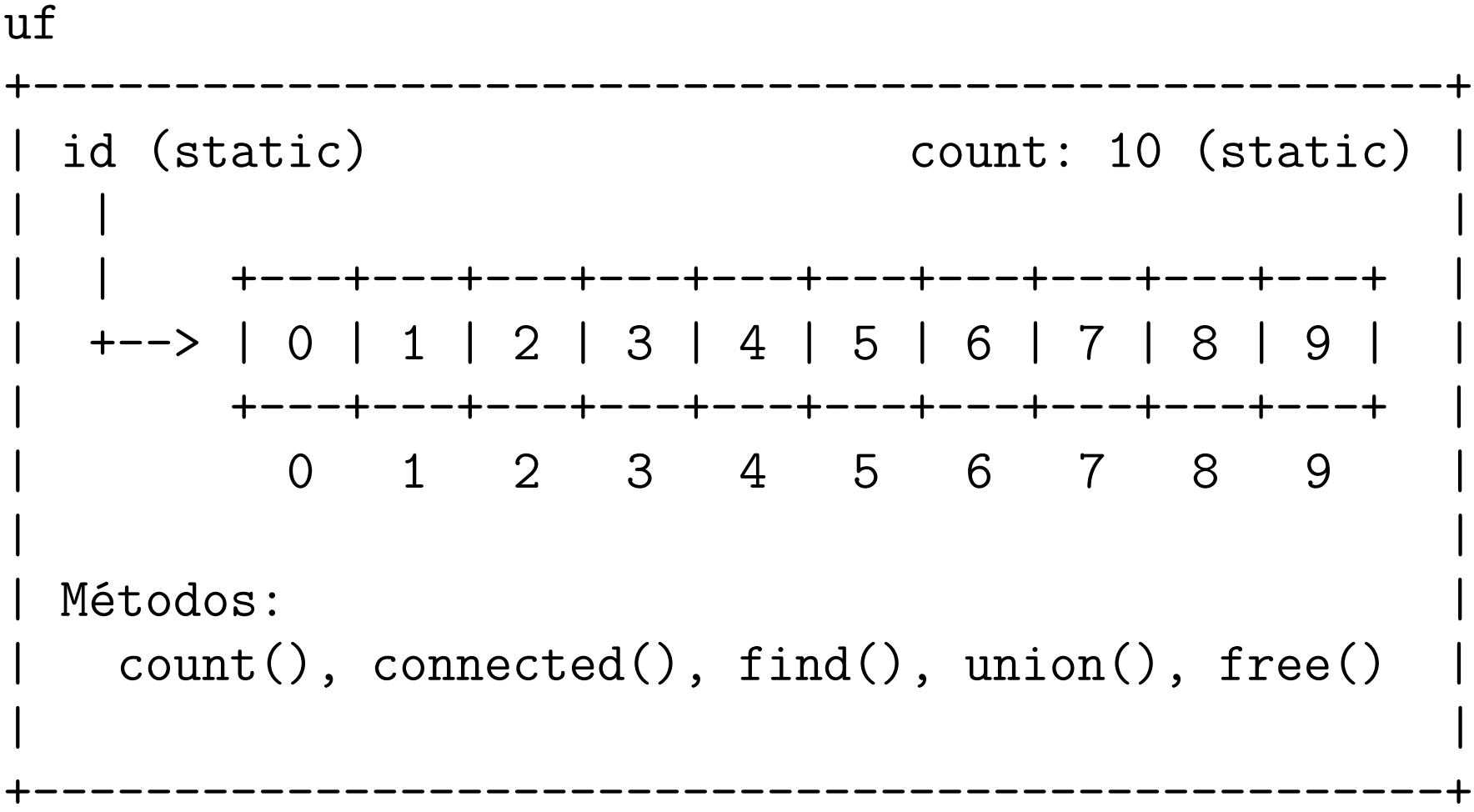
`ufInit`(10)



QuickFindUF

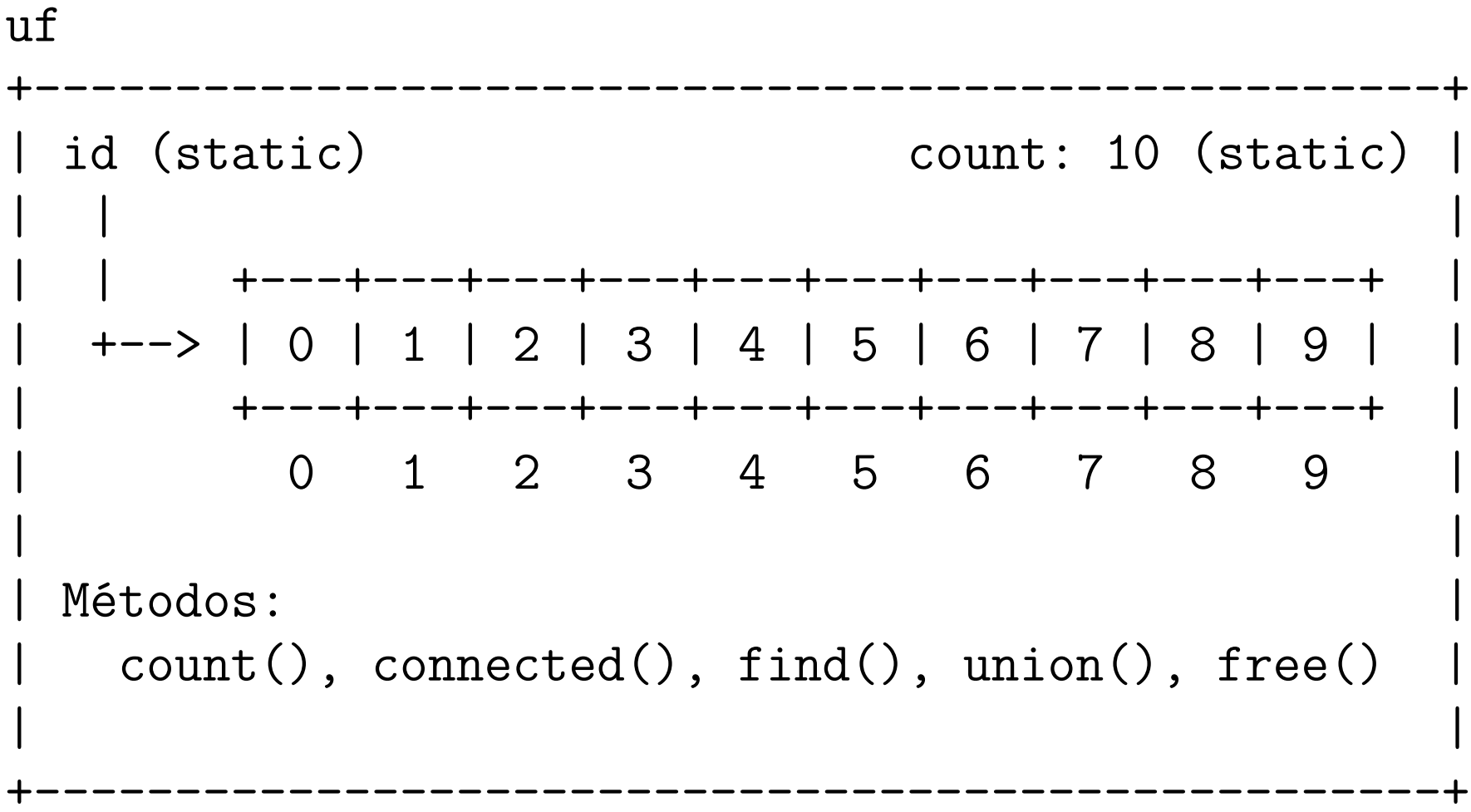
`ufFind`(3) retorna 3

`ufFind`(0) retorna 0



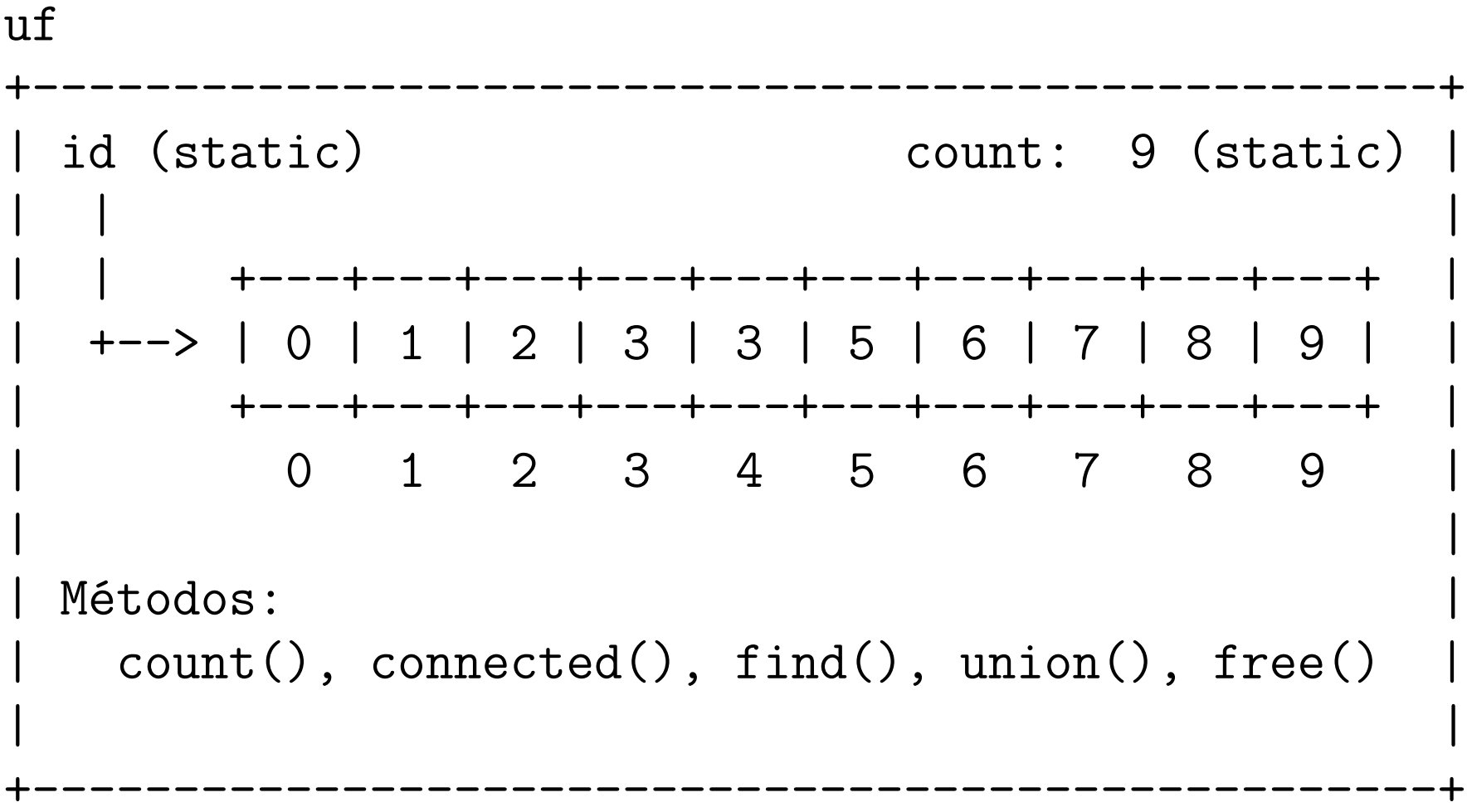
QuickFindUF

`ufUnion`(4, 3)?



QuickFindUF

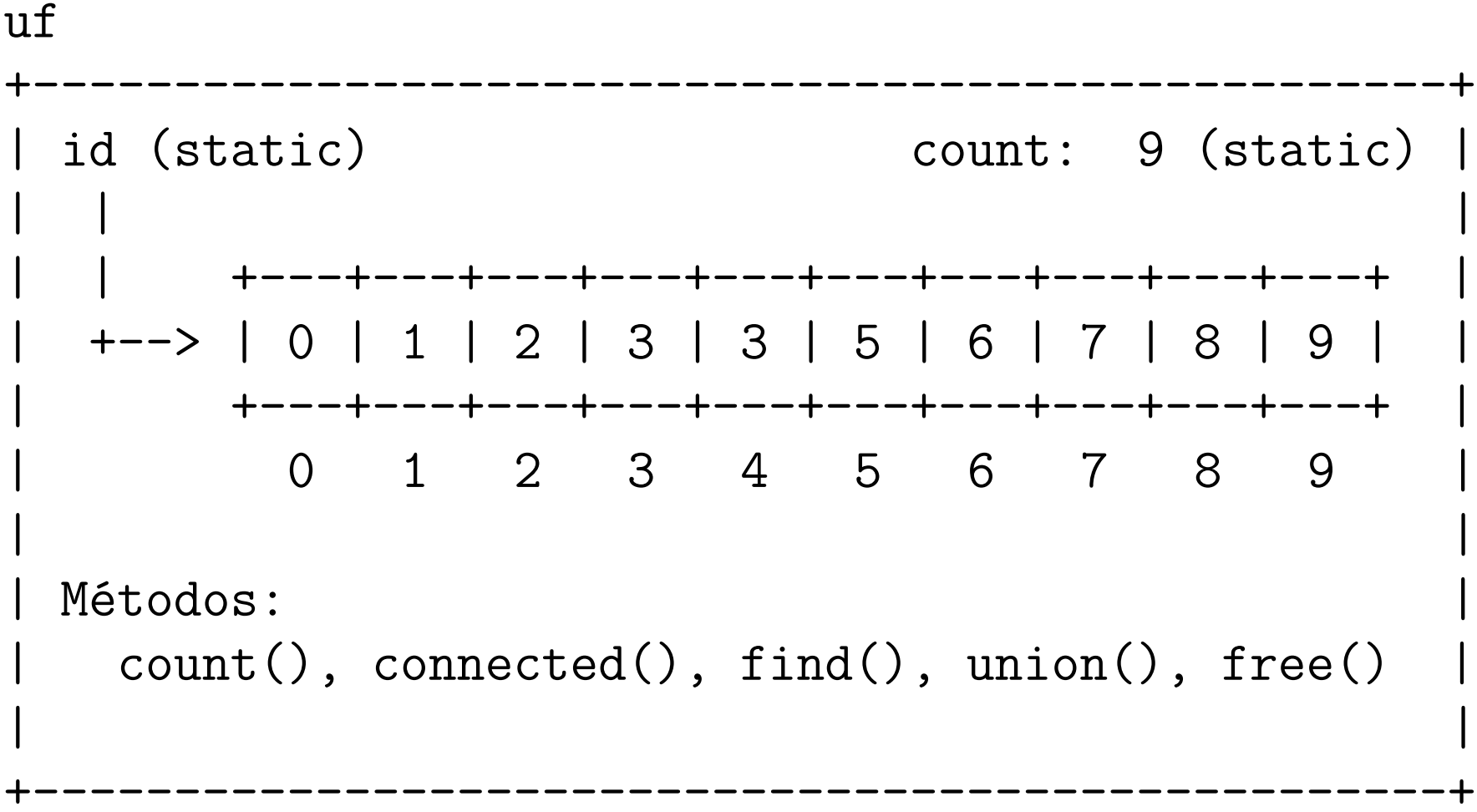
ufUnion(4, 3)



QuickFindUF

`ufUnion`(4, 3)

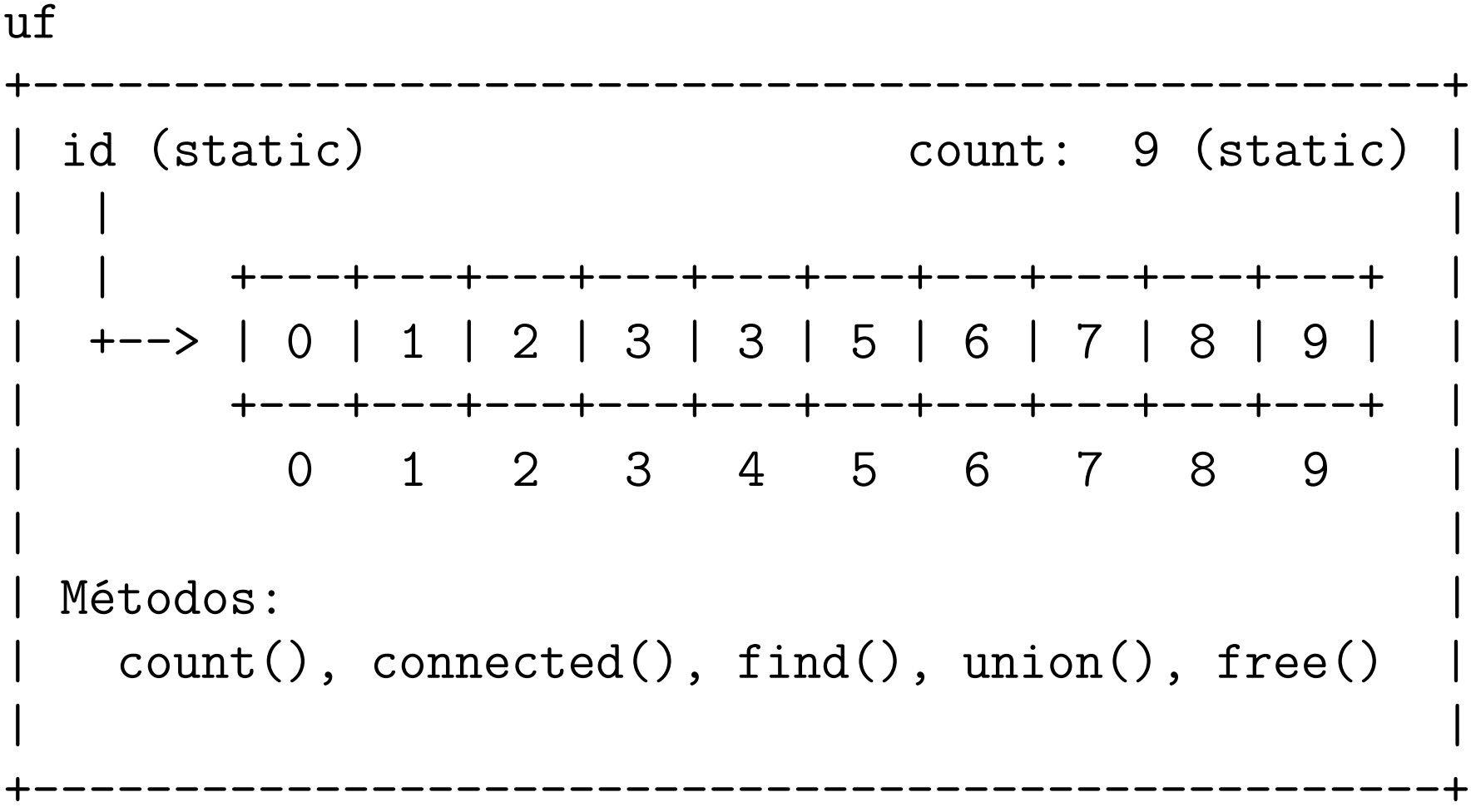
`ufFind`(3) retorna 3 `ufFind`(4) retorna 3



QuickFindUF

`ufFind`(3) retorna 3 `ufFind`(4) retorna 3

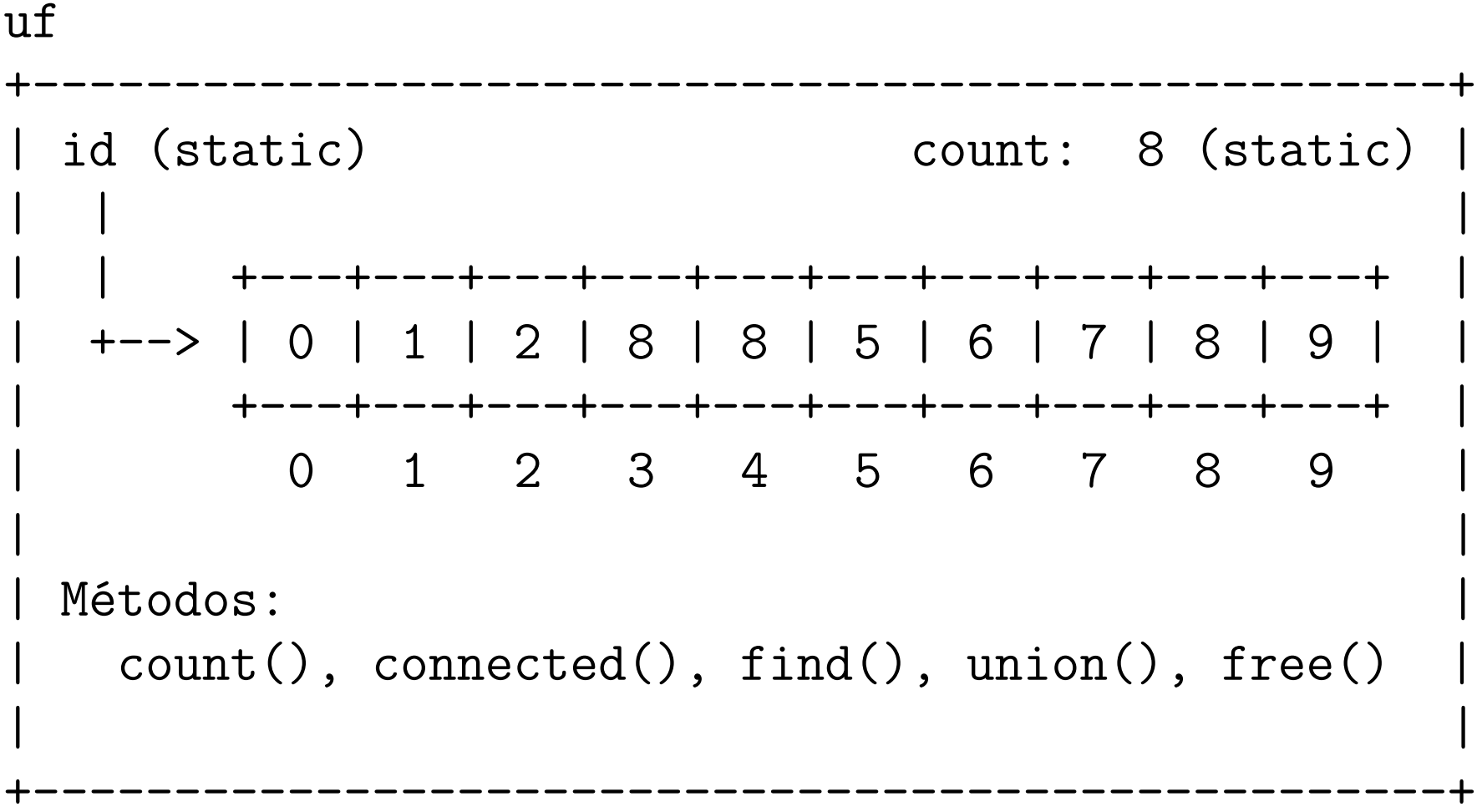
`ufUnion`(3, 8)?



QuickFindUF

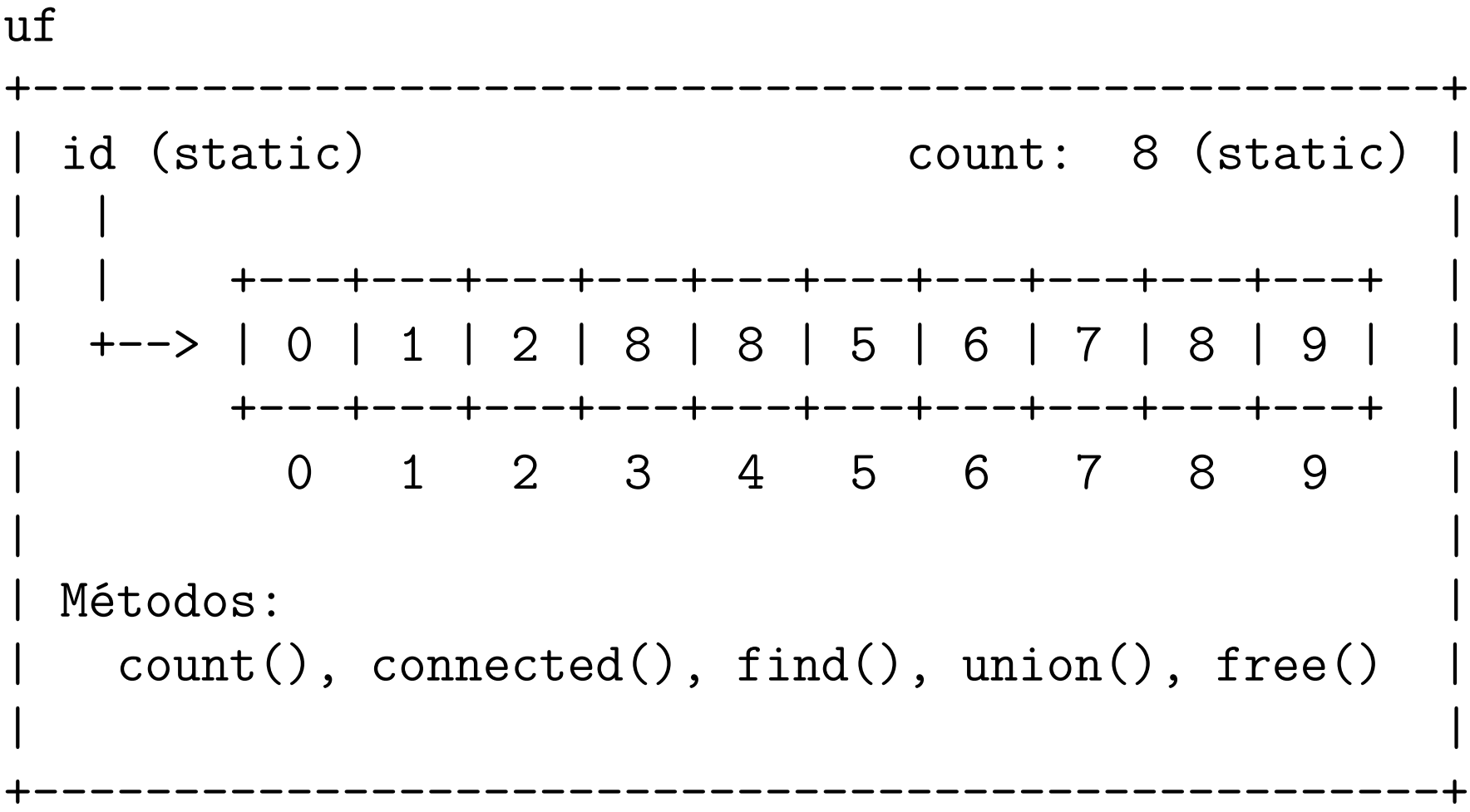
`ufFind`(3) retorna 3 `ufFind`(4) retorna 3

`ufUnion`(3, 8)



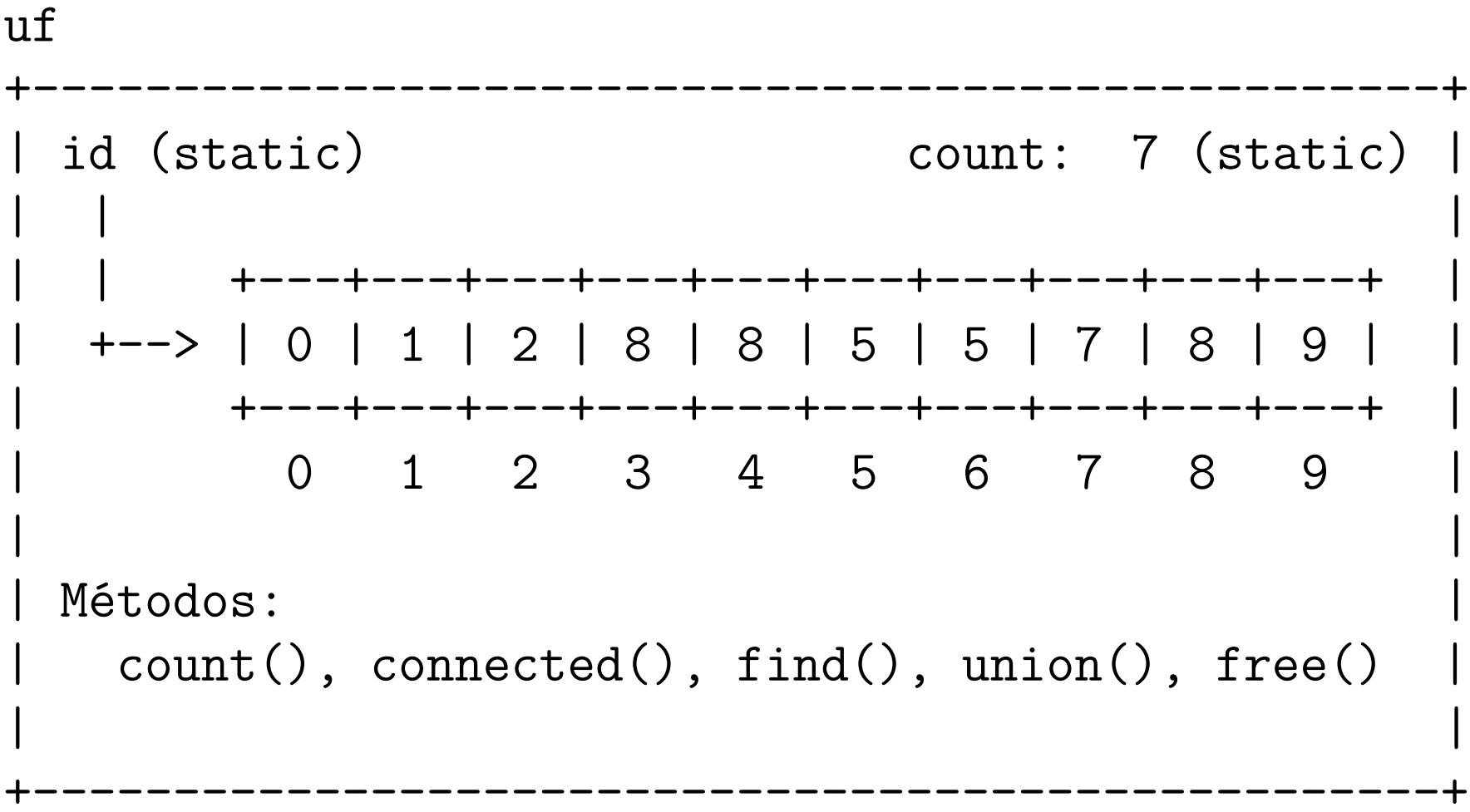
QuickFindUF

`ufUnion`(6, 5)?



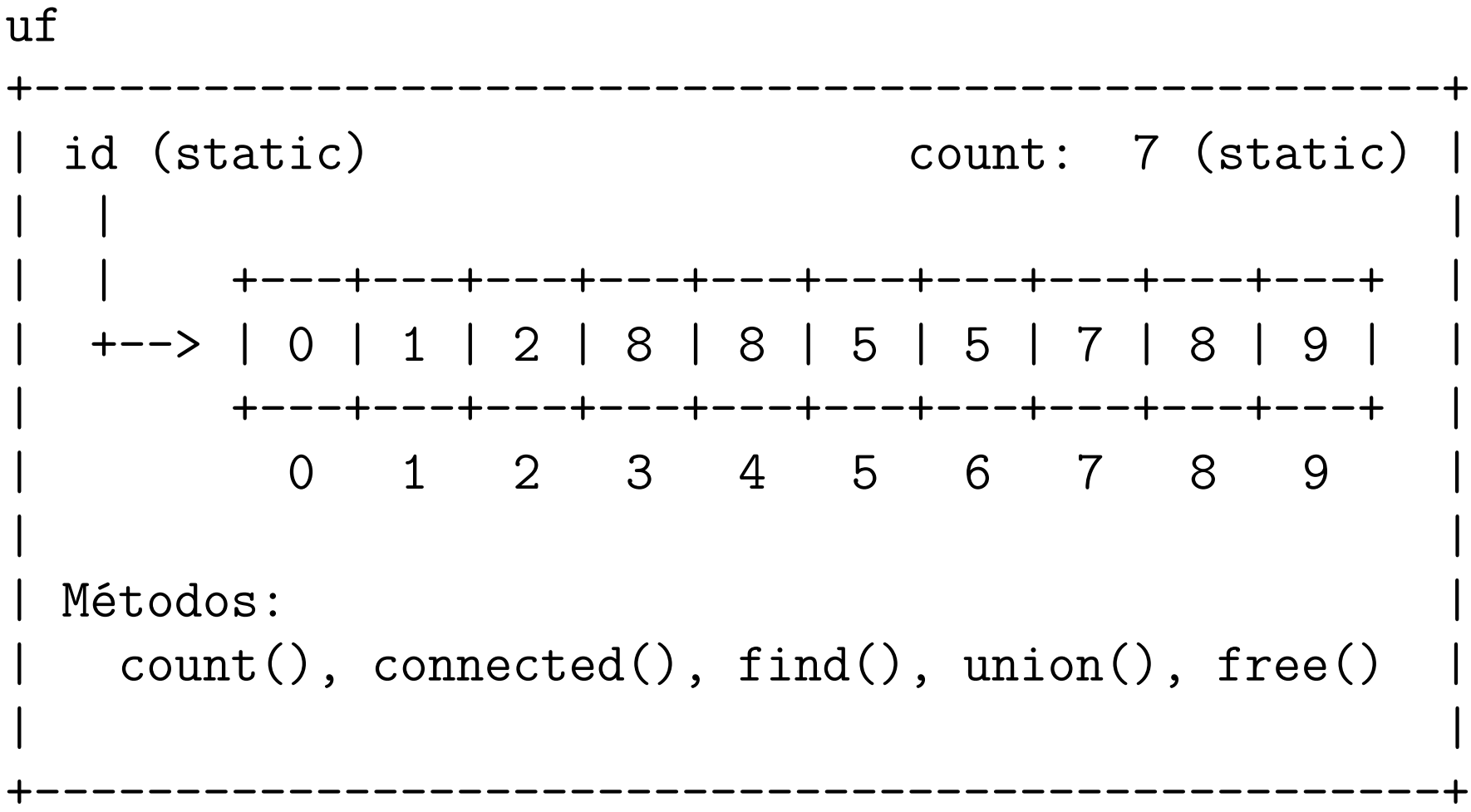
QuickFindUF

`ufUnion`(6, 5)



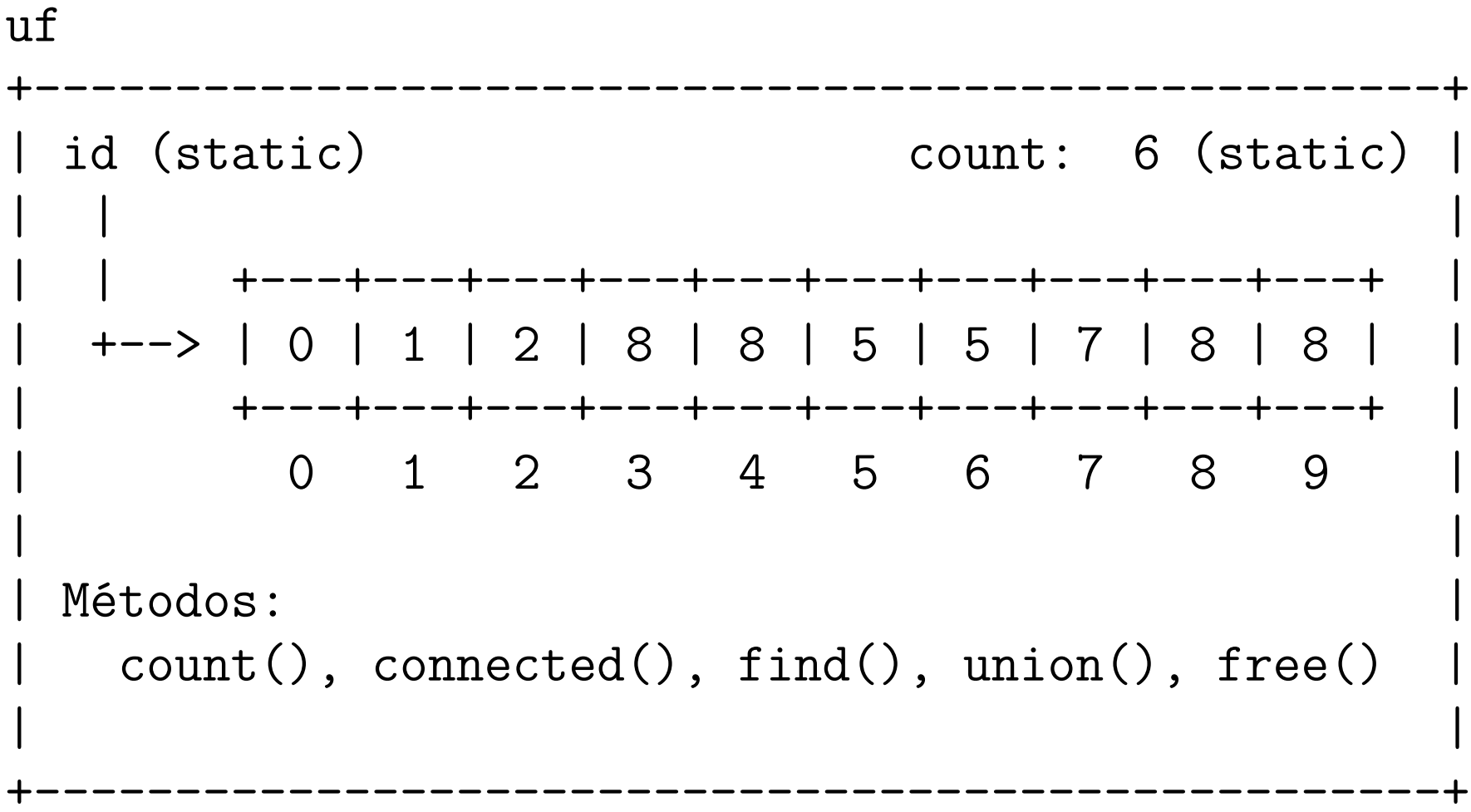
QuickFindUF

`ufUnion`(9, 4)?



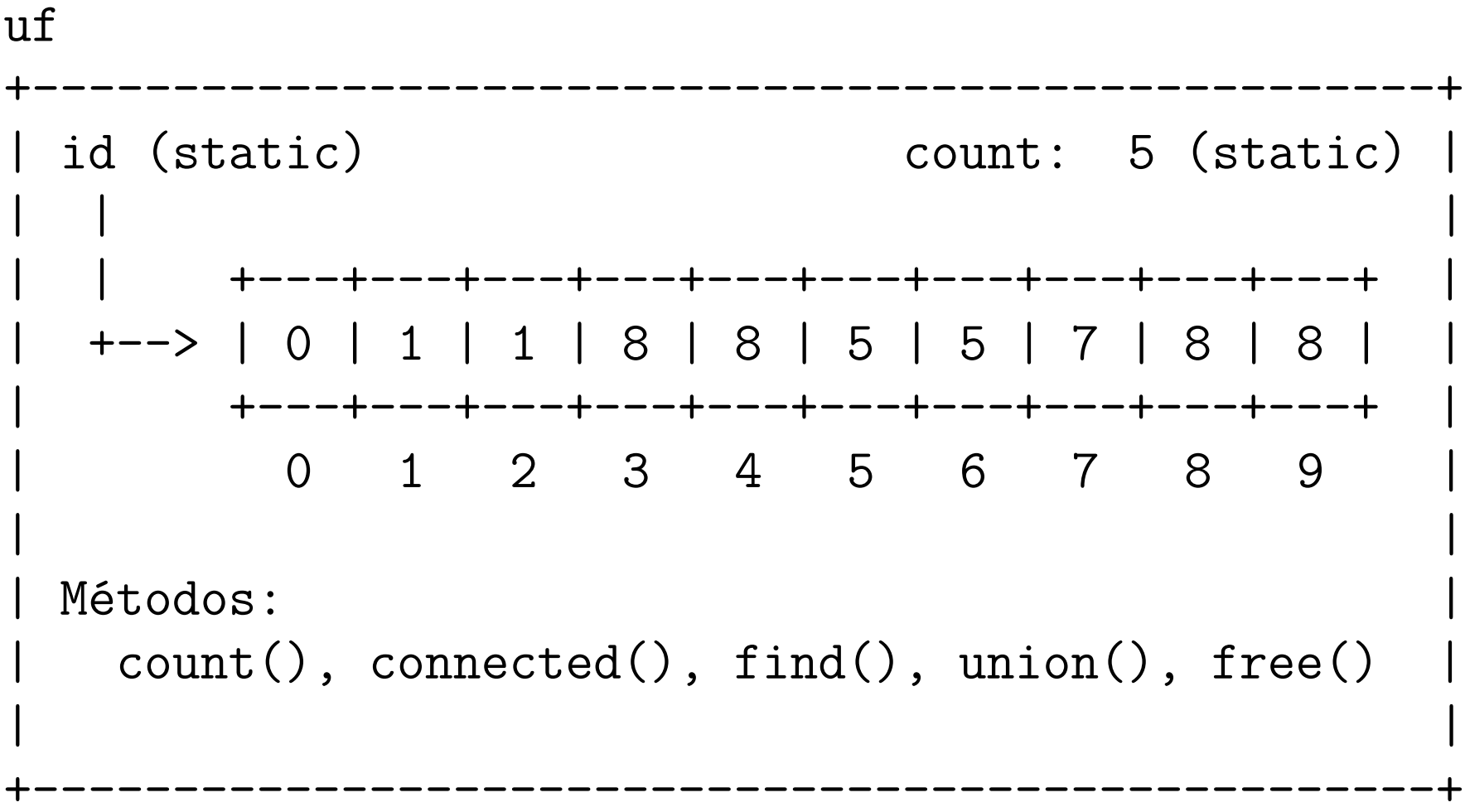
QuickFindUF

`ufUnion`(9, 4)



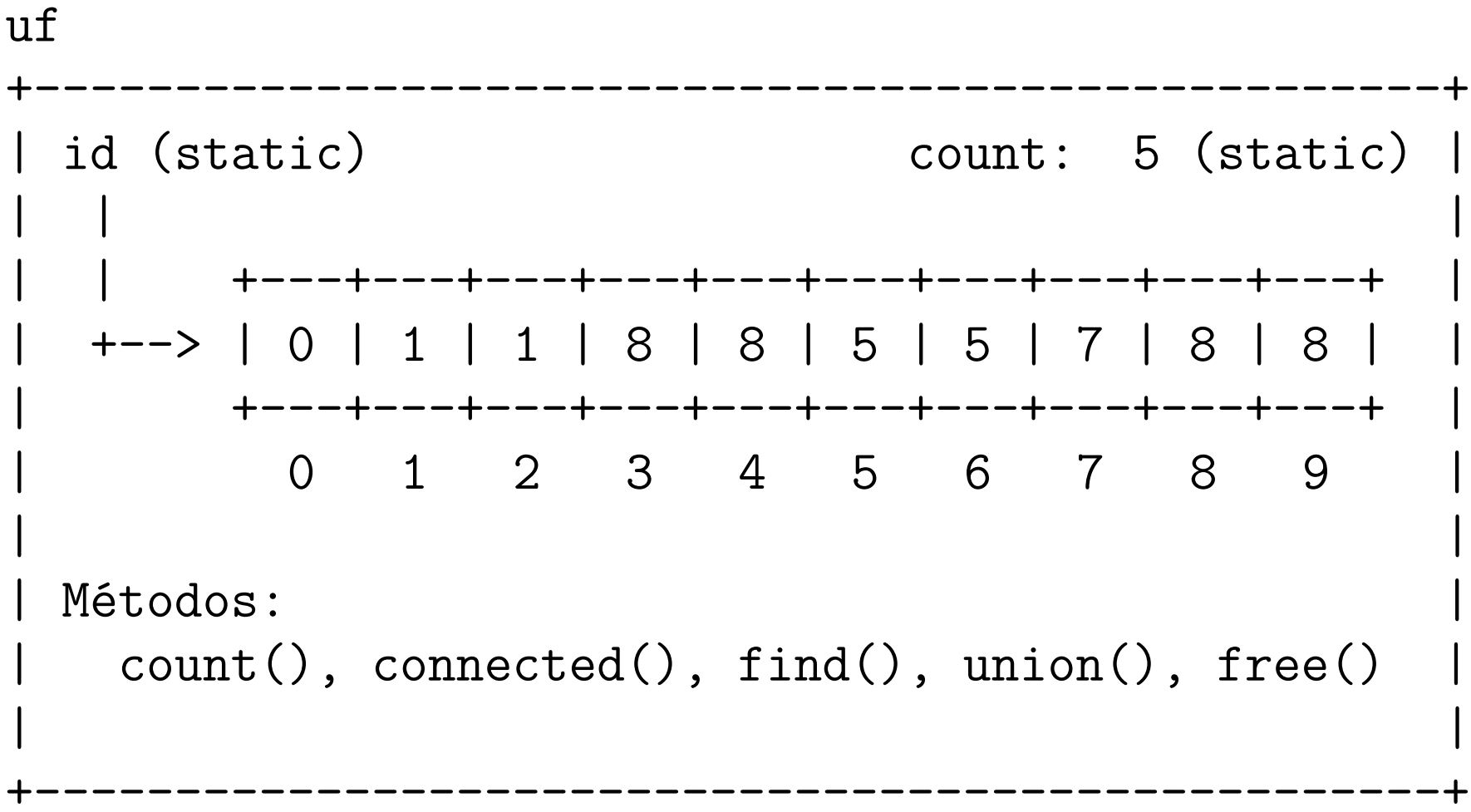
QuickFindUF

`ufUnion`(2, 1)



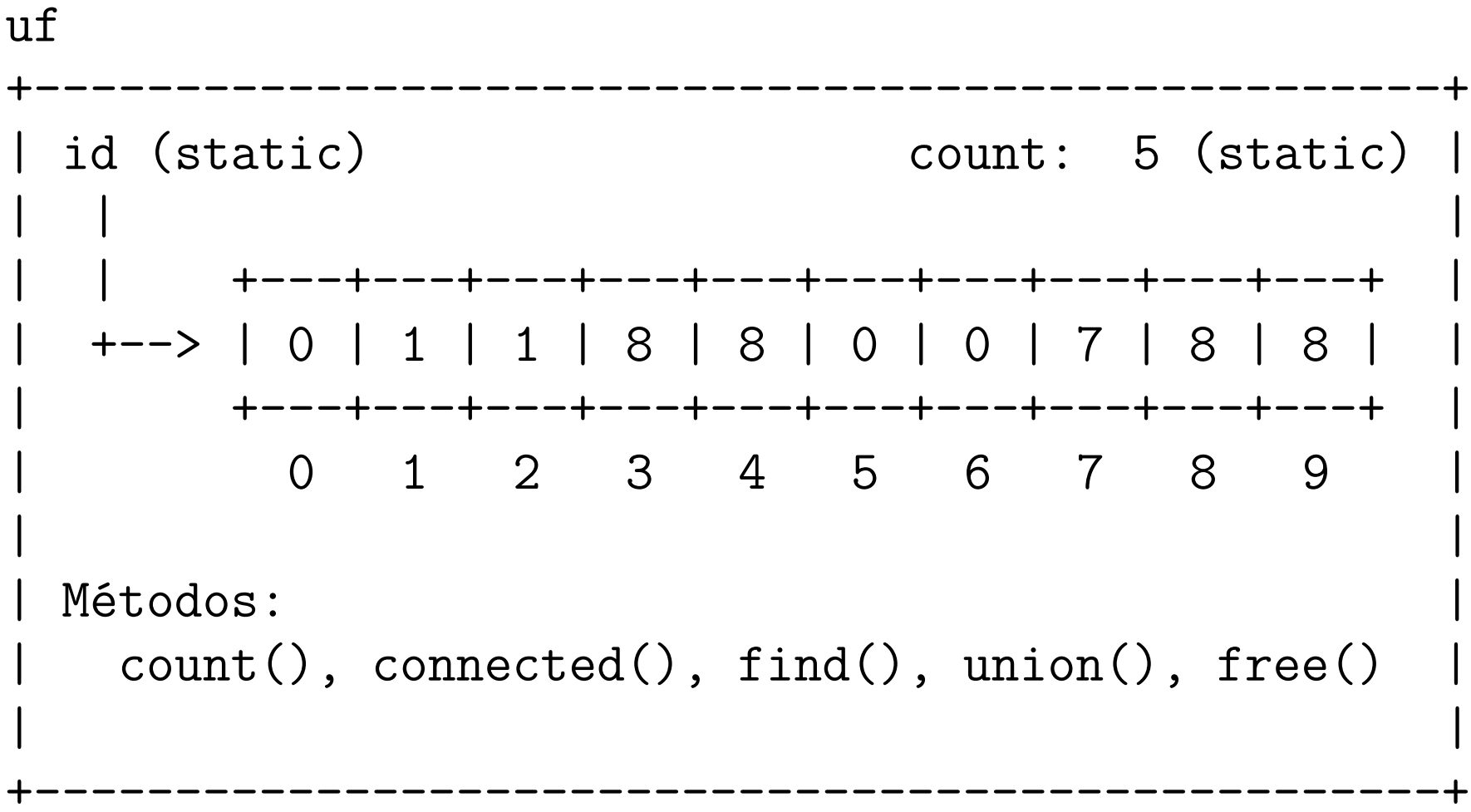
QuickFindUF

`ufUnion`(8, 9)



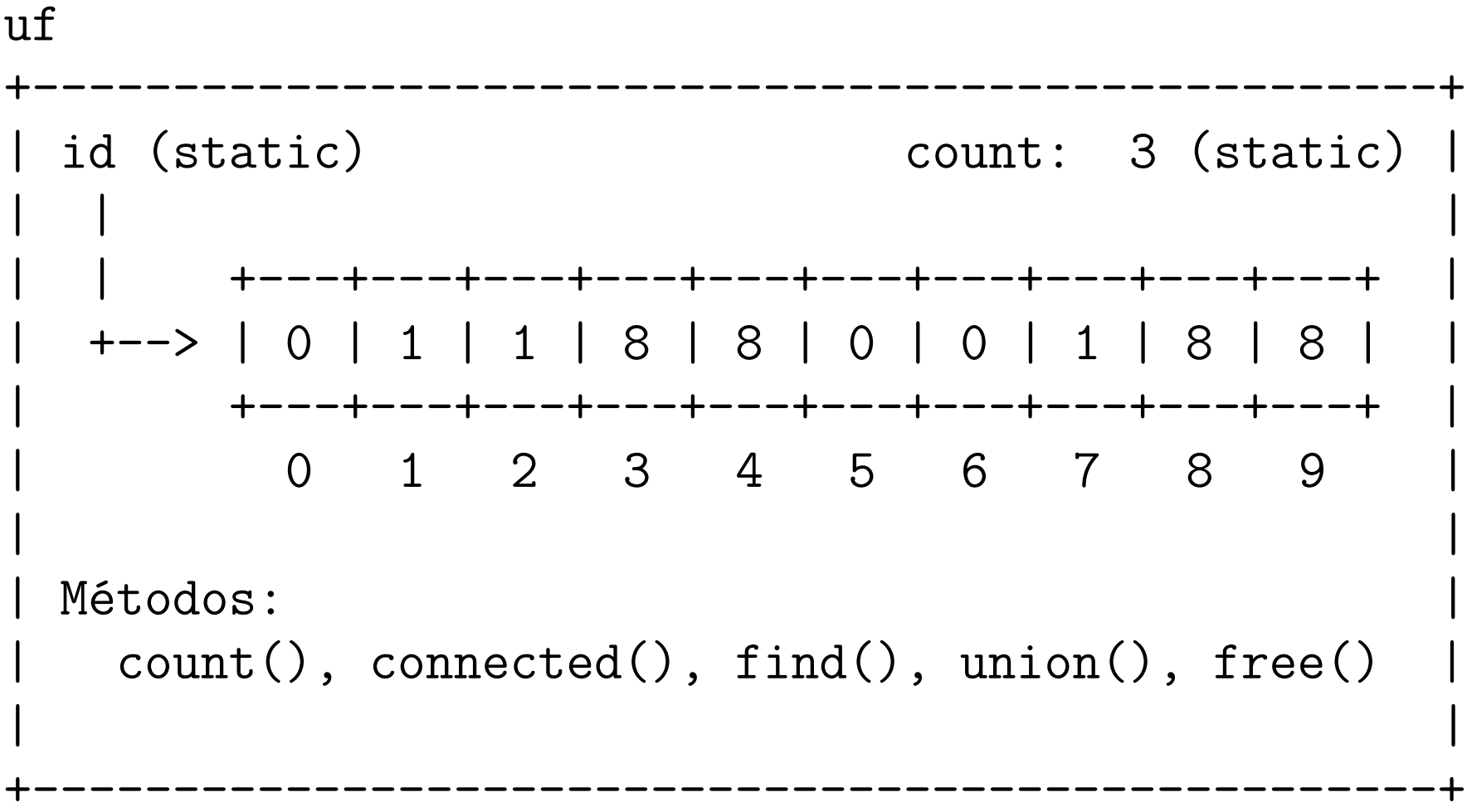
QuickFindUF

`ufUnion`(5, 0)



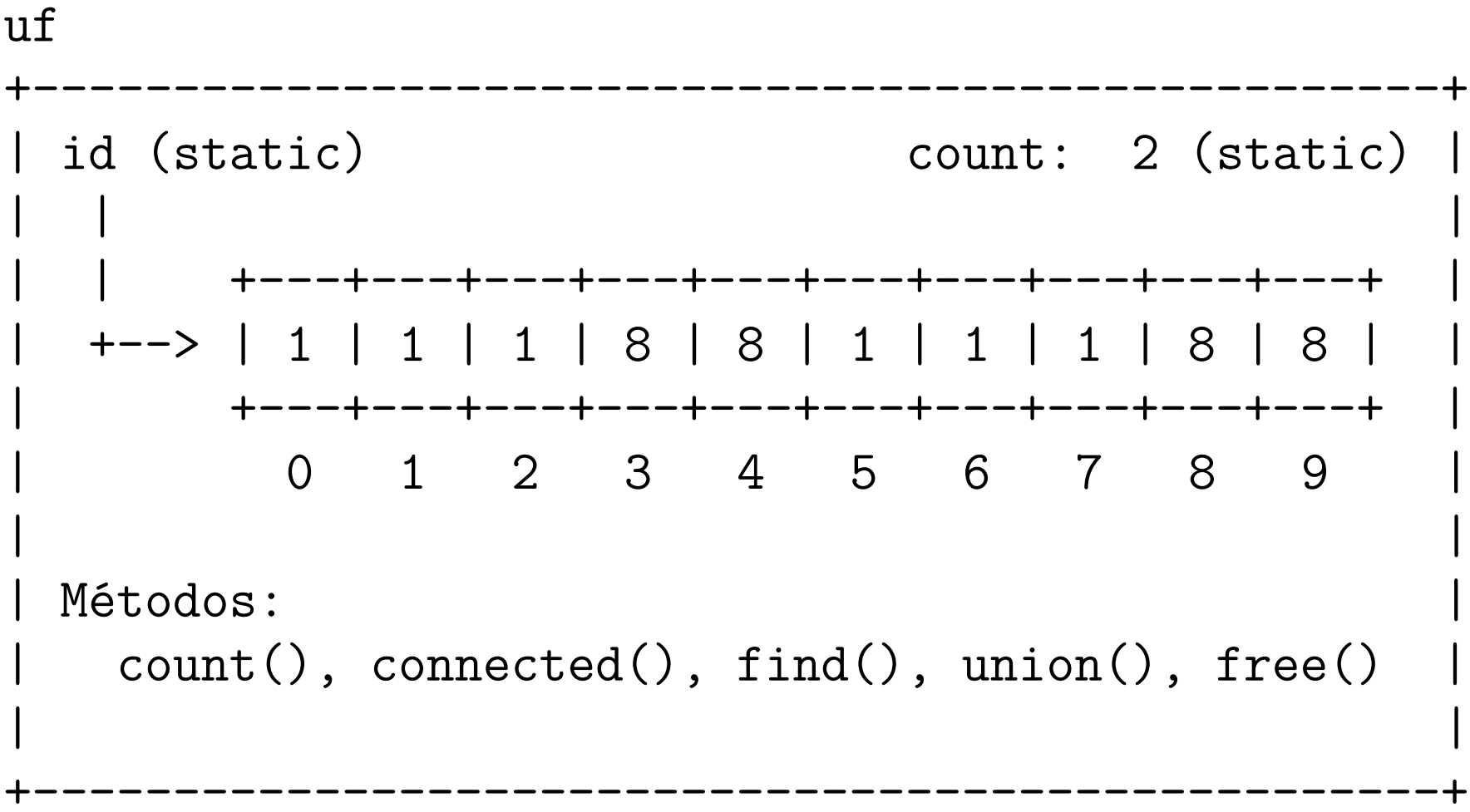
QuickFindUF

ufUnion(7, 2)



QuickFindUF

`ufUnion`(6, 1)



Arquivo QuickFindUF.c: esqueleto

```
#include "uf.h"

static int *id;
static int nUF;    /* no. elementos   */
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: 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;
}
```

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--;
}
```

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;
}
```

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 $O(m)$.

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)
:-(
```