Disciplina MCTA027-17 - Teoria dos Grafos

TAD Conjuntos disjuntos e Estrutura Union-Find

Profa. Carla Negri Lintzmayer

carla.negri@ufabc.edu.br

www.professor.ufabc.edu.br/∼carla.negri

Centro de Matemática, Computação e Cognição – Universidade Federal do ABC

Conjuntos Disjuntos

TAD Conjuntos Disjuntos

- Coleção dinâmica de elementos que estão particionados em grupos e que oferece operações de criação de um novo grupo (*make*), união de dois grupos existentes (*union*) e busca pelo grupo que contém um determinado elemento (*find*).

TAD Conjuntos Disjuntos

- Coleção dinâmica de elementos que estão particionados em grupos e que oferece operações de criação de um novo grupo (*make*), união de dois grupos existentes (*union*) e busca pelo grupo que contém um determinado elemento (*find*).
  - Estruturas de dados: Union-Find.

TAD Conjuntos Disjuntos

- Coleção dinâmica de elementos que estão particionados em grupos e que oferece operações de criação de um novo grupo (*make*), união de dois grupos existentes (*union*) e busca pelo grupo que contém um determinado elemento (*find*).
  - Estruturas de dados: Union-Find.
- Cada grupo tem um **representante**, que é um elemento contido no grupo.

Convenções

- Seja n o número de elementos armazenados na estrutura e m o número total de operações feitas.

Convenções

- Seja n o número de elementos armazenados na estrutura e m o número total de operações feitas.
  - Consideraremos que $m \geq n$, uma vez que n operações de *make* serão feitas sempre inicialmente.

Convenções

- Seja n o número de elementos armazenados na estrutura e m o número total de operações feitas.
  - Consideraremos que $m \geq n$, uma vez que n operações de *make* serão feitas sempre inicialmente.
- Os elementos serão números entre 1 e n.

Convenções

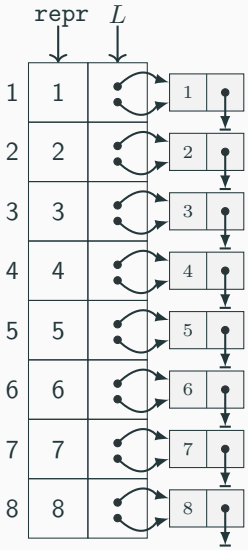
- Seja n o número de elementos armazenados na estrutura e m o número total de operações feitas.
  - Consideraremos que $m \geq n$, uma vez que n operações de *make* serão feitas sempre inicialmente.
- Os elementos serão números entre 1 e n.
  - Se não forem, podemos usar uma tabela de símbolos para fazer a conversão.

Listas ligadas

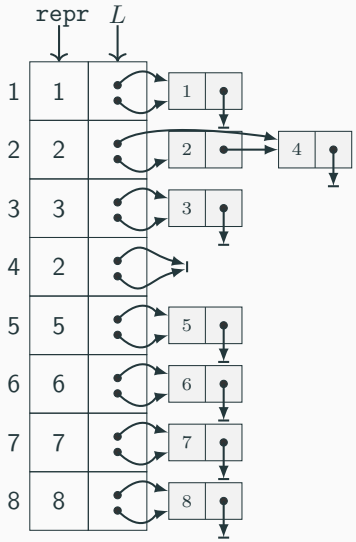
Implementação com listas ligadas

- Temos um vetor $repr$ de inteiros: $repr[i]$ guarda o elemento que é representante do grupo em que i está.
- Temos um vetor L de ponteiros para listas ligadas: se i é representante de um grupo, $L[i].head$ e $L[i].tail$ guardam um ponteiro para o início e o fim, respectivamente, de uma lista com os elementos do grupo; caso contrário, $L[i].head = L[i].tail = Null$.

Implementação com listas ligadas

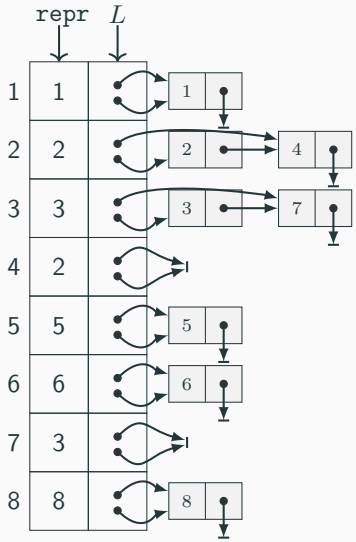


Implementação com listas ligadas



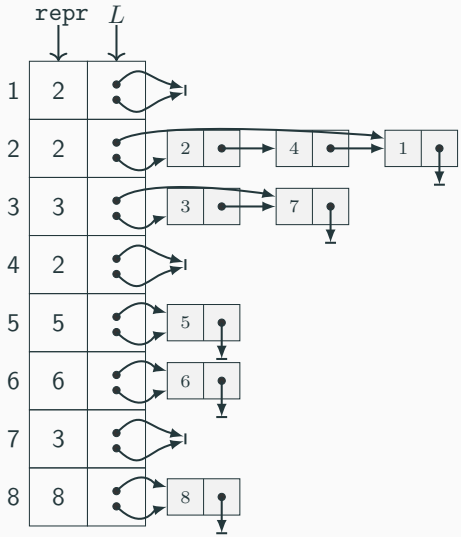
UNION(2, 4)

Implementação com listas ligadas



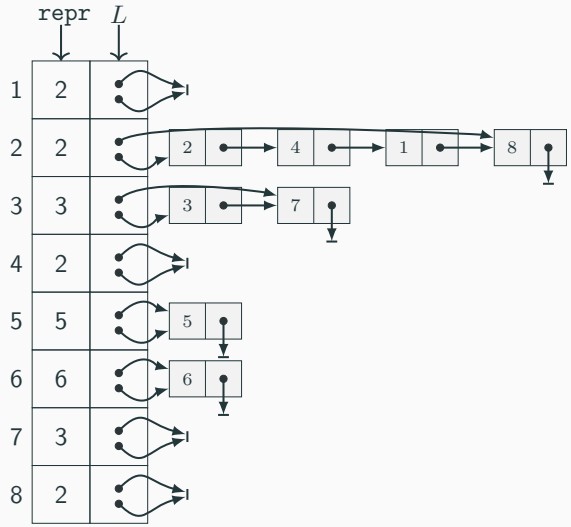
UNION(3, 7)

Implementação com listas ligadas



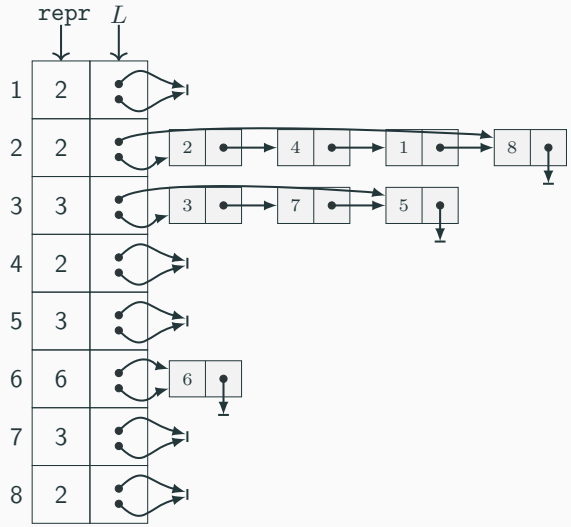
UNION(1, 4)

Implementação com listas ligadas



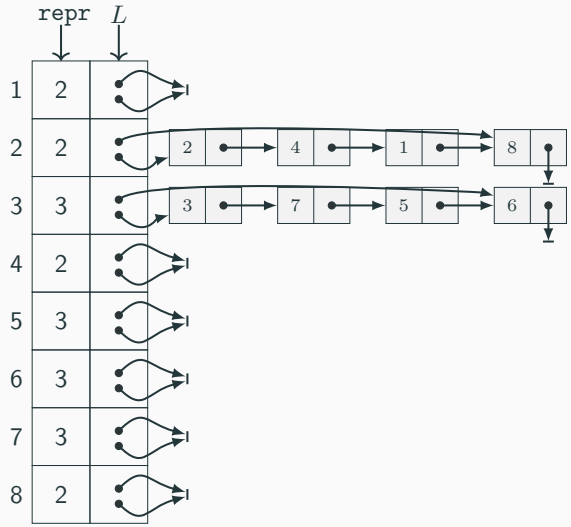
UNION(8, 2)

Implementação com listas ligadas



UNION(5, 3)

Implementação com listas ligadas



UNION(6, 7)

Implementação com listas ligadas

1: **Função** MakeSet(x)
2: $repr[x] \leftarrow x$
3: $n_x \leftarrow node(x)$
4: $L[x].head \leftarrow n_x$
5: $L[x].tail \leftarrow n_x$

1: **Função** Find(x)
2: **Devolve** $repr[x]$

1: **Função** Union(x, y)
2: $r_x \leftarrow$ Find(x)
3: $r_y \leftarrow$ Find(y)
4: **Para cada** v na lista $L[r_x]$ **faça**
5: $repr[v] \leftarrow r_y$
6: $L[r_y].tail.prox \leftarrow L[r_x].head$
7: $L[r_y].tail \leftarrow L[r_x].tail$
8: $L[r_x].head \leftarrow Null$
9: $L[r_x].tail \leftarrow Null$

Implementação com listas ligadas

- *Make* em $O(1)$.

Implementação com listas ligadas

- *Make* em $O(1)$.
- *Find* em $O(1)$.

Implementação com listas ligadas

- *Make* em $O(1)$.
- *Find* em $O(1)$.
- *Union* em $O(\ell)$, onde ℓ é o tamanho de um grupo ($\ell \leq n - 1$).

Implementação com listas ligadas

- *Make* em $O(1)$.
- *Find* em $O(1)$.
- *Union* em $O(\ell)$, onde ℓ é o tamanho de um grupo ($\ell \leq n-1$).

- A sequência de m operações pode levar tempo total $\Theta(n^2)$:

Implementação com listas ligadas

- *Make* em $O(1)$.
- *Find* em $O(1)$.
- *Union* em $O(\ell)$, onde ℓ é o tamanho de um grupo ($\ell \leq n - 1$).

- A sequência de m operações pode levar tempo total $\Theta(n^2)$:
  - Faça as n operações *make* iniciais serem seguidas das $n - 1$ operações *union* $\textsc{Union}(1, 2)$, $\textsc{Union}(2, 3)$, $\textsc{Union}(3, 4)$, $\ldots$, $\textsc{Union}(n - 1, n)$, de forma que a i-ésima operação gaste tempo $\Theta(i)$ por atualizar os representantes de i elementos.

Implementação com listas ligadas

- *Make* em $O(1)$.
- *Find* em $O(1)$.
- *Union* em $O(\ell)$, onde ℓ é o tamanho de um grupo ($\ell \leq n-1$).

- A sequência de m operações pode levar tempo total $\Theta(n^2)$:
  - Faça as n operações *make* iniciais serem seguidas das $n-1$ operações *union* $\textsc{Union}(1,\,2)$, $\textsc{Union}(2,\,3)$, $\textsc{Union}(3,\,4)$, $\ldots$, $\textsc{Union}(n-1,\,n)$, de forma que a i-ésima operação gaste tempo $\Theta(i)$ por atualizar os representantes de i elementos.
  - O tempo total é $\Theta(n) + \sum_{i=1}^{n-1} \Theta(i) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$.

Implementação com listas ligadas com union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.

Implementação com listas ligadas com union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

Implementação com listas ligadas com union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.
- *Union* continua sendo em $O(n)$.

Implementação com listas ligadas com union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.
- *Union* continua sendo em $O(n)$.

- As m operações agora levam tempo $O(m + n \log n)$:

Implementação com listas ligadas com union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.
- *Union* continua sendo em $O(n)$.

- As m operações agora levam tempo $O(m + n \log n)$:
  - Note que um objeto pode ter seu representante atualizado no máximo $\lg n$ vezes: ele começa em um grupo de tamanho 1; termina em um grupo de tamanho no máximo n; e toda vez que seu representante é atualizado, ele vai para um grupo no mínimo 2 vezes maior.

Implementação com listas ligadas com union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.
- *Union* continua sendo em $O(n)$.

- As m operações agora levam tempo $O(m + n \log n)$:
  - Note que um objeto pode ter seu representante atualizado no máximo $\lg n$ vezes: ele começa em um grupo de tamanho 1; termina em um grupo de tamanho no máximo n; e toda vez que seu representante é atualizado, ele vai para um grupo no mínimo 2 vezes maior.
  - Como são feitas no máximo m operações *make* e *find*, o tempo total é $O(m + n \lg n)$.

Implementação com listas ligadas com union by rank

```
1:  Função UNION(x, y)
2:      r_x ← FIND(x)
3:      r_y ← FIND(y)
4:      Se tam[r_x] < tam[r_y] então
5:          Para cada v na lista L[r_x] faça
6:              repr[v] ← r_y
7:          L[r_y].tail.prox ← L[r_x].head
8:          L[r_y].tail ← L[r_x].tail
9:          L[r_x].head ← Null
10:         L[r_x].tail ← Null
11:     Senão
12:         Para cada v na lista L[r_y] faça
13:             repr[v] ← r_x
14:         L[r_x].tail.prox ← L[r_y].head
15:         L[r_x].tail ← L[r_y].tail
16:         L[r_y].head ← Null
17:         L[r_y].tail ← Null
```

Florestas

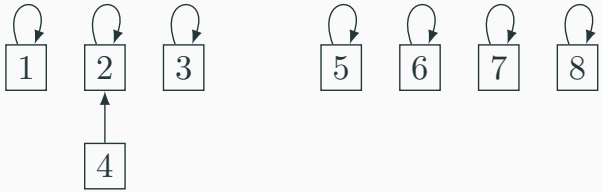
Implementação com florestas

- Cada grupo é uma árvore enraizada no representante.
  - Temos um vetor $pred$ de inteiros: $pred[i]$ guarda o elemento que é o predecessor de i na árvore do grupo em que i está, ou $pred[i] = i$, se i é o representante de um grupo.

Implementação com florestas

	1	2	3	4	5	6	7	8
pred	1	2	3	4	5	6	7	8

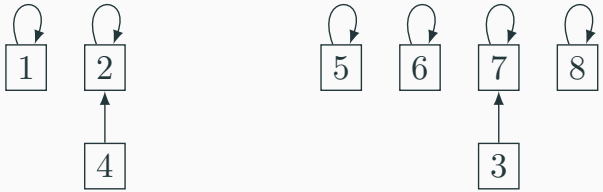
Implementação com florestas



	1	2	3	4	5	6	7	8
pred	1	2	3	2	5	6	7	8

UNION(2, 4)

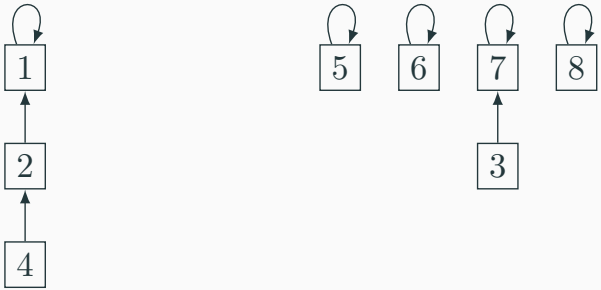
Implementação com florestas



	1	2	3	4	5	6	7	8
pred	1	2	7	2	5	6	7	8

UNION(3, 7)

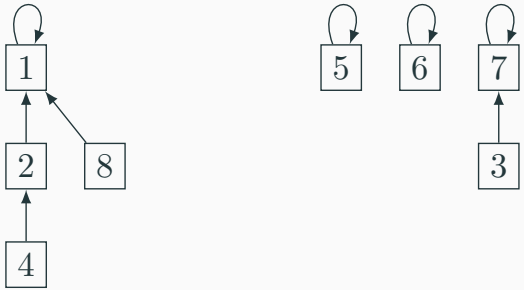
Implementação com florestas



	1	2	3	4	5	6	7	8
pred	1	1	7	2	5	6	7	8

UNION(1, 4)

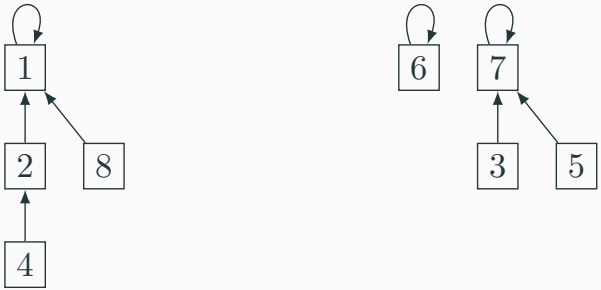
Implementação com florestas



	1	2	3	4	5	6	7	8
pred	1	1	7	2	5	6	7	1

UNION(8, 2)

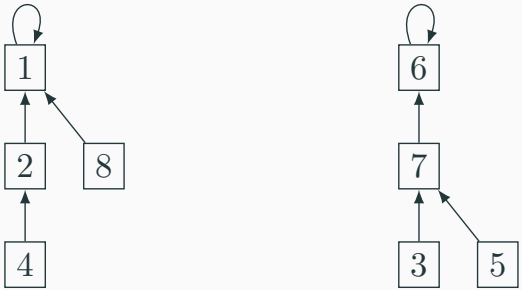
Implementação com florestas



	1	2	3	4	5	6	7	8
pred	1	1	7	2	7	6	7	1

UNION(5, 3)

Implementação com florestas



	1	2	3	4	5	6	7	8
pred	1	1	7	2	7	6	6	1

UNION(6, 7)

Implementação com florestas

1: **Função** MakeSet(x)
2: $pred[x] = x$

1: **Função** Find(x)
2: **Enquanto** $pred[x] \neq x$ **faça**
3: $x \leftarrow pred[x]$
4: **Devolve** $pred[x]$

1: **Função** Union(x, y)
2: $r_x \leftarrow$ Find(x)
3: $r_y \leftarrow$ Find(y)
4: $pred[r_y] \leftarrow r_x$

Implementação com florestas

- *Make* em $O(1)$.

Implementação com florestas

- *Make* em $O(1)$.
- *Union* em $O(1)$ tendo as raízes ou tempo do *find*.

Implementação com florestas

- *Make* em $O(1)$.
- *Union* em $O(1)$ tendo as raízes ou tempo do *find*.
- *Find* em $O(\ell)$, onde ℓ é a altura da árvore, pois vamos seguir pelos predecessores até a raiz.

Implementação com florestas

- *Make* em $O(1)$.
- *Union* em $O(1)$ tendo as raízes ou tempo do *find*.
- *Find* em $O(\ell)$, onde ℓ é a altura da árvore, pois vamos seguir pelos predecessores até a raiz.

- A sequência de m operações pode levar tempo total $\Theta(n^2)$:

Implementação com florestas

- *Make* em $O(1)$.
- *Union* em $O(1)$ tendo as raízes ou tempo do *find*.
- *Find* em $O(\ell)$, onde ℓ é a altura da árvore, pois vamos seguir pelos predecessores até a raiz.

- A sequência de m operações pode levar tempo total $\Theta(n^2)$:
  - Faça as n operações *make* iniciais serem seguidas das $n-1$ operações *union* $\textsc{Union}(1,\,2)$, $\textsc{Union}(2,\,3)$, $\textsc{Union}(3,\,4)$, $\ldots$, $\textsc{Union}(n-1,\,n)$, de forma que a árvore final tenha formato linear.

Implementação com florestas

- *Make* em $O(1)$.
- *Union* em $O(1)$ tendo as raízes ou tempo do *find*.
- *Find* em $O(\ell)$, onde ℓ é a altura da árvore, pois vamos seguir pelos predecessores até a raiz.

- A sequência de m operações pode levar tempo total $\Theta(n^2)$:
  - Faça as n operações *make* iniciais serem seguidas das $n-1$ operações *union* $\textsc{Union}(1,\,2)$, $\textsc{Union}(2,\,3)$, $\textsc{Union}(3,\,4)$, $\ldots$, $\textsc{Union}(n-1,\,n)$, de forma que a árvore final tenha formato linear.
  - Faça as n operações *find* $\textsc{Find}(1)$, $\textsc{Find}(2)$, $\ldots$, $\textsc{Find}(n)$. Note que a i-ésima operação gasta tempo $\Theta(i)$ por seguir i predecessores.

Implementação com florestas

- *Make* em $O(1)$.
- *Union* em $O(1)$ tendo as raízes ou tempo do *find*.
- *Find* em $O(\ell)$, onde ℓ é a altura da árvore, pois vamos seguir pelos predecessores até a raiz.

- A sequência de m operações pode levar tempo total $\Theta(n^2)$:
  - Faça as n operações *make* iniciais serem seguidas das $n-1$ operações *union* $\textsc{Union}(1,\,2)$, $\textsc{Union}(2,\,3)$, $\textsc{Union}(3,\,4)$, $\ldots$, $\textsc{Union}(n-1,\,n)$, de forma que a árvore final tenha formato linear.
  - Faça as n operações *find* $\textsc{Find}(1)$, $\textsc{Find}(2)$, $\ldots$, $\textsc{Find}(n)$. Note que a i-ésima operação gasta tempo $\Theta(i)$ por seguir i predecessores.
  - O tempo total é $\Theta(n) + \Theta(n) + \sum_{i=1}^{n-1} \Theta(i) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$.

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

- As m operações agora levam tempo $O(m \log n)$:

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

- As m operações agora levam tempo $O(m \log n)$:
  - É possível mostrar que em uma árvore de ordem k nessa floresta, o nível de um vértice é no máximo $\lg k$, por indução em k.

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

- As m operações agora levam tempo $O(m \log n)$:
  - É possível mostrar que em uma árvore de ordem k nessa floresta, o nível de um vértice é no máximo $\lg k$, por indução em k.
    - Uma árvore T que tem mais de um vértice certamente foi gerada por uma operação de *union* entre duas árvores de ordem menor T_1 e T_2;

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

- As m operações agora levam tempo $O(m \log n)$:
  - É possível mostrar que em uma árvore de ordem k nessa floresta, o nível de um vértice é no máximo $\lg k$, por indução em k.
    - Uma árvore T que tem mais de um vértice certamente foi gerada por uma operação de *union* entre duas árvores de ordem menor T_1 e T_2;
    - Por hipótese de indução, $nivel_{T_1}(u) \leq \lg |V(T_1)|$ para todo $u \in V(T_1)$ e $nivel_{T_2}(u) \leq \lg |V(T_2)|$ para todo $u \in V(T_2)$;

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

- As m operações agora levam tempo $O(m \log n)$:
  - É possível mostrar que em uma árvore de ordem k nessa floresta, o nível de um vértice é no máximo $\lg k$, por indução em k.
    - Uma árvore T que tem mais de um vértice certamente foi gerada por uma operação de *union* entre duas árvores de ordem menor T_1 e T_2;
    - Por hipótese de indução, $nivel_{T_1}(u) \leq \lg |V(T_1)|$ para todo $u \in V(T_1)$ e $nivel_{T_2}(u) \leq \lg |V(T_2)|$ para todo $u \in V(T_2)$;
    - S.p.g., $|V(T_1)| \leq |V(T_2)|$, de forma que a união fez a raiz de T_1 apontar para a raiz de T_2;

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

- As m operações agora levam tempo $O(m \log n)$:
  - É possível mostrar que em uma árvore de ordem k nessa floresta, o nível de um vértice é no máximo $\lg k$, por indução em k.
    - Uma árvore T que tem mais de um vértice certamente foi gerada por uma operação de *union* entre duas árvores de ordem menor T_1 e T_2;
    - Por hipótese de indução, $nivel_{T_1}(u) \leq \lg |V(T_1)|$ para todo $u \in V(T_1)$ e $nivel_{T_2}(u) \leq \lg |V(T_2)|$ para todo $u \in V(T_2)$;
    - S.p.g., $|V(T_1)| \leq |V(T_2)|$, de forma que a união fez a raiz de T_1 apontar para a raiz de T_2;
    - O resultado segue observando-se que $nivel_T(u) = nivel_{T_1}(u) + 1$ para todo $u \in V(T_1)$, $nivel_T(u) = nivel_{T_2}(u)$ para todo $u \in V(T_2)$ e mostrando-se que $nivel_T(u) \leq \lg |V(T)|$.

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

- As m operações agora levam tempo $O(m \log n)$:
  - É possível mostrar que em uma árvore de ordem k nessa floresta, o nível de um vértice é no máximo $\lg k$, por indução em k.
    - Uma árvore T que tem mais de um vértice certamente foi gerada por uma operação de *union* entre duas árvores de ordem menor T_1 e T_2;
    - Por hipótese de indução, $nivel_{T_1}(u) \leq \lg |V(T_1)|$ para todo $u \in V(T_1)$ e $nivel_{T_2}(u) \leq \lg |V(T_2)|$ para todo $u \in V(T_2)$;
    - S.p.g., $|V(T_1)| \leq |V(T_2)|$, de forma que a união fez a raiz de T_1 apontar para a raiz de T_2;
    - O resultado segue observando-se que $nivel_T(u) = nivel_{T_1}(u) + 1$ para todo $u \in V(T_1)$, $nivel_T(u) = nivel_{T_2}(u)$ para todo $u \in V(T_2)$ e mostrando-se que $nivel_T(u) \leq \lg |V(T)|$.
  - Isso implica que a altura de qualquer árvore é no máximo $\lg n$, o que por sua vez indica que qualquer operação *find* leva tempo $O(\lg n)$ para ser feita.

Implementação com florestas e union by rank

- Temos também um vetor tam de inteiros: se i é representante de um grupo, $tam[i]$ guarda o tamanho desse grupo; caso contrário, $tam[i] = 0$.
- Para *union*, fazemos o grupo menor ser representado pelo grupo maior.

- As m operações agora levam tempo $O(m \log n)$:
  - É possível mostrar que em uma árvore de ordem k nessa floresta, o nível de um vértice é no máximo $\lg k$, por indução em k.
    - Uma árvore T que tem mais de um vértice certamente foi gerada por uma operação de *union* entre duas árvores de ordem menor T_1 e T_2;
    - Por hipótese de indução, $nivel_{T_1}(u) \leq \lg |V(T_1)|$ para todo $u \in V(T_1)$ e $nivel_{T_2}(u) \leq \lg |V(T_2)|$ para todo $u \in V(T_2)$;
    - S.p.g., $|V(T_1)| \leq |V(T_2)|$, de forma que a união fez a raiz de T_1 apontar para a raiz de T_2;
    - O resultado segue observando-se que $nivel_T(u) = nivel_{T_1}(u) + 1$ para todo $u \in V(T_1)$, $nivel_T(u) = nivel_{T_2}(u)$ para todo $u \in V(T_2)$ e mostrando-se que $nivel_T(u) \leq \lg |V(T)|$.
  - Isso implica que a altura de qualquer árvore é no máximo $\lg n$, o que por sua vez indica que qualquer operação *find* leva tempo $O(\lg n)$ para ser feita.
  - Como são feitas no máximo m operações *find*, o tempo total é $O(m \lg n)$.

Implementação com florestas e path compression

- No *find*, fazemos cada nó apontar diretamente para a raiz da árvore que o contém.

Implementação com florestas e path compression

- No *find*, fazemos cada nó apontar diretamente para a raiz da árvore que o contém.

- As m operações agora levam tempo $\Theta(n + f + f \log_{2+f/n} n)$, onde f é o número de operações de procura apenas.

Implementação com florestas, union by rank e path compression

- Agora, as m operações levam tempo $O(m\,\alpha(n))$, onde $\alpha(n)$ é a função inversa de Ackermann.
  - $\alpha(n) \leq 4$ se $n << 10^{80}$.

Implementação com florestas, union by rank e path compression

```
1: Função MakeSet(x)
2:     pred[x] = x
3:     tam[x] = 1
```

```
1: Função Find(x)
2:     Se pred[x] ≠ x então
3:         pred[x] ← Find(pred[x])
4:     Devolve pred[x]
```

```
1: Função Union(x, y)
2:     r_x ← Find(x)
3:     r_y ← Find(y)
4:     Se tam[r_x] < tam[r_y] então
5:         pred[r_x] ← r_y
6:         tam[r_y] ← tam[r_y] + tam[r_x]
7:         tam[r_x] ← 0
8:     Senão
9:         pred[r_y] ← r_x
10:        tam[r_x] ← tam[r_x] + tam[r_y]
11:        tam[r_y] ← 0
```

Spoiler

Código

```
1  typedef struct ufind* UF;
2
3  struct ufind {
4      int* pred;
5      int* size;
6  };
```

Essa não tem segredo nenhum, não é?