DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estruturas de Dados

Estruturas para Conjuntos Disjuntos

Professores: Anisio Lacerda
Wagner Meira Jr.

Distribuição de notícias

Em uma rede social, existem *n* usuários e *m* grupos. Vamos analisar o processo de distribuição de notícias.

Inicialmente, algum usuário *x* recebe uma notícia.

Esse usuário encaminha essa notícia para seus amigos

dois usuários são amigos se existe pelo menos um grupo em comum

Amigos continuam enviando notícias para seus amigos.

O processo termina quando não existir nenhum par de amigos tal que um sabe a notícia e o outro não.

Para cada usuário *x* determine o número de usuários que saberão a notícia se, inicialmente, *x* começou a distribuí-la.

Motivação

- Representar conjuntos disjuntos
  - Seja S = {S1 , S2 , . . . , Sn } uma coleção de conjuntos disjuntos, ou seja, Si ∩ Sj = ∅
  - Representam classes de equivalência, conectividade etc.
- Aplicações normalmente modificam esses conjuntos.
- O objetivo é encontrar uma representação que nos permite operar de forma eficiente sobre esses conjuntos.

Formulação

- Nossa entrada consiste de um conjunto universo (ou *ground set*) de elementos $U = \{x_1, x_2, \ldots, x_n\}$.
- Uma coleção $S = \{S_1, S_2, \ldots, S_n\}$ de conjuntos tais que para todo i $S_i \subseteq U$ e para todo $i \neq j$ vale $S_i \cap S_j = \varnothing$.

Operações

- **Make:** Dado um elemento de U, construir um conjunto contendo apenas U.
- **Union:** Dado dois conjuntos S_i e S_j, queremos unir ambos conjuntos.
- **Find:** Dado um elemento x_k, retornar a qual subconjunto ele pertence.

Implementações

- Chamamos esse tipo de estrutura de *Disjoint Set Union* (DSU) ou *Union-Find*.
- Vamos ver duas formas de se implementar DSU:
  - Utilizando vetores.
  - Utilizando árvores.

Implementações

Observação importante!

As implementações nesta seção são apenas ilustrativas. Não necessariamente são as mais otimizadas, o objetivo é apenas exemplificar o TAD.

DSU com vetores

- É uma forma bem simples de se implementar a ideia de um DSU.
- Se temos *n* elementos como entrada vamos criar um vetor com *n* elementos.
- Os conjuntos serão implementados de forma implícita, através do que chamamos de **representantes**.
- Dois elementos estão em um mesmo conjunto se e somente se eles possuem representantes iguais.
- Inicialmente vamos utilizar a função *make* para criar um conjunto para cada elemento, então cada elemento será o representante de seu próprio conjunto.

DSU com vetores

```
class DSU{
    public:
        DSU(int quantidade_conjuntos);
        ~DSU();
        void Make(int x);
        int Find(int x);
        void Union(int x, int y);
    private:
        int tamanho;
        int* conjuntos;
};

DSU::DSU(int n){
    conjuntos = new int[n];
    tamanho = n;
}
DSU::~DSU(){
    delete[] conjuntos;
}
```

DSU com vetores - Make

- Nossa função make simplesmente faz com que o elemento se torne seu próprio representante, o tornando disjunto dos outros conjuntos.

```
void DSU::Make(int x){
    conjuntos[x] = x;
}
```

Complexidade?

DSU com vetores - Make

- Nossa função make simplesmente faz com que o elemento se torne seu próprio representante, o tornando disjunto dos outros conjuntos.

```
void DSU::Make(int x){
    conjuntos[x] = x;
}
```

Complexidade?

O(1)!

DSU com vetores - Find

- Como identificamos os conjuntos da coleção pelo representante, a função *find* apenas retorna o representante do conjunto desejado

```
int DSU::Find(int x){
    return conjuntos[x];
}
```

Complexidade?

DSU com vetores - Find

- Como identificamos os conjuntos da coleção pelo representante, a função *find* apenas retorna o representante do conjunto desejado

```
int DSU::Find(int x){
    return conjuntos[x];
}
```

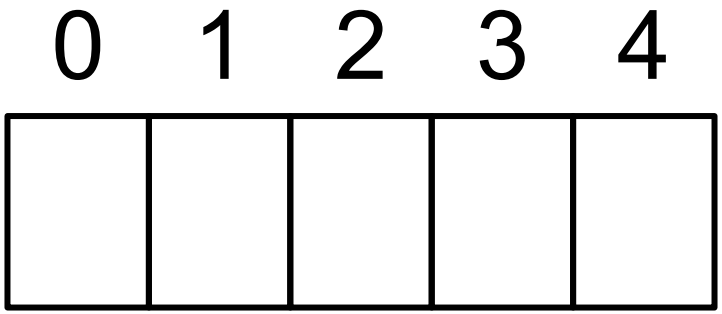
Complexidade?

O(1)!

DSU com vetores - Exemplo

Seja {0,1,2,3,4} os elementos da nossa entrada.

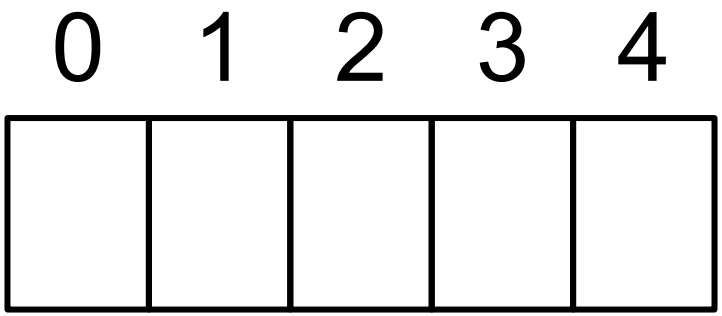
Nosso primeiro passo é construir o vetor:



DSU com vetores - Exemplo

Seja {0,1,2,3,4} os elementos da nossa entrada.

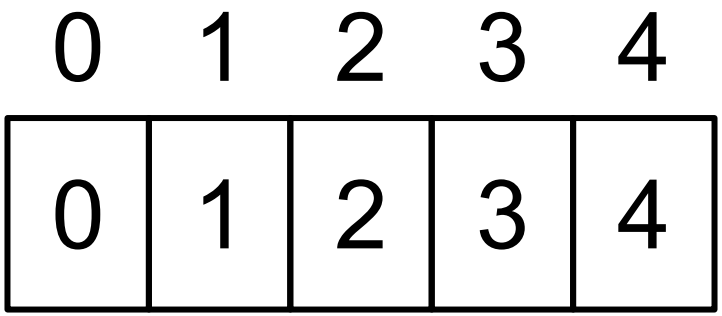
Agora vamos utilizar a função *make* para criar um conjunto para cada elemento, formando a coleção S={ {0}, {1}, {2}, {3}, {4} }.



DSU com vetores - Exemplo

Seja {0,1,2,3,4} os elementos da nossa entrada.

Na nossa representação isso significa que cada elemento terá um representante diferente.

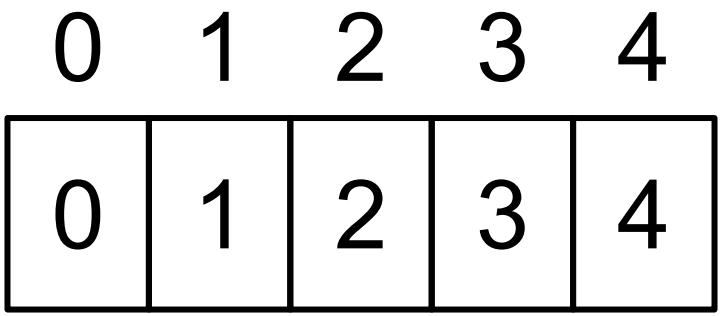


DSU com vetores - Exemplo

Vamos unir agora os conjuntos {0} e {2}.

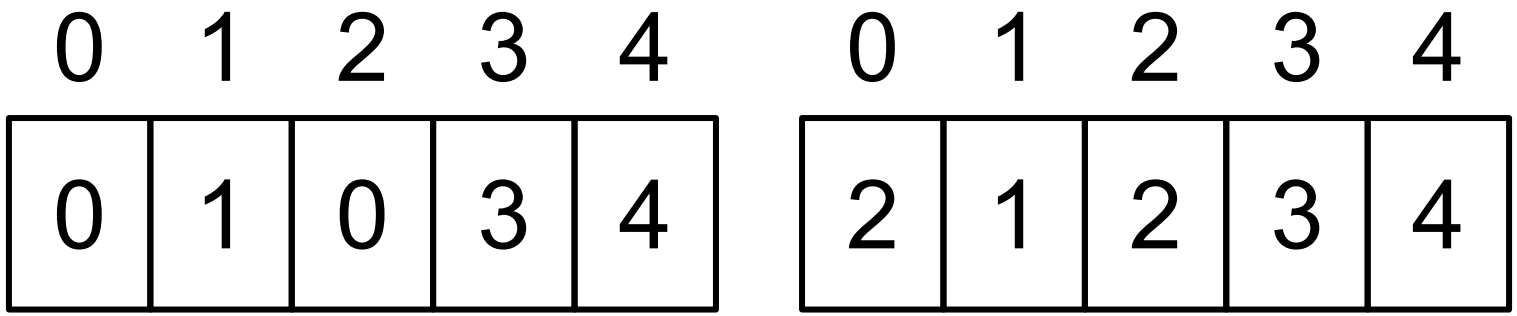
Após a execução da união nossa coleção será: S={ {0, 2}, {1}, {3}, {4} }.

Como devemos alterar nossa representação?



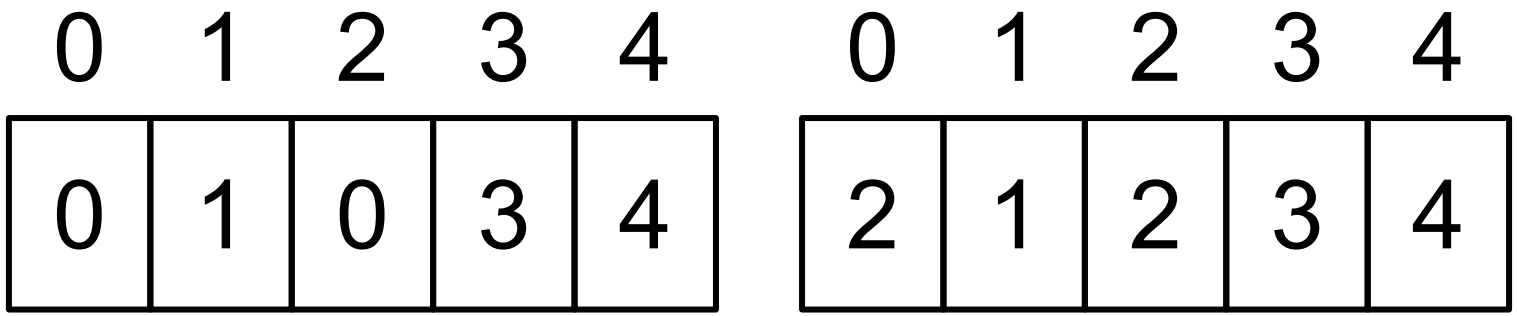
DSU com vetores - Exemplo

Devemos fazer com que 0 e 2 possuam os mesmos representantes. Logo temos duas saídas possíveis:



DSU com vetores - Exemplo

Não faz diferença qual dos dois caminhos tomar, no entanto precisamos garantir que **todos** os elementos que possuíam representantes 0 ou 2 agora tenham o mesmo representante.

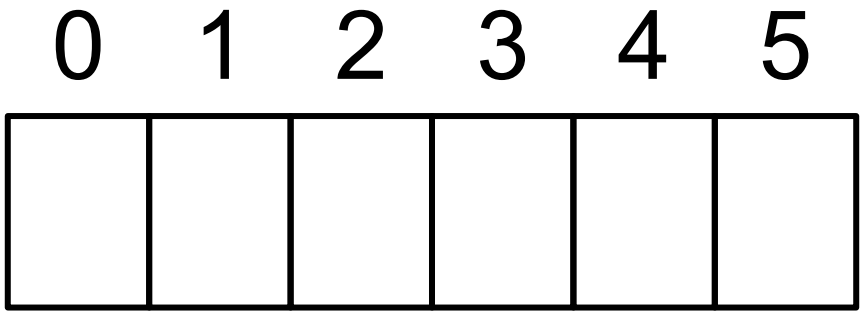


DSU com vetores - Exemplo

Para ficar mais claro, vamos considerar outro exemplo.

Seja S={ {0, 3}, {1, 4}, {2, 5} }.

Como devemos preencher nossa representação?

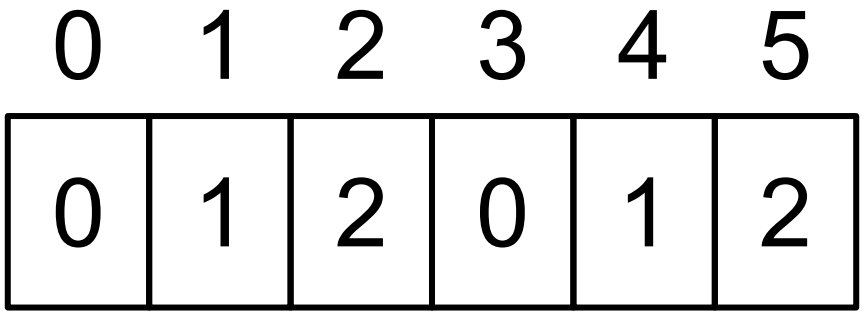


DSU com vetores - Exemplo

Para ficar mais claro, vamos considerar outro exemplo.

Seja S={ {0, 3}, {1, 4}, {2, 5} }.

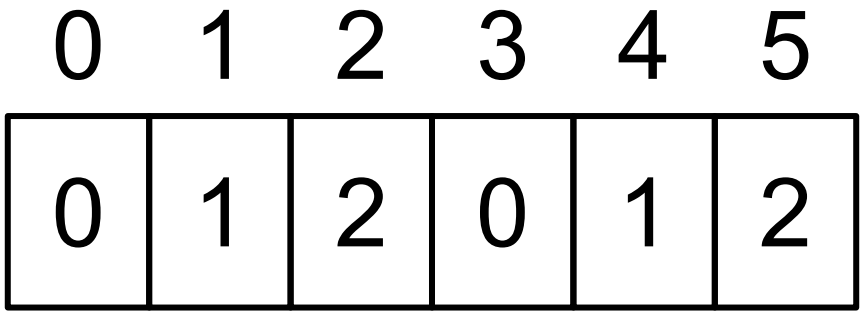
Como devemos preencher nossa representação?



DSU com vetores - Exemplo

Agora desejamos unir {0, 3} e {2, 5}.

Como modificamos nossa representação?

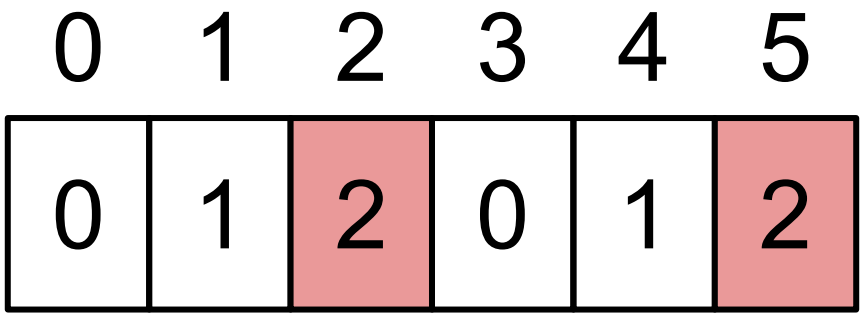


DSU com vetores - Exemplo

Agora desejamos unir {0, 3} e {2, 5}.

Como modificamos nossa representação?

Devemos modificar o representante de **todos** os elementos do conjunto {2, 5}.

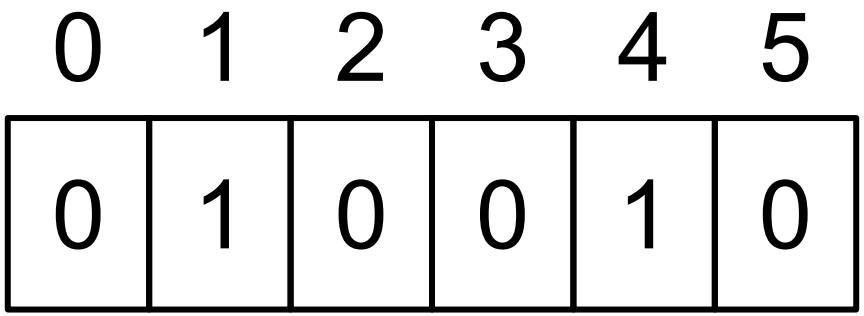


DSU com vetores - Exemplo

Agora desejamos unir {0, 3} e {2, 5}.

Como modificamos nossa representação?

Devemos modificar o representante de **todos** os elementos do conjunto {2, 5}.



DSU com vetores - Union

```
void DSU::Union(int x, int y){
    x = Find(x);
    y = Find(y);
    if(x != y){
        for(int i = 0; i < tamanho; i++){
            if(conjuntos[i] == y){
                conjuntos[i] = x;
            }
        }
    }
}
```

Complexidade?

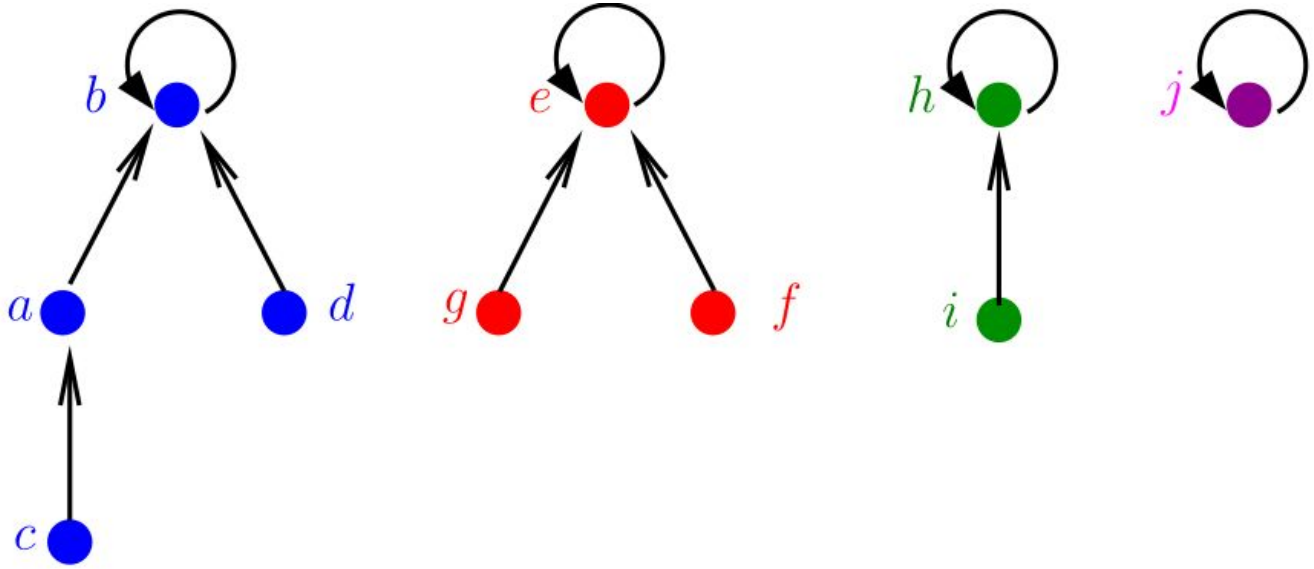
O(n)

DSU com árvores

- Cada conjunto é uma árvore.
- O representante de cada conjunto é sua raiz.
- Todos os elementos tem um “ponteiro” **pai**.
- Na raiz, o “ponteiro” **pai** aponta para ele mesmo.

DSU com árvores

- Cada conjunto é uma árvore.
- O representante de cada conjunto é sua raiz.
- Todos os elementos tem um “ponteiro” **pai**.
- Na raiz, o “ponteiro” **pai** aponta para ele mesmo.



DSU com árvores

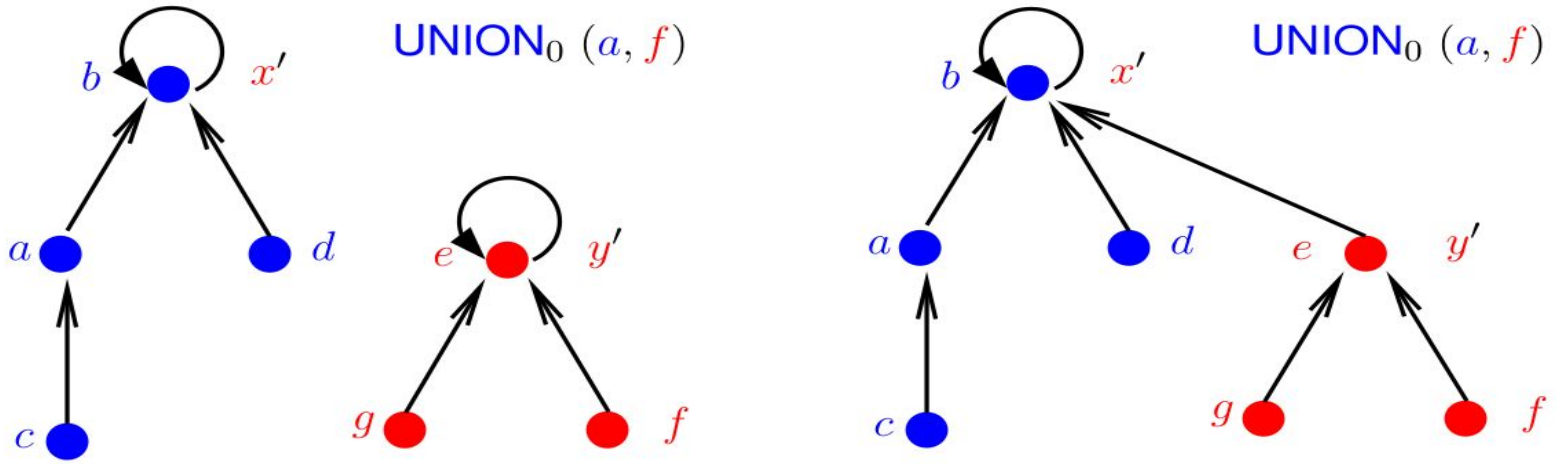
- Assim como já fizemos em outras estruturas de dados, nós iremos simular nossa abstração de árvore em um vetor.
- Nosso vetor será a representação dos ponteiros pai. Então na i-ésima posição deste vetor estará gravado qual elemento é o pai do elemento x_i.
- A função *find* agora deve navegar pela árvore até encontrar a raiz (o nó cujo pai é ele próprio).

DSU com árvores

- Para unir dois conjuntos basta conectar as duas árvores
- Fazemos o ponteiro de uma das raízes apontar para a outra.

DSU com árvores

- Para unir dois conjuntos basta conectar as duas árvores
- Fazemos o ponteiro de uma das raízes apontar para a outra.



DSU com árvores

```
class DSU{
    public:
        DSU(int quantidade_conjuntos);
        ~DSU();
        void Make(int x);
        int Find(int x);
        void Union(int x, int y);
    private:
        int tamanho;
        int* conjuntos;
};

DSU::DSU(int n){
    conjuntos = new int[n];
    tamanho = n;
}
DSU::~DSU(){
    delete[] conjuntos;
}
```

DSU com árvores - Operações

```
void DSU::Make(int x){
    conjuntos[x] = x;
}
int DSU::Find(int x){
   while(conjuntos[x] != x){
       x = conjuntos[x];
   }
   return x;
}
void DSU::Union(int x, int y){
   int raiz_x = Find(x);
   int raiz_y = Find(y);
   if(raiz_x != raiz_y){
       conjuntos[raiz_y] = raiz_x;
   }
}
```

Quais são os problemas desta implementação?

DSU com árvores

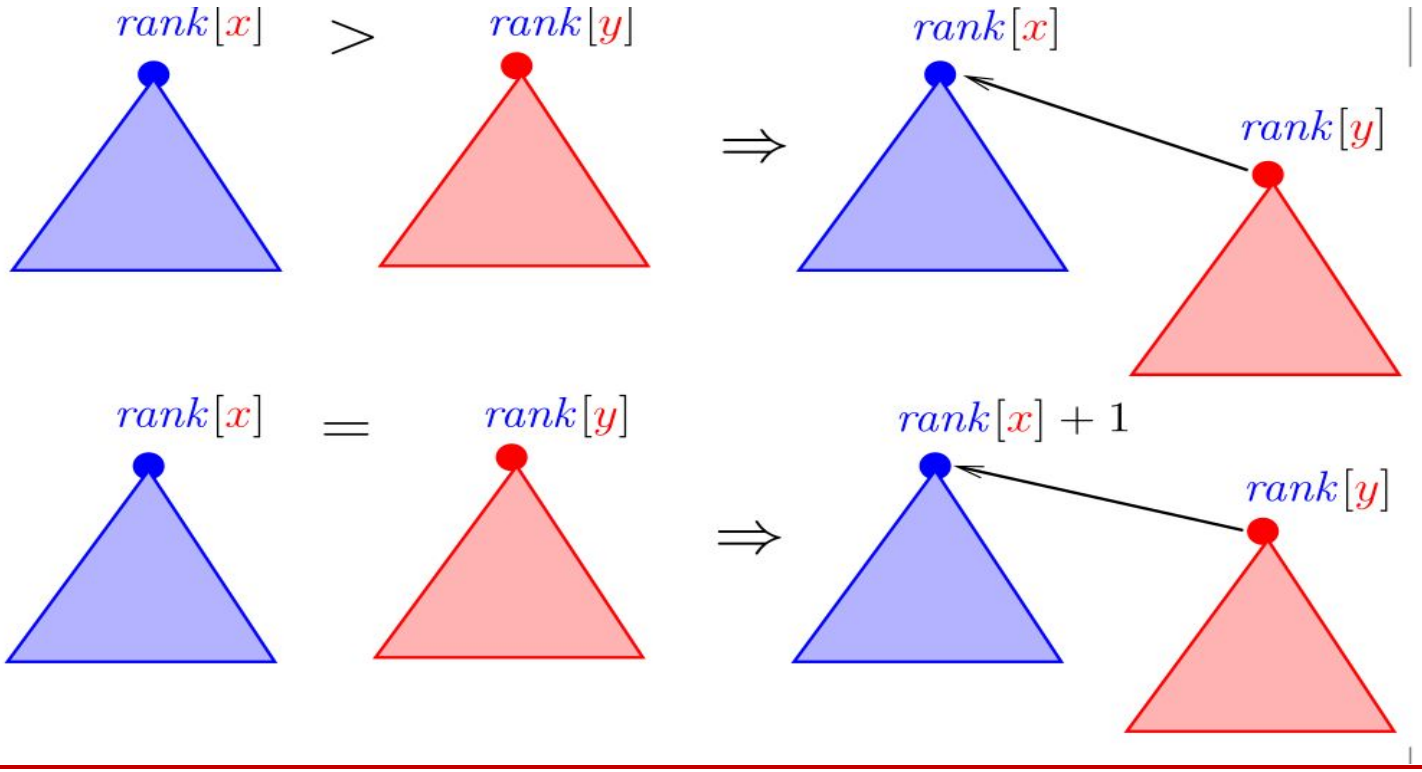
- A junção das árvores é feita sem critério.
- Isso pode fazer com que a árvore resultante seja degenerada (p.ex., lista linear).
- Gostaríamos que a árvore fosse balanceada, para podermos garantir complexidade logarítimica para o *find*.
- Nosso objetivo agora é encontrar artifícios para otimizar a junção das árvores.
  - Para isso vamos utilizar a altura da árvore!
  - Cada elemento agora possuirá um campo **rank**, que é a sua altura na árvore.

DSU com árvores - Rank

- Qual das raízes é a nova raiz da árvores? A de maior rank
- Atualizamos o rank apenas se forem iguais.

DSU com árvores - Rank

- Qual das raízes é a nova raiz da árvores? A de maior rank
- Atualizamos o rank apenas se forem iguais.



DSU com árvores - Union por rank

```
class DSU{
    public:
        DSU(int quantidade_conjuntos);
        ~DSU();
        void Make(int x);
        int Find(int x);
        void Union(int x, int y);

    private:
        int tamanho;
        int* conjuntos;
        int* rank;
};
```

DSU com árvores - Union por rank

```
DSU::DSU(int n){
    conjuntos = new int[n];
    rank = new int[n];
    tamanho = n;
}
DSU::~DSU(){
    delete[] conjuntos;
    delete[] rank;
}
void DSU::Make(int x){
    conjuntos[x] = x;
    rank[x] = 0;
}
int DSU::Find(int x){
   while(conjuntos[x] != x){
        x = conjuntos[x];
    }
    return x;
}
```

DSU com árvores - Union por rank

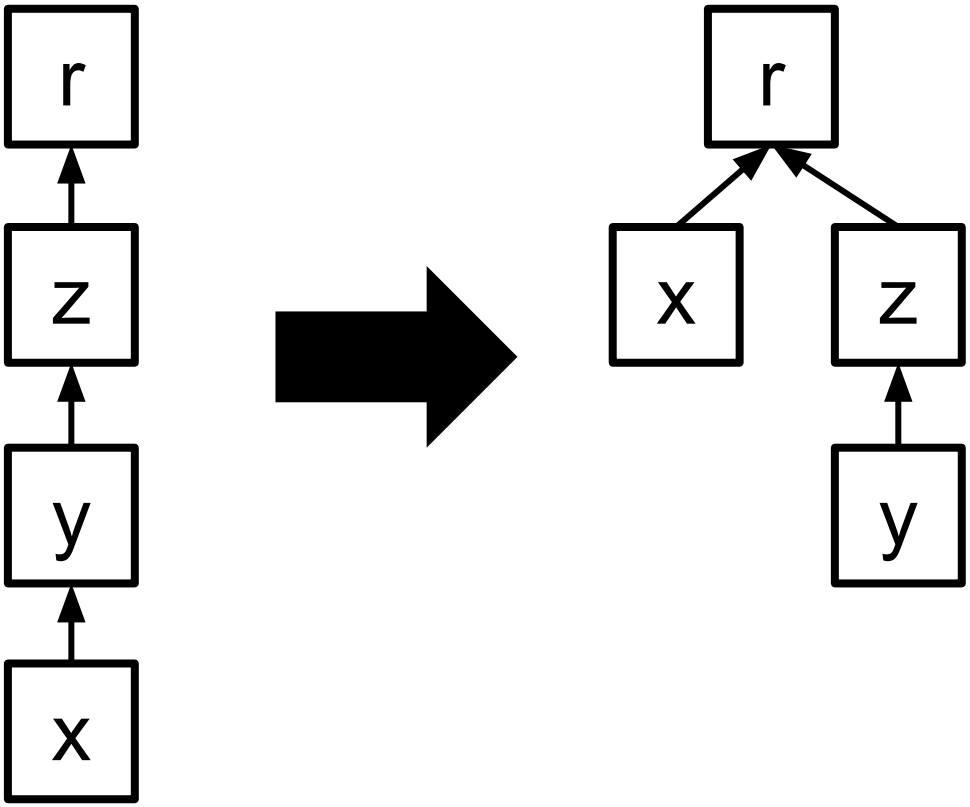
```
void DSU::Union(int x, int y){
    int raiz_x = Find(x);
    int raiz_y = Find(y);
    if(raiz_x != raiz_y){
        if(rank[raiz_x] < rank[raiz_y]{
            conjuntos[raiz_x] = conjuntos[raiz_y];
        } else if(rank[raiz_x] > rank[raiz_y]{
            conjuntos[raiz_y] = conjuntos[raiz_x];
        } else {
            conjuntos[raiz_x] = conjuntos[raiz_y];
            rank[raiz_x]++;
        }
    }
}
```

DSU com árvores - Union por rank

- Agora a junção das árvores é feita de forma controlada, o que nos garante uma complexidade logarítmica para o *find*.
- Mas ainda existe mais uma otimização que podemos fazer.
- A idéia é comprimir os caminhos de forma que o pai do elemento seja a raiz da árvore.
- Vamos fazer esse procedimento apenas sob demanda, pois caso contrário iríamos aumentar a complexidade da função *union*.

DSU com árvores - *path compression*

- A ideia é que após chamar a função find para um nó x, o pai de x será a raiz de sua árvore.



DSU com árvores - *path compression*

```
int DSU::Find(int x){
   int novaRaiz = x;
   while(conjuntos[novaRaiz] != novaRaiz){
      novaRaiz = conjuntos[novaRaiz];
   }
   conjuntos[x] = novaRaiz;
   return novaRaiz;
}
```

DSU com árvores - *path compression*

- Mas isso melhora a complexidade assintótica?
  - Para uma execução da função find não.
  - Mas se observarmos uma sequência de *m* operações e estivermos usando DSU com árvores, *path compression* e união por *rank*, conseguiremos melhorar nossa análise!
  - A sequência de operações aplicada a um DSU de tamanho *n* sob essas condições executa em tempo $O(m\alpha(n))$.
  - Conseguimos concluir que o custo amortizado de cada operação é $O(\alpha(n))$.

DSU com árvores - *path compression*

- Mas isso melhora a complexidade assintótica?
  - Neste contexto, $\alpha(n)$ é a **função inversa de Ackermann**.
  - A função de Ackermann é uma função que cresce extraordinariamente rápido, logo sua inversa cresce muito devagar!
  - Ela possui valor 4 ou menos para qualquer n que possa ser fisicamente escrito!
  - Isso nos leva a concluir que com essas otimizações, as operações de um DSU tem custo amortizado quase **constante**!

Distribuição de notícias

Em uma rede social, existem *n* usuários e *m* grupos. Vamos analisar o processo de distribuição de notícias.

Inicialmente, algum usuário *x* recebe um notícia.

Esse usuário encaminha essa notícia para seus amigos

dois usuários são amigos se existe pelo menos um grupo em comum

IMPLEMENTE UMA SOLUÇÃO USANDO DSU

Discuta as complexidades de tempo/espaço

Para cada usuário *x* determine o número de usuários que saberão a notícia se, inicialmente, somente *x* começou a distribuí-la.

Distribuição de notícias

Entrada: $1 \leq n,m \leq 5.10^5$, $0 \leq k_i \leq n$

primeira linha: n m (#usuários e #grupos)

m linhas seguintes:

i-ésima linha refere-se ao i-ésimo grupo

k_i lista_de_usuários_no_grupo_i

k_i = # usuários no grupo

Saída:

n inteiros, i-ésimo inteiro igual ao número de usuários que saberão a notícia se o usuário i começou a distribuí-la.

Entrada:
7 5
3 2 5 4
0
2 1 2
1 1
2 6 7

Saída:
4 4 1 4 4 2 2