```
-------------------------------------------------------------------------------
 Estrutura de Floresta para Conjuntos Disjuntos (Cormen, Cap. 21 até seção 3)
-------------------------------------------------------------------------------
```

Os seguintes algoritmos mantêm a estrutura:

```
  ===============================================================================
  criar_conj(x)  // Pré-condição: "x" não pode já participar de uma árvore.
                 /* Motivo: se "x" já participasse de uma árvore de raiz "r",
                    então, quando "x" se tornasse raiz de sua própria árvore,
                    "x" sairia da árvore de "r", a qual poderia passar a não ter
                    pelo menos 2^h[r] nós. */
  -------------------------------------------------------------------------------
  1. pai[x] := x
  2. h[x] := 0  /* Limite superior para a altura da subárvore enraizada em "x",
                   computando-se tal altura como o maior número de arestas num
                   caminho de "x" a uma folha dessa subárvore. */
                // Toda árvore de raiz "r" (pai[r] = r) tem 2^h[r] nós ou mais.
  ===============================================================================

  ===============================================================================
  enc_repr(x)  /* medida de terminação: "h[r] - h[x]", sendo "r" a raiz da
                  árvore de "x". */
  -------------------------------------------------------------------------------
  1. SE pai[x] != x
  2. | pai[x] := enc_repr( pai[x] )
  3. RETORNE pai[x].
  ===============================================================================

  ===============================================================================
  unir_conj (x,y)  // Pré-condição: "x" e "y" pertencem a árvores distintas.
  -------------------------------------------------------------------------------
  1. a := enc_repr(x) , b := enc_repr(y)
  2. SE h[a] > h[b]
  3. | pai[b] := a
  4. SENÃO
  5. | pai[a] := b
  6. | SE h[a] = h[b]
  7. | | ++h[b]
  ===============================================================================
```

Dados os algoritmos acima, podemos provar:

1. LEMA: numa sequência de chamadas às 3 funções acima, enquanto "criar_conj"
   ---- tiver sido chamada no máximo "n" vezes, então, ∀x, h[x] <= ⌊lg n⌋.

```
===============================================================================
```
PROVA:

Nós provaremos primeiramente que o seguinte fato é mantido sempre verdadeiro numa sequência de chamadas às 3 funções acima -- abaixo, por uma "raiz r" nos referimos a um nó tal que "pai[r] = r":

  Toda árvore cuja raiz é um nó "r" tem pelo menos 2^h[r] nós. (INV)

De fato:

  1. A função "criar_conj" cria uma árvore cujo único nó é "x", ou seja, cujo
     número de nós é

       1 = 2^0 = 2^h[x],

     ou seja, INV vale com relação à árvore de "x". Além disso, como, pela

pré-condição de criar_conj, "x" não pertencia a nenhuma árvore anteriormente, então INV não deixa de valer para nenhuma outra árvore da estrutura, o que conclui o argumento.

2. A função "enc_repr" não altera o campo "h" de nenhum nó e nem altera o número de nós de nenhuma árvore, e portanto mantém INV válido.

3. A função "unir_conj" faz a união de árvores distintas (devido à pré-cond.), as quais são enraizadas em nós "a" e "b" e inicialmente possuem pelo menos 2^h[a] e 2^h[b] nós. Se a linha 7 não é executada, então INV obviamente permanece verdadeiro, pois continua a valer para as árvores que não possuem nem "a" nem "b" e também vale para a nova árvore enraizada em "a" ou "b", pois o campo "h" dessa raiz ("a" ou "b") não mudou e o número de nós da árvore nela enraizada "aumentou" (isto é, é maior que o número de nós da árvore que inicialmente tinha essa raiz). Se, porém, a linha 7 é executada, então, para um certo "k", inicialmente temos "h[a] = h[b] = k", e ao final temos "h[b] = k+1". Além disso, ao final, a árvore enraizada em "b" terá pelo menos

   2^h[a] + 2^h[b] = 2^k + 2^k = 2^(k+1)

   nós, ou seja, INV também vale para a nova árvore, concluindo o argumento.

A partir de INV, o enunciado pode ser provado como segue. Suponha, por absurdo, que, ao final de uma certa sequência de chamadas às 3 funções acima, das quais no máximo "n" foram chamadas a "criar_conj", um nó "x" é tal que "h[x] > ⌊lg n⌋". Como h[x] é um valor inteiro, então segue que "h[x] > lg(n)" (observe que isso vale tanto quando lg(n) é inteiro quanto no caso contrário). Seja então "r" a raiz da árvore a que "x" pertence no momento em questão. É fácil verificar que as 3 funções acima garantem que, se um nó "p" é ascendente de um nó "q", então "h[p] >= h[q]", e disso segue que "h[r] >= h[x] > lg(n)". Por INV, portanto, a árvore enraizada em "r" tem pelo menos

  2^h[r] > 2^(lg n) = n

nós, ou seja, pelo menos "n+1" nós, um absurdo, pois a função "criar_conj" foi chamada no máximo "n" vezes até então. Logo, a suposição de que havia um nó "x" tal que "h[x] > ⌊lg n⌋" é falsa, o que prova o enunciado.
==========

2. TEOREMA: qualquer sequência válida de "m" chamadas às 3 funções acima,
   ------- das quais "n" sejam a "criar_conj", executa em tempo O(m*lg n).

==========
PROVA:

Observe primeiramente que as 3 funções em questão garantem que, para qualquer nó "x", h[x] é sempre um limite superior para a altura da subárvore enraizada em "x", computando-se tal altura como o maior número de arestas num caminho de "x" a uma folha dessa subárvore.

Considere agora a execução de uma chamada "enc_repr(x)" qualquer. É evidente que o número de chamadas recursivas realizadas a partir dessa chamada é limitado pela altura da raiz "r" da árvore do nó "x". Logo, pela observação acima, esse número de chamadas recursivas é limitado por h[r]. Além disso, pelo lema 1, se o número de chamadas a "criar_conj" é "n", então "h[r] <= ⌊lg n⌋", e portanto a chamada "enc_repr(x)" realiza no máximo ⌊lg n⌋ chamadas recursivas. Por fim, como cada chamada a "enc_repr" leva, por si só (isto é, excluindo-se o tempo das chamadas recursivas), tempo constante, então a chamada "enc_repr(x)" executa em tempo O(lg n).

Pela análise anterior, portanto, numa sequência de "m" chamadas às funções "criar_conj", "enc_repr" e "unir_conj", das quais "n" sejam a "criar_conj",

as chamadas a "enc_repr" levam tempo O(lg n) cada uma. Além disso, o mesmo vale para cada chamada a "unir_conj", pois essa função apenas realiza duas chamadas a "enc_repr" e depois executa um pequeno código cujo tempo de execução pode ser limitado superiormente por uma constante. Finalmente, como cada chamada a "criar_conj" leva tempo constante, então a sequência das "m" chamadas em questão executa em tempo O(m*lg n), CQD.
==========