INF 1010
Estruturas de Dados Avançadas

Partições dinâmicas

Partições de Conjuntos Dinâmicas

Estrutura de dados de conjuntos-disjuntos

- Uma estrutura de dados de conjuntos-disjuntos (Disjoint-set data structure), também conhecida como "Union Find" ou ainda "Merge Find", é uma estrutura de dados que manipula um conjunto de elementos disjuntos particionados em subconjuntos.

Definições

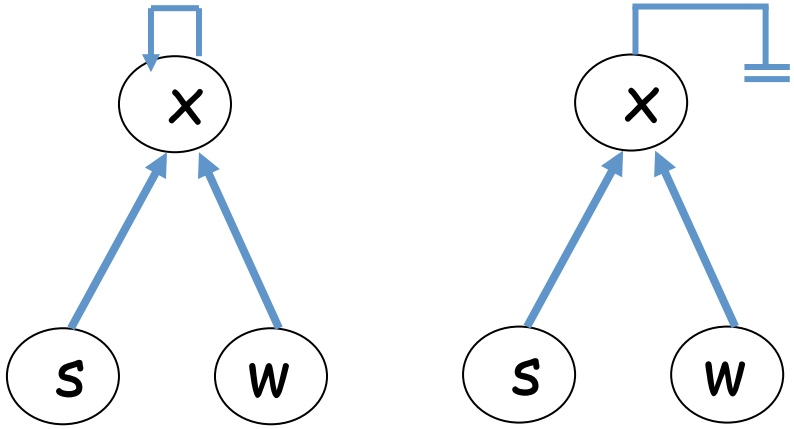
- Universo: $U = \{x_1, x_2, \ldots, x_n\}$
- Coleções: $C = \{S_1, \ldots, S_k\}$ de conjuntos disjuntos
- $S_i \subseteq U$ para qualquer i
- $\cup S_i = U$ (COBERTURA)
- $S_i \cap S_j = \phi$ sempre que $i \neq j$ (DISJUNTOS)
- Cada S_i é identificado por um de seus elementos $x \in S_i$ (**REP**RESENTANTE)

Operações Básicas

- MakeSet(x) : Cria um conjunto com um só elemento. Esta configuração também é conhecida como singleton;
  - $S_i = \{x\}$
- Find(x): Informa de que conjunto um elemento faz parte, retornando um elemento que representa o conjunto. Também útil para determinar se dois elementos estão no mesmo conjunto;
  - {k,j,n,o,y,**w**,x,z} := w
- Union(x,y): Junta dois conjuntos em um único. Substituindo assim S_i e S_j em C por um novo conjunto S_k.
  - $S_i \cup S_j = S_k$

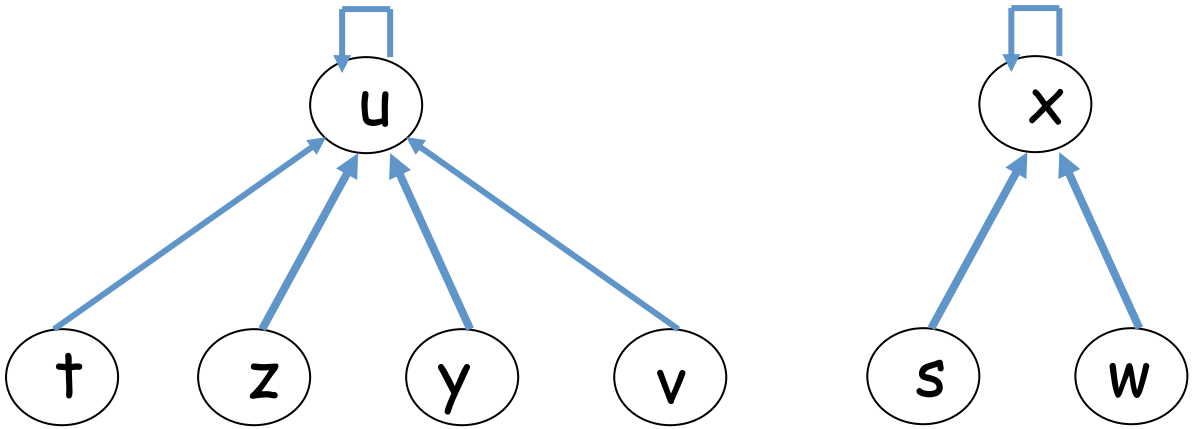
Árvore Reversa

- Uma forma tradicional de se representar e implementar estas estruturas é usando árvores reversas (Reversed Trees)
- Nesta configuração, cada nó aponta para o seu pai.
- Um nó da estrutura de dados Union-Find é um líder se ele é a raiz da árvore (ou seja, o nó no topo de uma árvore).
- Depois de um nó deixar de ser a raiz, ele nunca pode se tornar uma raiz novamente.
- O apontamento do nó raiz pode ser definido de duas formas principais:
  - a raiz aponta para si mesmo
  - a raiz aponta para NULL



Exemplo de Conjuntos Disjuntos

- Um conjunto é representado pela sua raiz.

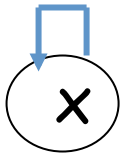


$S_1 = \{t,\underline{u},v,y,z\}$ $S_2 = \{s,w,\underline{x}\}$

Exemplo de Operações: MakeSet(x)

- O MakeSet(x) cria um conjunto com um elemento só (singleton).

MakeSet(x) = {x}

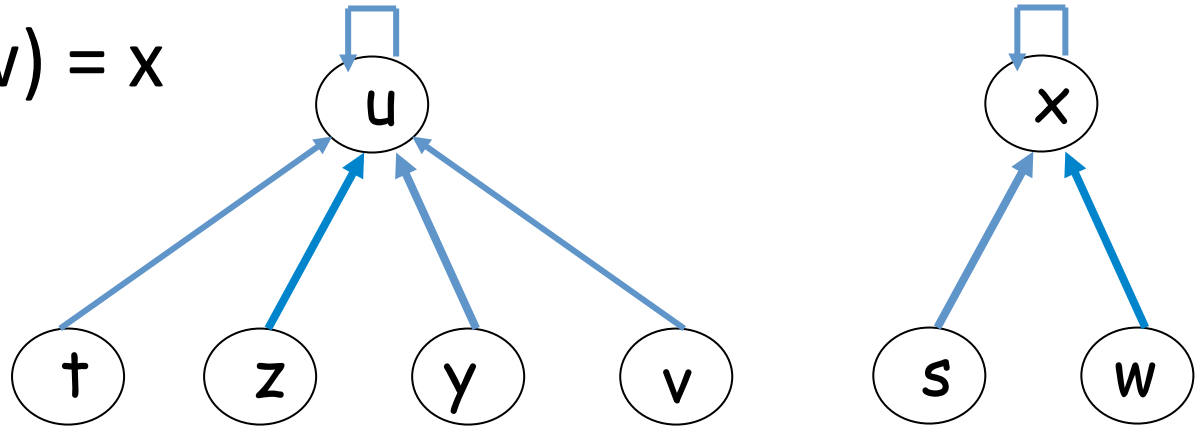


Exemplo de Operações: Find(x)

- O Find(x) retorna o elemento que representa o conjunto naquele instante. Assim o algoritmo busca a raiz da árvore que contém o elemento.

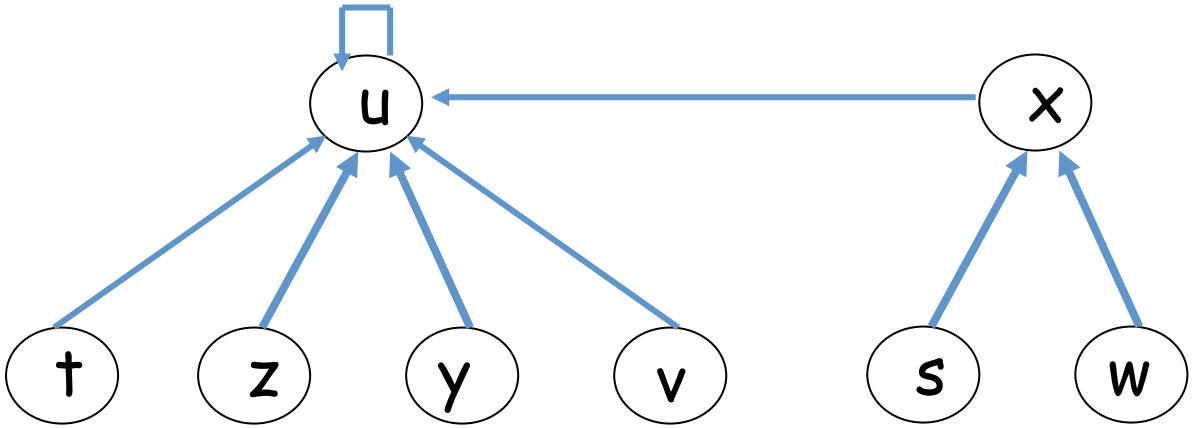
Find(z) = u

Find(w) = x



Exemplo de Operações: Union(u,x)

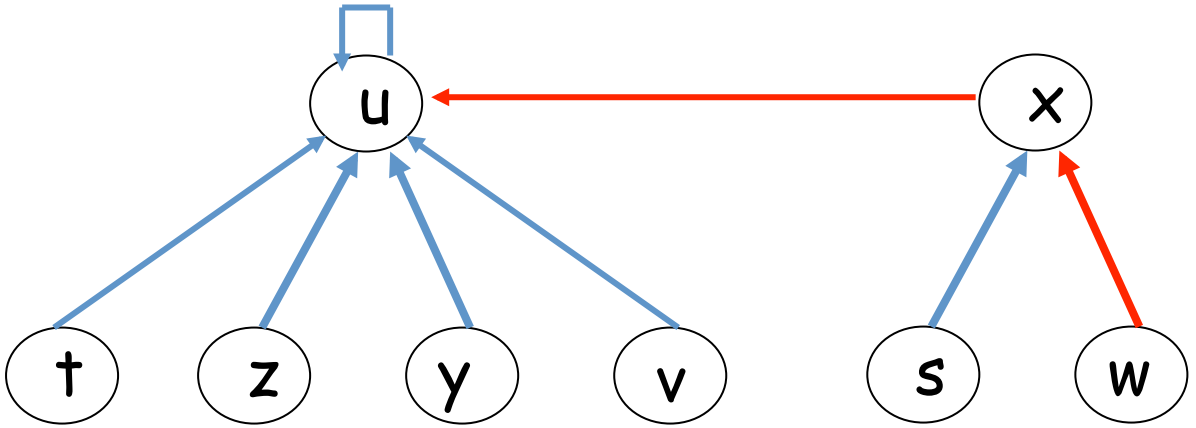
- O Union(u,x) une dois grupos fazendo com que a raiz de um grupo aponte para a raiz do outro grupo. Desta forma uma árvore vira a sub-árvore da outra.
- Union(u,x) => $S_1 \cup S_2 = \{t,\underline{\mathbf{u}},v,y,z, s,w,x\}$.



Já é possível perceber aqui, que conforme as uniões forem realizada, uma árvore completamente degenerada pode ser criada, fazendo com que as operações de Find se realizem em tempo linear O(n)

Exemplo de Operações: Find(w)

- O comando de Find vai percorrendo toda a estrutura até encontrar o elemento que representa o conjunto.
- Find(w) = u



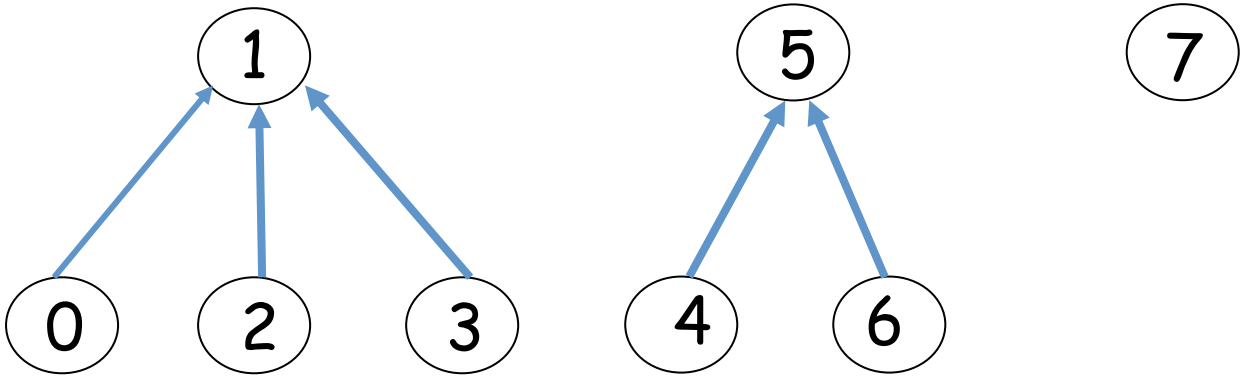
Representação Simplificada por Vetor

- Um exemplo didático de como estrutura de dados de conjuntos-disjuntos pode ser implementada é com um simples vetor contínuo. Cada elemento do vetor é diretamente mapeado para os elementos do universo. Cada elemento do vetor possui um apontador para o pai dele.

Representação Simplificada por Vetor

- Veja aqui um exemplo de conjuntos de alguns números. Note que neste exemplo a representação de raiz é quando o elemento aponta para -1.

elemento	0	1	2	3	4	5	6	7
ponteiro	1	-1	1	1	5	-1	5	-1



Algoritmo Simples de Union-Find

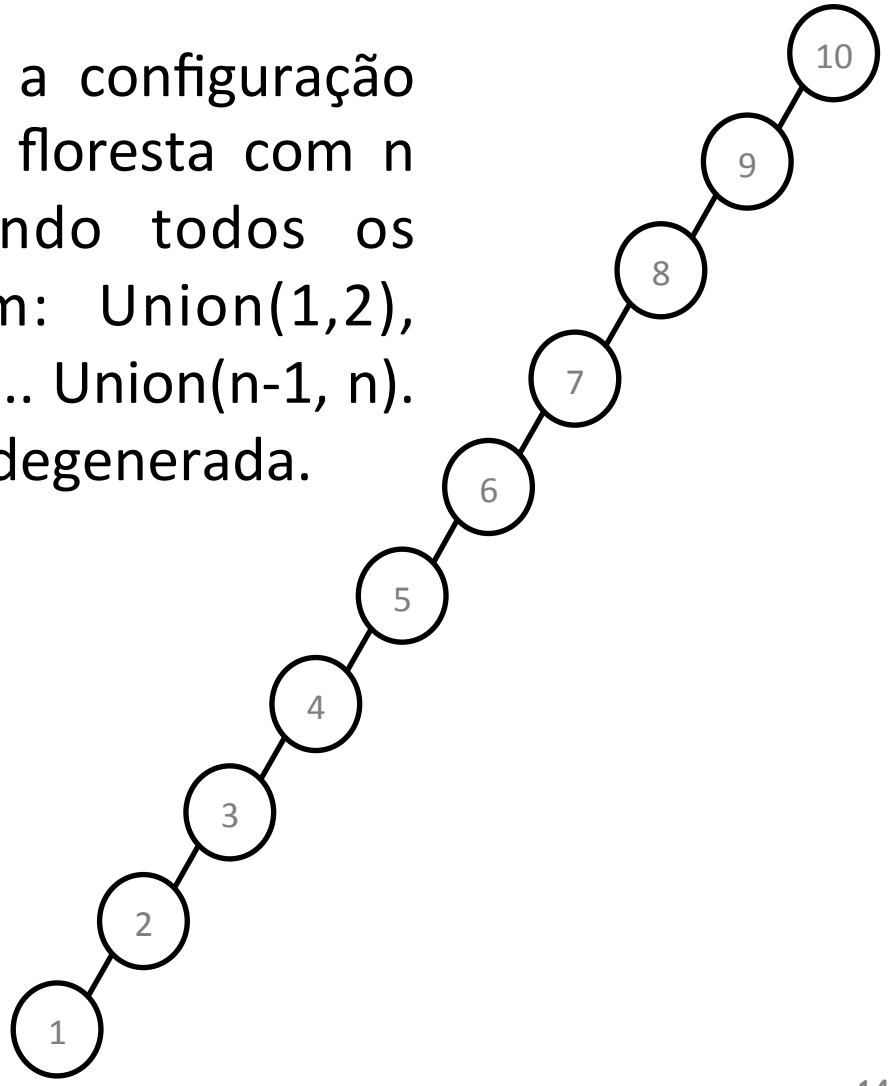
```
void simpleCreateUnionFind(int p[], int size) {
        int i;
        for(i=0;i<size;++i) p[i] = -1;
}

int simpleFind(int p[],int u){
        return (  ( p[u] == -1 ) ? u : simpleFind(p,p[u]) );
}

void simpleUnion(int p[], int u,int v) {
        p[u] = v;
}
```

Problemas do Algoritmo Simples

- Num primeiro instante a configuração inicial consiste de uma floresta com n nós (singletons). Unindo todos os elementos em ordem: Union(1,2), Union(2,3), Union(3,4), ... Union(n-1, n). Resulta em uma árvore degenerada.



Union Seguro

- Para fazer a união de dois grupos é necessário informar a raiz do conjunto, caso isso não seja feito o algoritmo ira ter um comportamente inconsistente.
- Contudo fazer os Finds dentro da rotina de Union para encontrar as raizes vai fazer com que o algoritmo se torne O(n).

Algoritmo de Union Seguro

```
void simpleSafeUnion(int p[], int u,int v) {
        u = simpleFind(p, u);
        v = simpleFind(p, v);
        p[u] = v;
}
```

Eficiência

- MakeSet
  - ACESSO DIRETO A NÓ : O(1)
- Find
  - NECESSÁRIO PERCORRER OS ELEMENTOS : O(n)
- Union
  - TEMPO CONSTANTE EM SI : O(1)
  - PORÉM É NECESSÁRIO ENCONTRAR OS REPRESENTANTES CASO NÃO SE TENHA CERTEZA DA ORDERM DE INSERÇÃO : O(n)

Eficiência dos Algoritmos

- Caso seja necessário unir todos os elementos e depois verificar cada um a que conjunto ele pertence, todos os elementos precisariam sofrem um Union e depois um Find.
- O tempo de uma **Union** é constante assim unir todos os elemento leva o tempo O(n). Mas cada execução do **find** segue uma sequência até a raiz do conjunto.
- Como o tempo para cada **find** de um elemento no nível i de uma árvore é O(i), o tempo total necessário para processar os n **find**s é:

$$O(\sum_{i=1}^{n} i) = O(n^2)$$

Implementação com Florestas

- Uma forma de se implementar os elementos dos conjuntos é por uma florestas com nós e ponteiros. Dessa forma será possível ter maior flexibilidade no tipo de dado a ser usado.

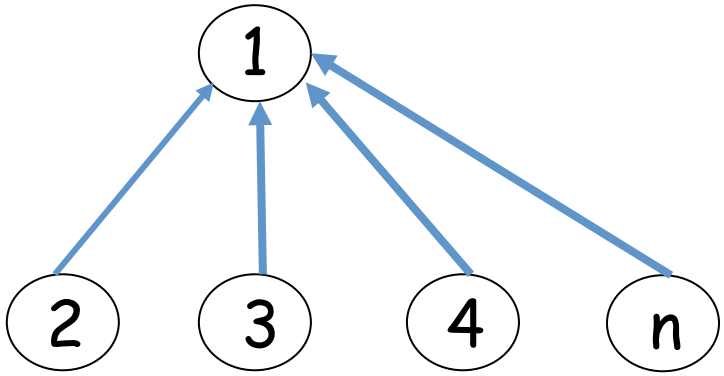
Algoritmo de Union-Find com Florestas

```
typedef struct forest_node_t {
   void* value;
   struct forest_node_t* parent;
   int rank;
} forest_node;

forest_node* MakeSet(void* value) {
   forest_node* node = malloc(sizeof(forest_node));
   node->value = value;
   node->parent = NULL;
   node->rank = 0;
   return node;
}
```

Regra de Ponderação WeightedUnion

- Melhorar o desempenho do algoritmo **Union**, não permitindo a criação de árvores degeneradas:
  - **Union(i,j)**: Se o número de nós na árvore com raiz i é menor que o número na árvore com raiz j, então faça j ser o pai de i; caso contrário, faça i ser o pai de j.

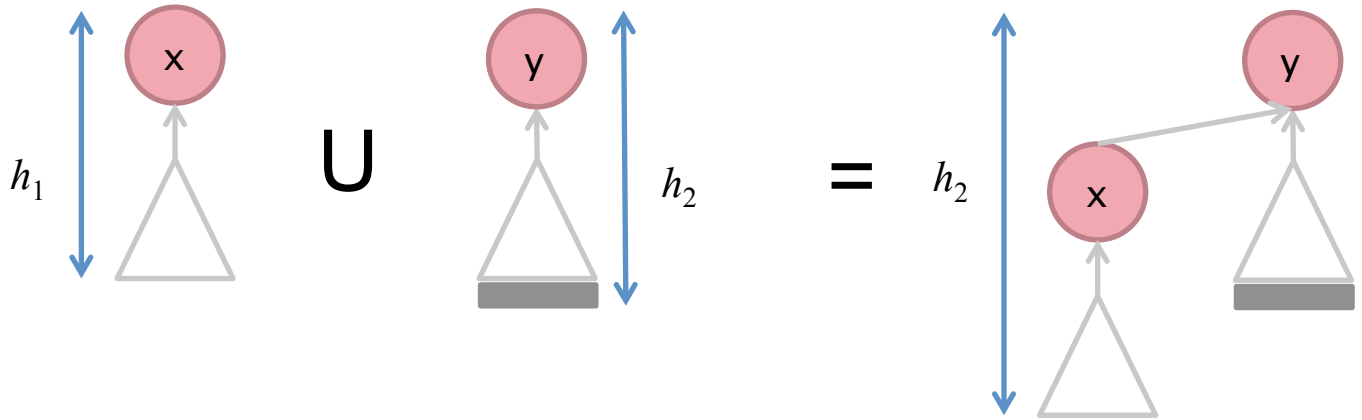


Algoritmo União Ponderada

- Em outras palavras, sempre colocar a menor árvore na maior árvore
- Cada representante do conjunto possui um valor que define quem é maior que quem.

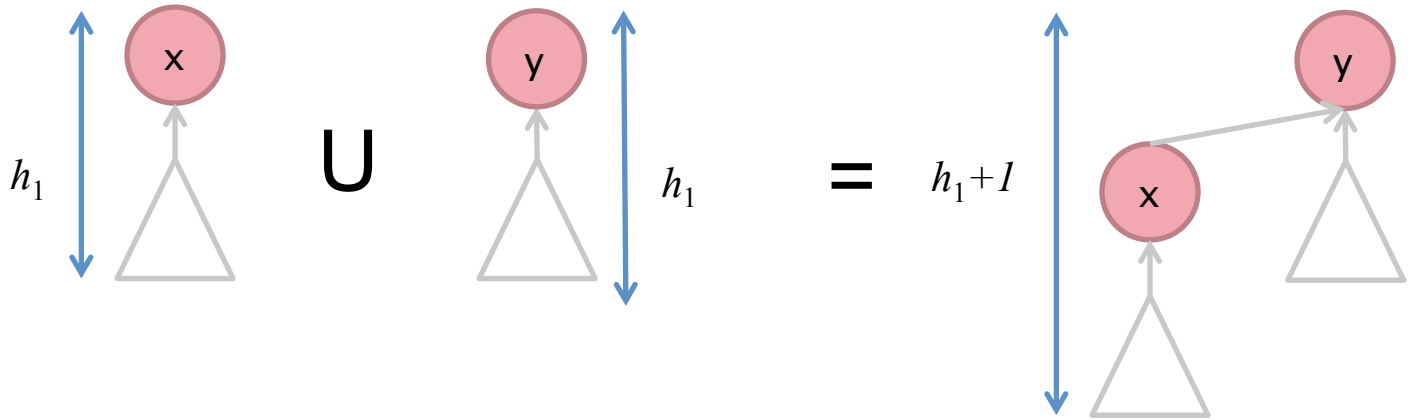
Algoritmo União por Altura

- No algoritmo de união por altura é verificada a altura de cada conjunto para determinar a união.



Algoritmo União por Altura

- Se forem iguais as alturas, a ordem não importa, e a altura aumentará de um.



Resumo de União por Altura

- BAIXO PENDURA NO ALTO

  ALTURA **NÃO** CRESCE

- EMPATE ENTÃO INDIFERENTE

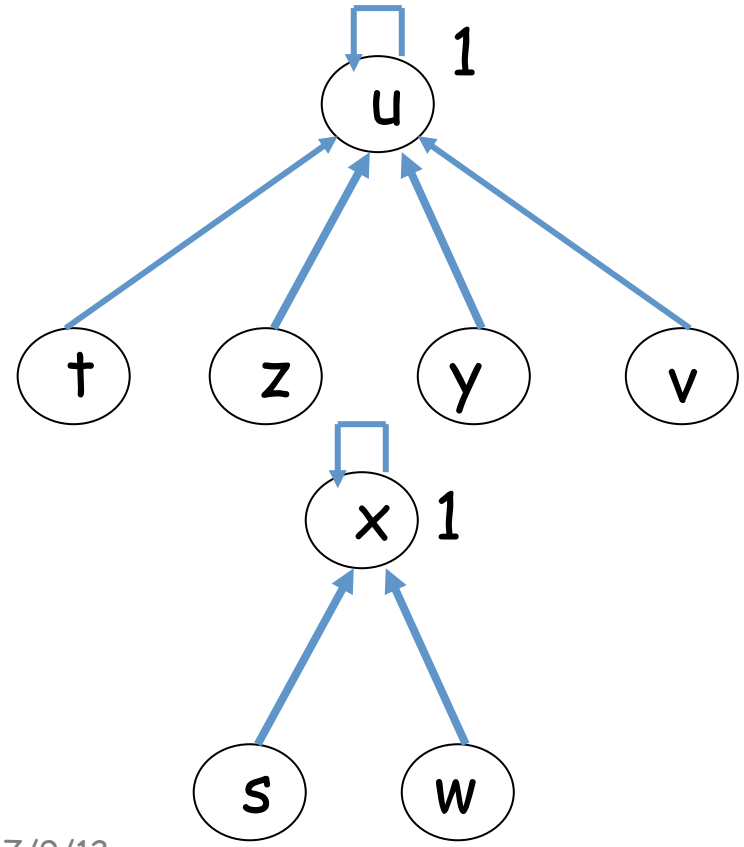
  ALTURA CRESCE DE **+1**

Algoritmo União por Altura

```
forest_node* Union(forest_node* node1, forest_node* node2) {
   if (node2-> rank > node1-> rank) {
      node1->parrent = node2;
      return node2;
    } else if (node1-> rank > node2->rank) {
      node2->parrent = node1;
   } else { /* iguais */
      node2->parrent = node1;
      node1-> rank++;
   }
   return node1;
}
```

Exemplo com Floresta

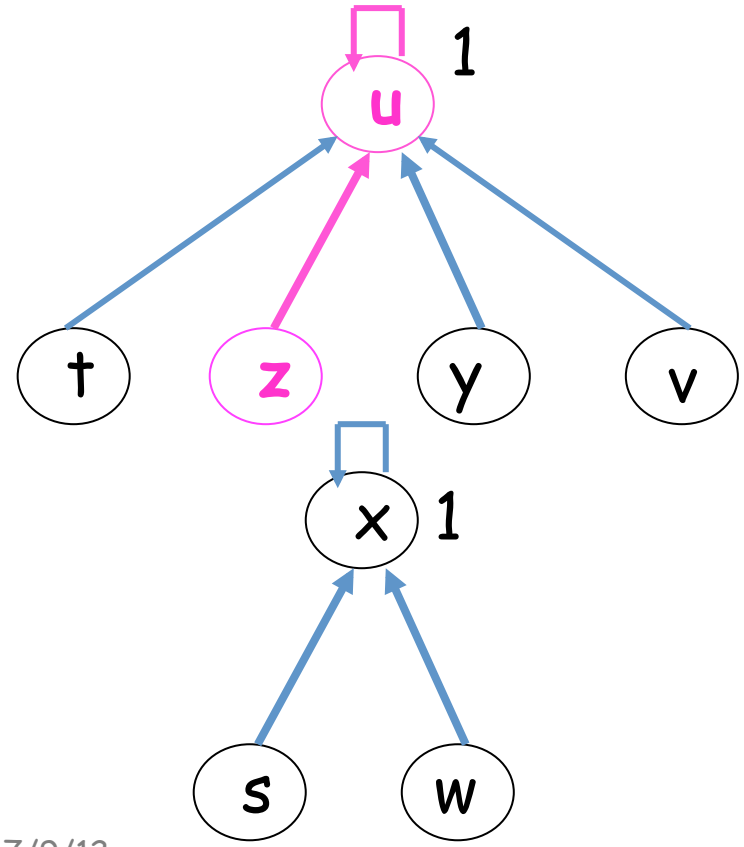
{ {t,u,v,y,z}, {s,w,x} }



	REP	ALTURA
s	x	0
t	u	0
u	u	1
v	u	0
w	x	0
x	x	1
y	u	0
z	u	0

Exemplo com Floresta

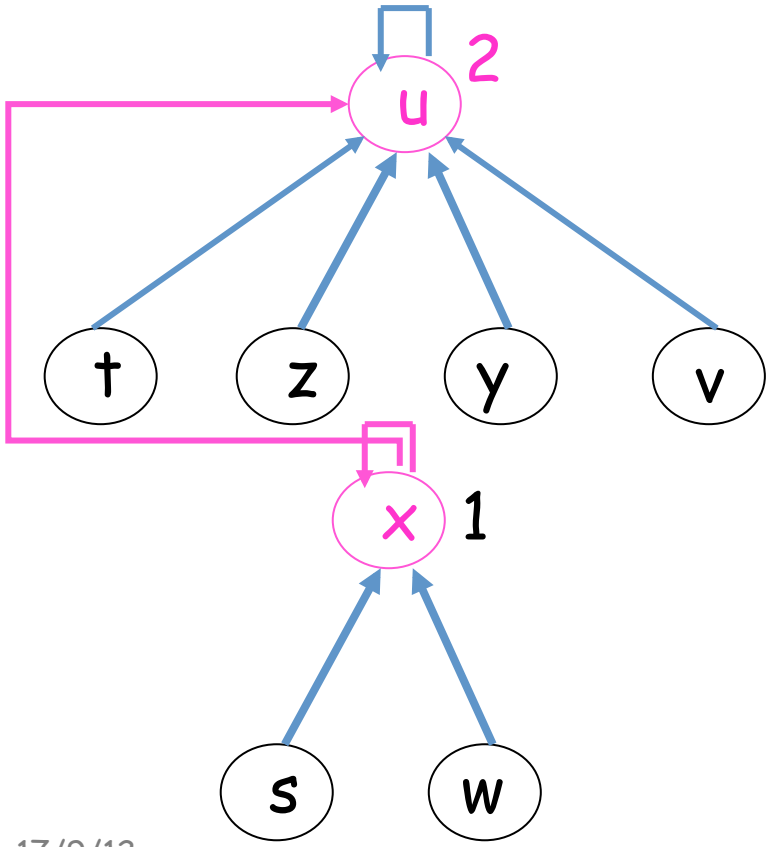
- FIND_SET (z)



	REP	ALTURA
s	x	0
t	u	0
u	u	1
v	u	0
w	x	0
x	x	1
y	u	0
z	u	0

Exemplo com Floresta

- UNION (u, x)



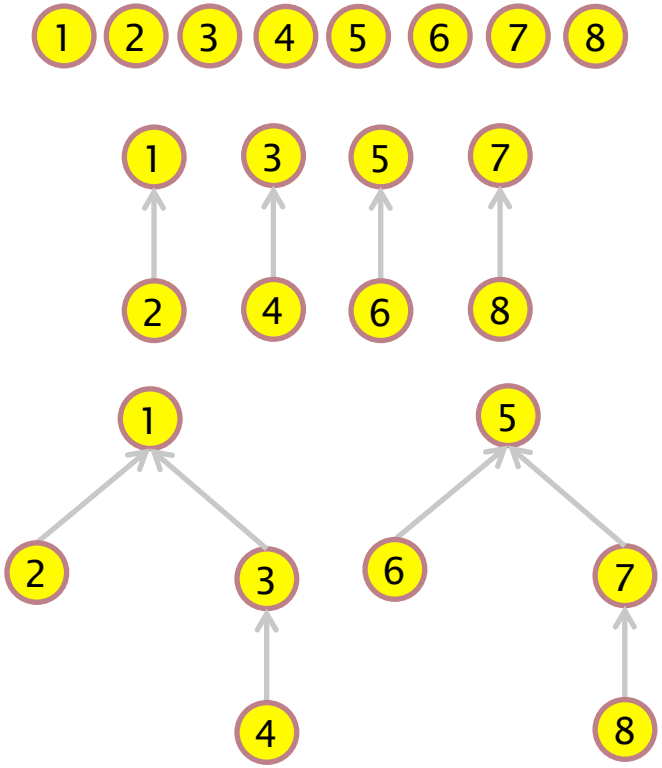
	REP	ALTURA
s	x	0
t	u	0
u	u	2
v	u	0
w	x	0
x	u	1
y	u	0
z	u	0

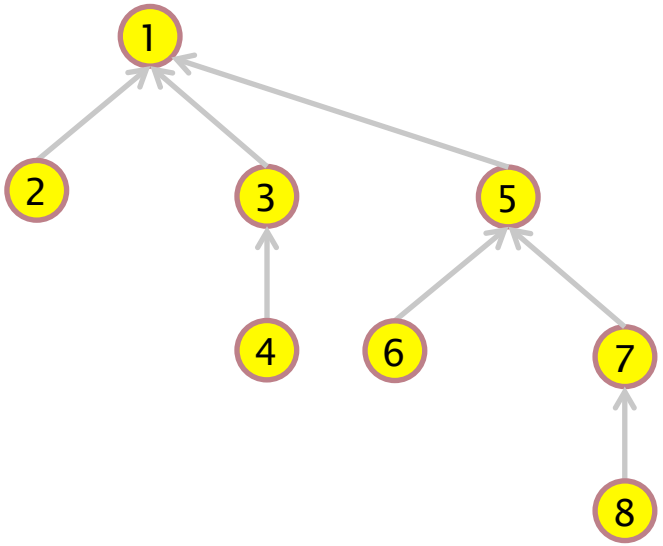
Definindo o Posto (rank)

- Os postos (ranks) definem a altura das árvores
- Depois de um nó deixar de ser um líder seu posto não muda mais.
- Postos são monotonicamente crescente nas árvores invertidas, conforme se viaja de um nó para sua raiz.
- Quando um nó obtém um posto k significa que há pelo menos 2^k-1 elementos em sua subárvore.

Exemplo de União por Altura

- **union(1,2),union(3,4),union(5,6), union(7,8), union(1,3), union(5,7), union(1,5)**





A altura máxima de cada árvore é $\log_2 m$

União por Altura

- $\mathbf{n_i(x)}$

  número de nós na árvore de raíz x após a *i-ésima união envolvendo* x

- $\mathbf{n_0(x)} = 1$

- $\mathbf{h_i(x)}$

  altura da árvore de raíz x após a *i-ésima união envolvendo* x

- $\mathbf{h_0(x)} = 0$

União por Altura

- $\mathbf{n_i(x) \geq 2^{h_i(x)}}$

$$n_{i+1}(x) = n_i(x) + n_i(y)$$

$$n_{i+1}(x) \geq 2^{h_i(x)} + 2^{h_i(y)}$$

- baixo no alto

$$n_{i+1}(x) \geq 2^{\max\{h_i(x),h_i(y)\}} = 2^{h_{i+1}(x)}$$

- empate

$$n_{i+1}(x) \geq 2^{h_i(x)+1} = 2^{h_{i+1}(x)}$$

- $\mathbf{h_i(x) \leq \lfloor lg(n) \rfloor}$

$$n \geq n_i(x) \geq 2^{h_i(x)}$$

$$lg(n) \geq h_i(x)$$

$h_i(x)$ é inteiro

- FIND_SET

$$\mathbf{O(\lfloor lg(n) \rfloor)}$$

Eficiência

- MakeSet, Union
  - O TEMPO PARA FAZER UNIÃO CRESCEU PORÉM AINDA É UM ACESSO DIRETO A NÓS
  - O(1)
- Find
  - PERCORRER TRAJETÓRIA ATÉ RAÍZ: O(log n)

Lema de União por Altura

- Assuma que se inicia com uma floresta de árvores, cada uma tendo um nó. Seja T a árvore com m nós criados como resultado de uma sequência de m uniões ponderada. Assim a altura de T não é maior que $\log_2 m$.
- Desta forma um Find tem tempo de O(log m). Assim a operação de m Unions + n Finds leva : O (m + n*log(m))
- Ainda pode ser levado em conta o fato que é necessário O(n) para iniciar os n singletons.

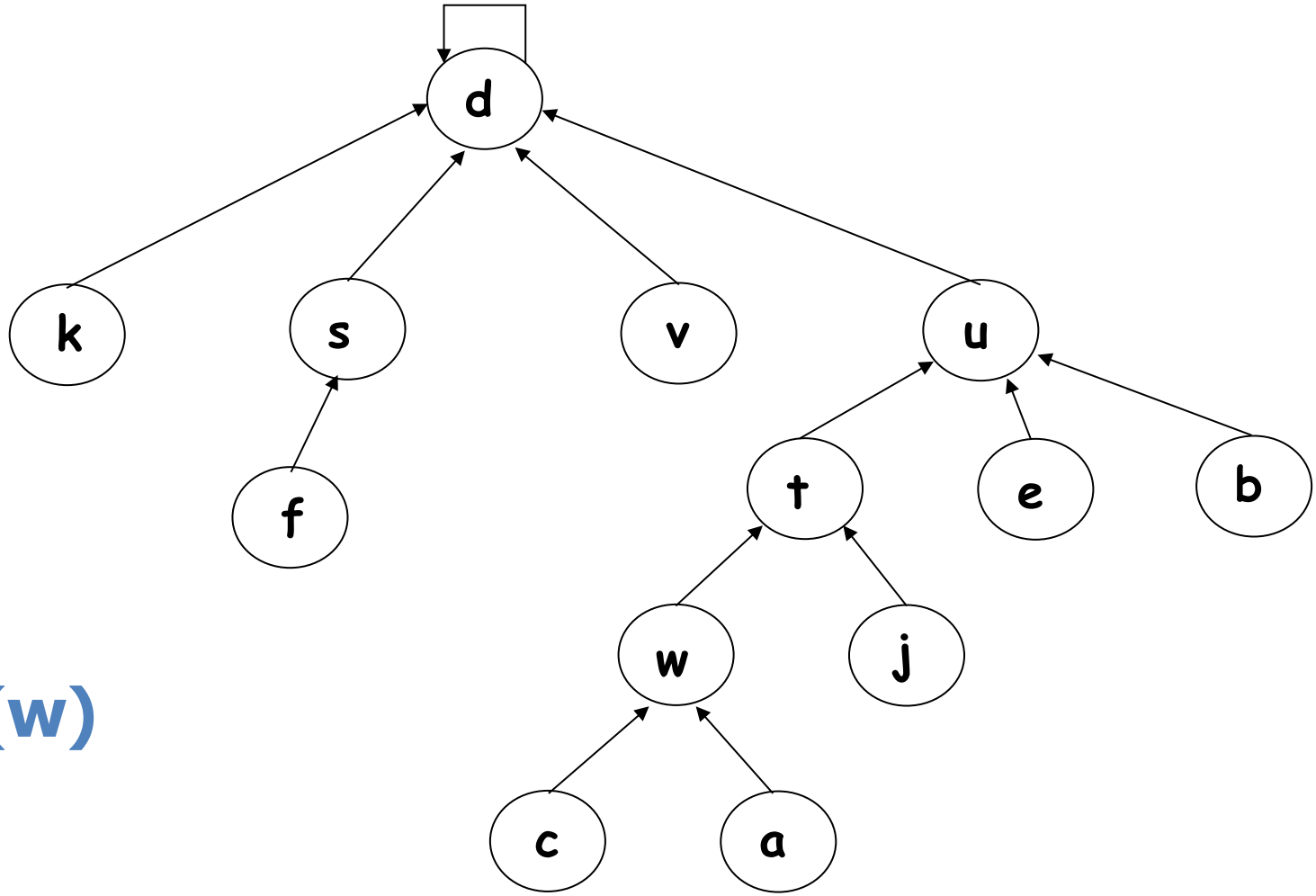
Algoritmo do Desmoronamento **CollapsingFind**

- Embora árvores degenaradas não são mais geradas, cada vez que um Find é realizado, este leva O(log m) de tempo de execução. A regra de desmoronamento, evita esse tempo em chamadas repetidas, comprimindo o caminha após sua execução.
  - Se **j** é um nó no caminho de **i** até sua raiz, faça **j** apontar para raiz.

Algoritmo do Desmoronamento

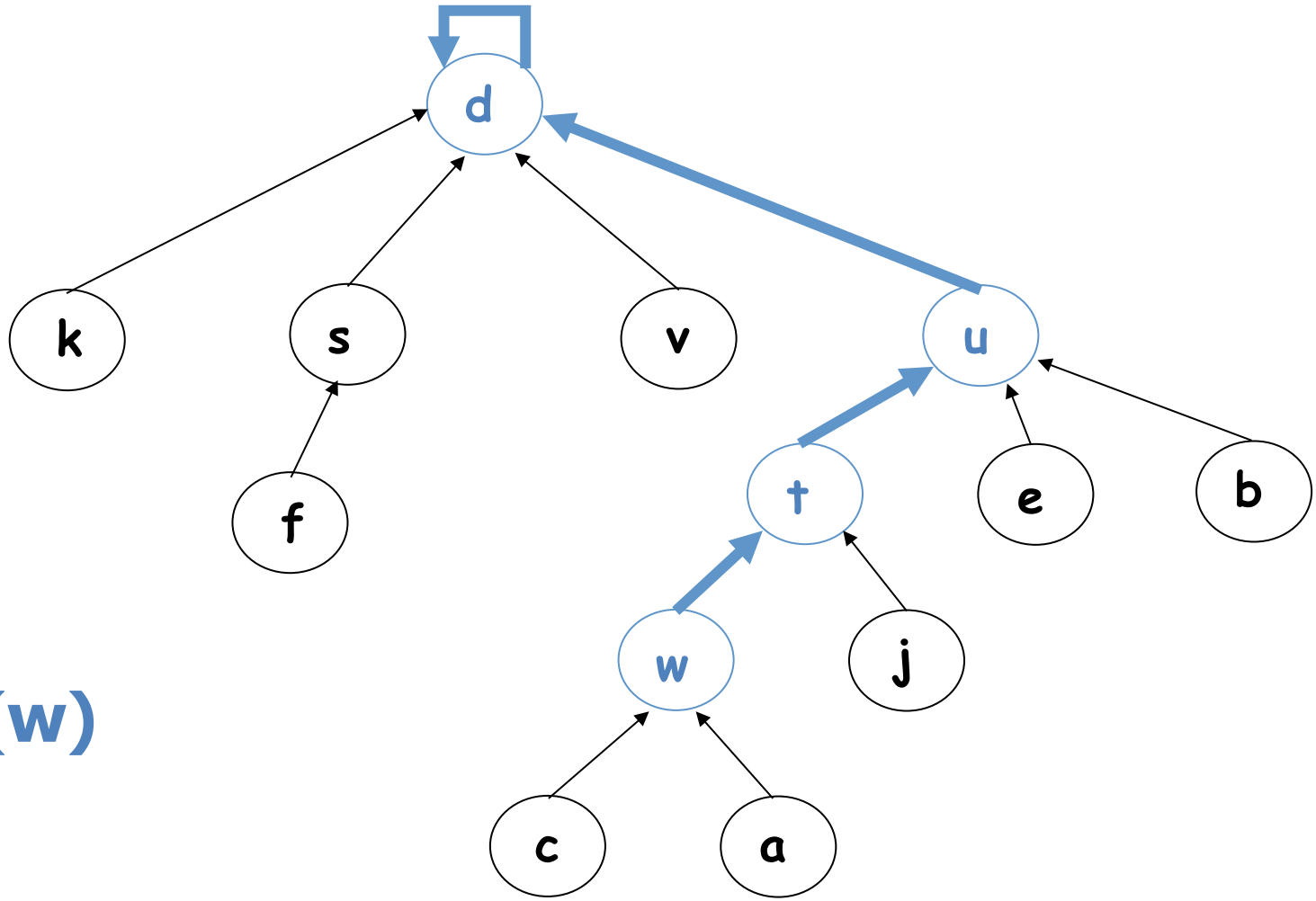
```
forest_node* Find(forest_node* node) {
   forest_node* temp;
   forest_node* root = node;
   while (root-> parrent != NULL) {
      root = root-> parrent;
   }
   while (node->parrent != root) {
      temp = node->parrent;
      node->parrent = root;
      node = temp;
   }
   return root;
}
```

Compressão de Trajetória



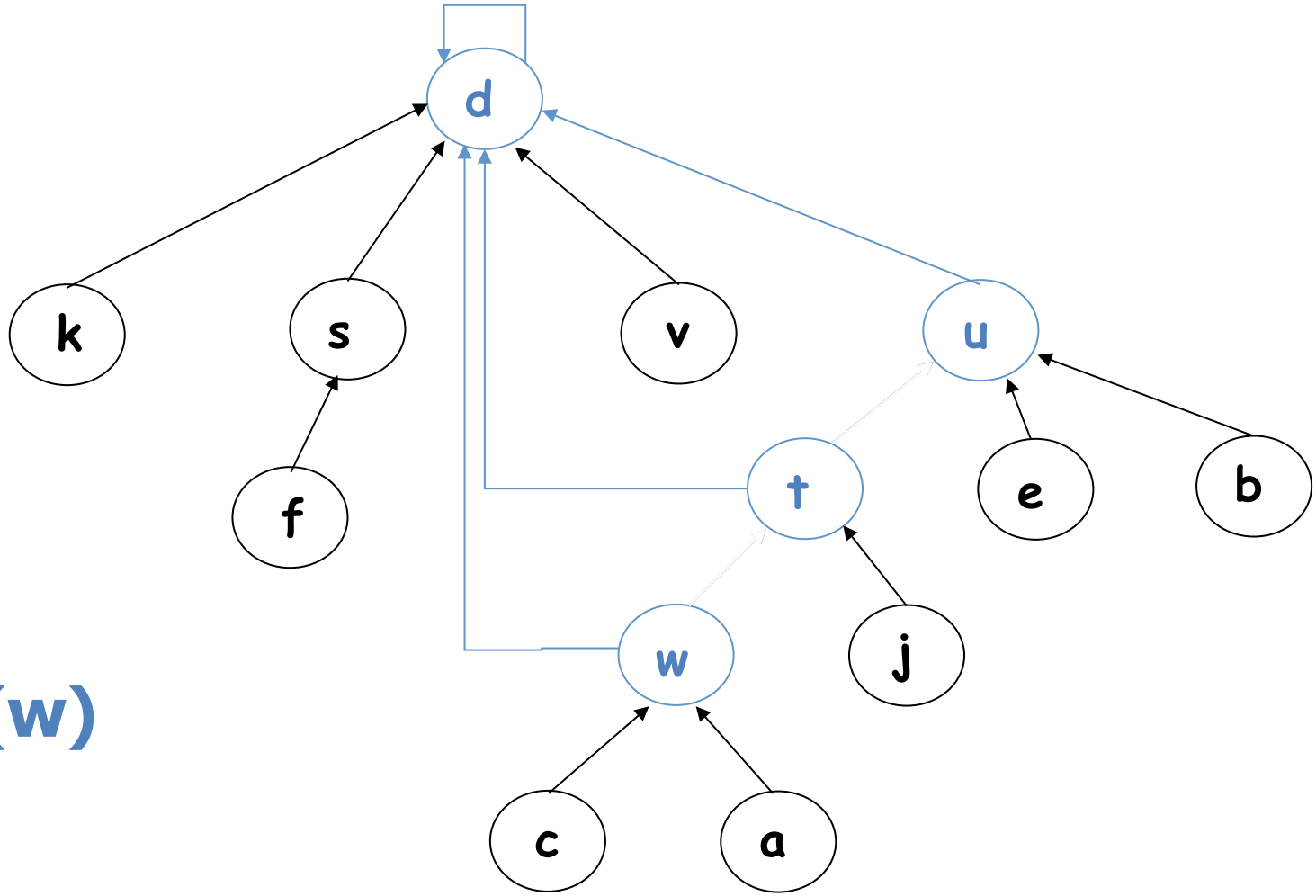
Find(w)

Compressão de Trajetória



Find(w)

Compressão de Trajetória



Find(w)

Exemplo de Ganho de Desempenho

- Usando o exemplo de árvore criada após a união ponderada de 8 elementos, um Find(8) requer três movimentos de nós até chegar a raiz. Assim se executarmos 10 vezes o Find(8) seriam necessários 30 movimentos.
- Usando o algoritmo de desmoronamento para os Finds, numa primeira execução é necessário os três movimentos e refazer duas conexões. Porém na segunda execução o Find já encontra o 8 em apenas um movimento. Levando a um custo total de apenas 14 movimentos.
- Dessa forma, embora o desmoronamento aumente o tempo de um elemento num primeiro insntante, para as chamadas seguintes existe uma redução significativa do tempo total.

Função de Ackerman

- A função A(x) é definida por indução:

$A_{i+1}(x) = A_i(A_i(A_i(A_i(A_i...(x)))))$, onde A é feito x+1 vezes

$A_1(x) = 2^{x+1}$

$A_2(x) = A_1(A_1(A_1(A_1(....x)))) > 2^{x+1}$

$A_3(x) = A_2(A_2(A_2(A_2(....x)))) > 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$

- A **função inversa de Ackerman** é

$\alpha(n) = \min\{ k: A_k(1) > n \}$ < 5 na prática

Tarjan e Van Leeuwen

- Em um floresta de árvores com somente nós singletons.
- Seja T(f, u) o tempo máximo necessário para processar qualquer sequência de f **find**s e u **union**s.
- Assuma que u ≥ ½ n. Então:

$k1[n + f\,\alpha(f + n, n)] \leq T(f, u) \leq k2[n + f\,\alpha(f + n, n)]$
para algumas constantes positivas k1 e k2.

História

- Embora as ideias utilizadas em florestas de conjuntos disjuntos sejam antigas, Robert Tarjan foi o primeiro a provar o limite superior (e uma versão restrita do limite inferior) em termos da função inversa Ackermann em 1975.
- Até este momento o melhor tempo por operação conseguido, comprovado por Hopcroft e Ullman, é de O(log*n), o logaritmo iterado de n, uma outra função de crescimento lento (mas não tão lento como a função inversa de Ackermann).

A Função lg*

lg*n = o número de vezes necessário para termos log(2) repetidamente para chegarmos a 1.

- $\lg(2) = 1$ $\quad\quad$ $\lg^*(2) = \mathbf{1}$
- $\lg(\lg(4))=1$ $\quad\quad$ $\lg^*(4) = \mathbf{2}$
- $\lg(\lg(\lg(16)))=1$ $\quad\quad$ $\lg^*(16) = \mathbf{3}$
- $\lg(\lg(\lg(\lg(65.536))))=1$ $\quad\quad$ $\lg^*(65.536) = \mathbf{4}$
- $\lg(\lg(\lg(\lg(\lg(2^{65.536})))))=1$ $\quad\quad$ $\lg^*(2^{65.536}) = \mathbf{5}$
- $1 = 1$ $\quad\quad$ $\lg^*(1) = \mathbf{0}$

Teorema (Tarjan)

- Para uma sequencia de n Unions e Finds usando os algoritmos de União por Altura e Desmoronamento, o pior caso é O(n lg*n)
- O tempo médio é de O(lg*n) por operação.

Eficiência com Compressão de Trajetória

- Union

  $O(1)$

- Find

  $O(\lg^*(n))$ amortizado

- $\lg^*(n) \leq 5$ PARA $n \leq 2^{65.536}$

  ÁTOMOS NO UNIVERSO $= 10^{80} << 2^{65.536}$

Descendentes

- que acontece com a organização da árvore?

Trajetória de x a y

$$x \rightarrow y$$

Descendente

- $x \rightarrow y$
- x é descedente de y

Descendentes X Operações

Find

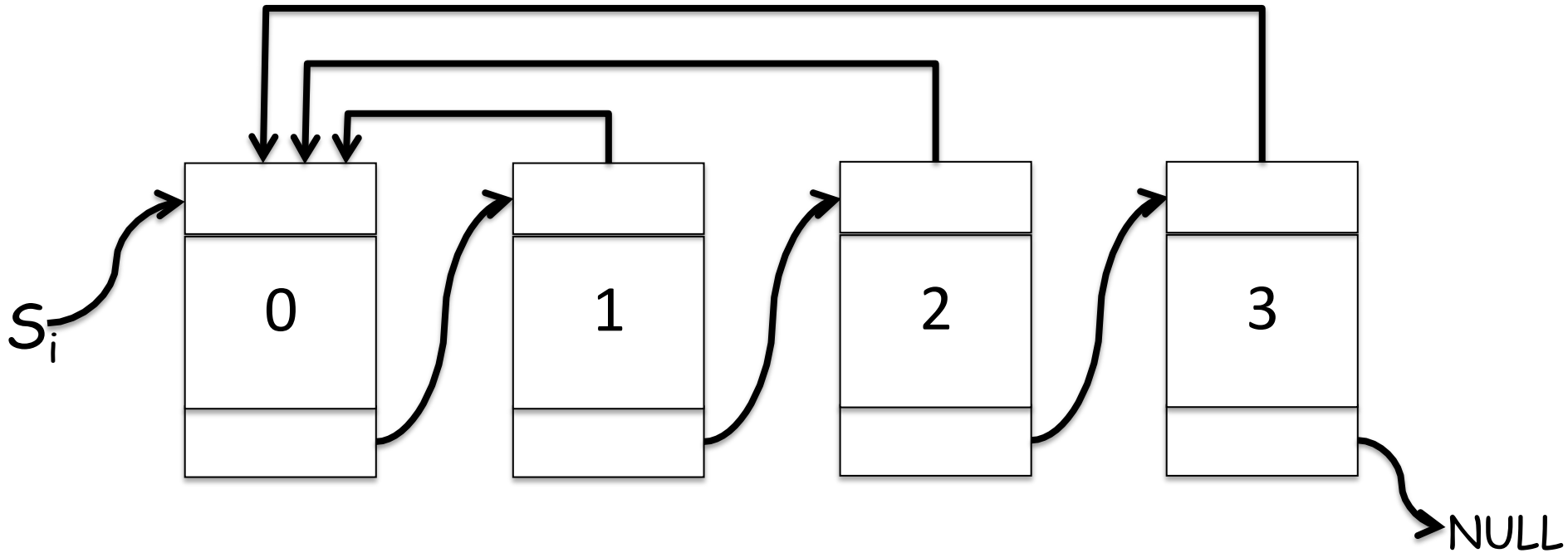
- Nós de uma trajetória perdem descendentes
- Nenhum nó ganha descendentes que ja não possuiam

Union

- Nenhum nó perde descendentes
- Um nó raíz ganha descendentes

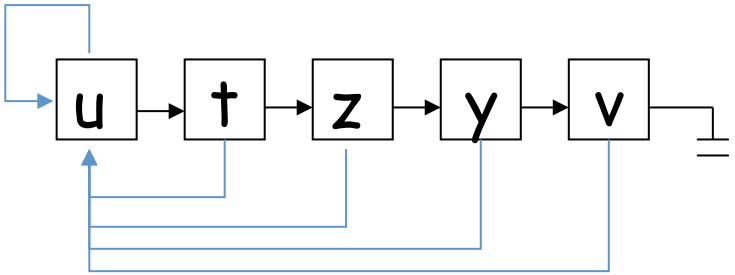
Implementação Por Lista Ligada

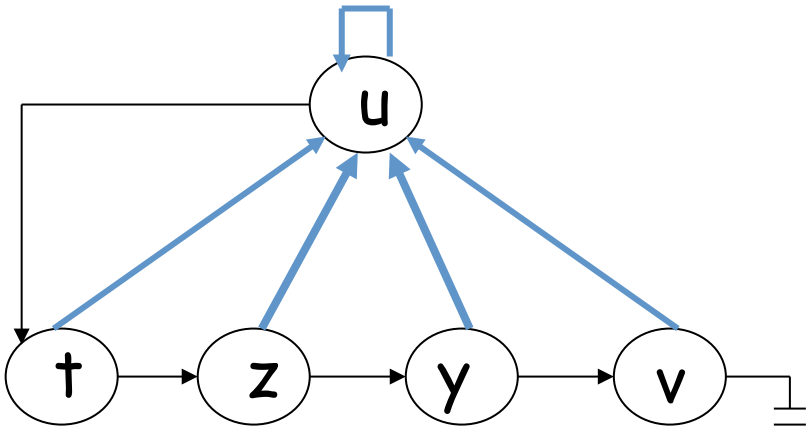
- Lista possui um ponteiro para o próximo e o nó representante do conjunto.



Exemplo de Implementação Por Listas Ligadas

{ {t,u,v,y,z}, {s,w,x}, … }





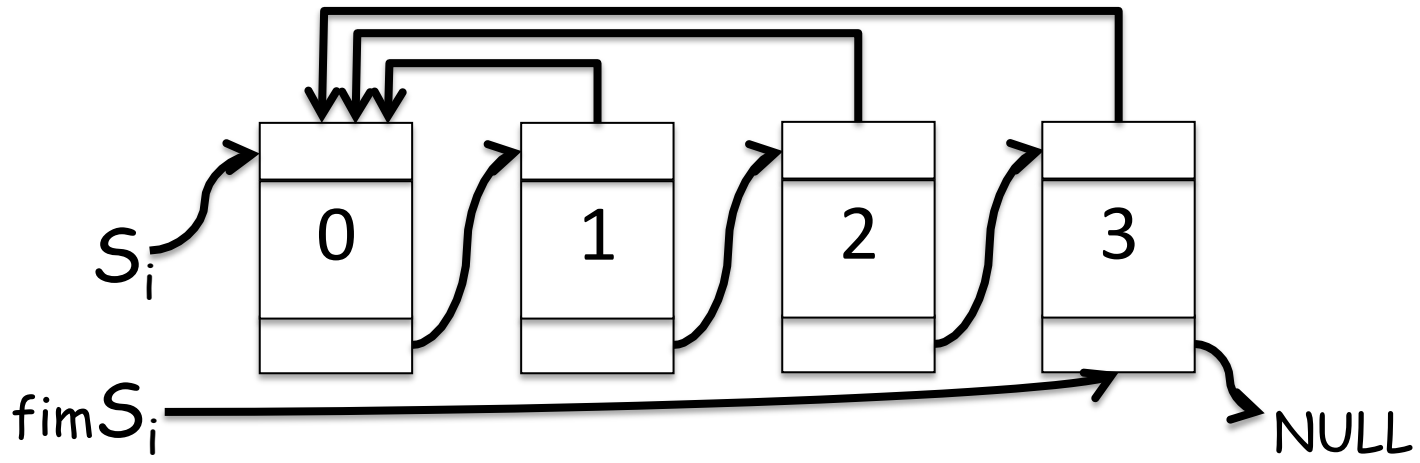
	REP	NEXT
s	x	w
t	u	z
u	u	t
v	u	NULL
w	x	NULL
x	x	s
y	u	v
z	u	y

Eficiência

- MakeSet, Find
  - ACESSO DIRETO A NÓS
  - O (1)
- Union
  - UNIR LISTAS: linear
  - ATUALIZAR REPRESENTANTE: linear

União Por Tamanho

- A união por tamanho, minimiza operações de atualizar os representantes, colocando sempre a menor lista na maior.
- Para se otimizar esse processo é também definido um ponteiro para o final da lista.



Exemplo de União por Tamanho

- MENOR PENDURA NO MAIOR

  PIOR CASO n/2

  AMORTIZADO lg (n)

- MAKE_SET, FIND _SET

  O (1)

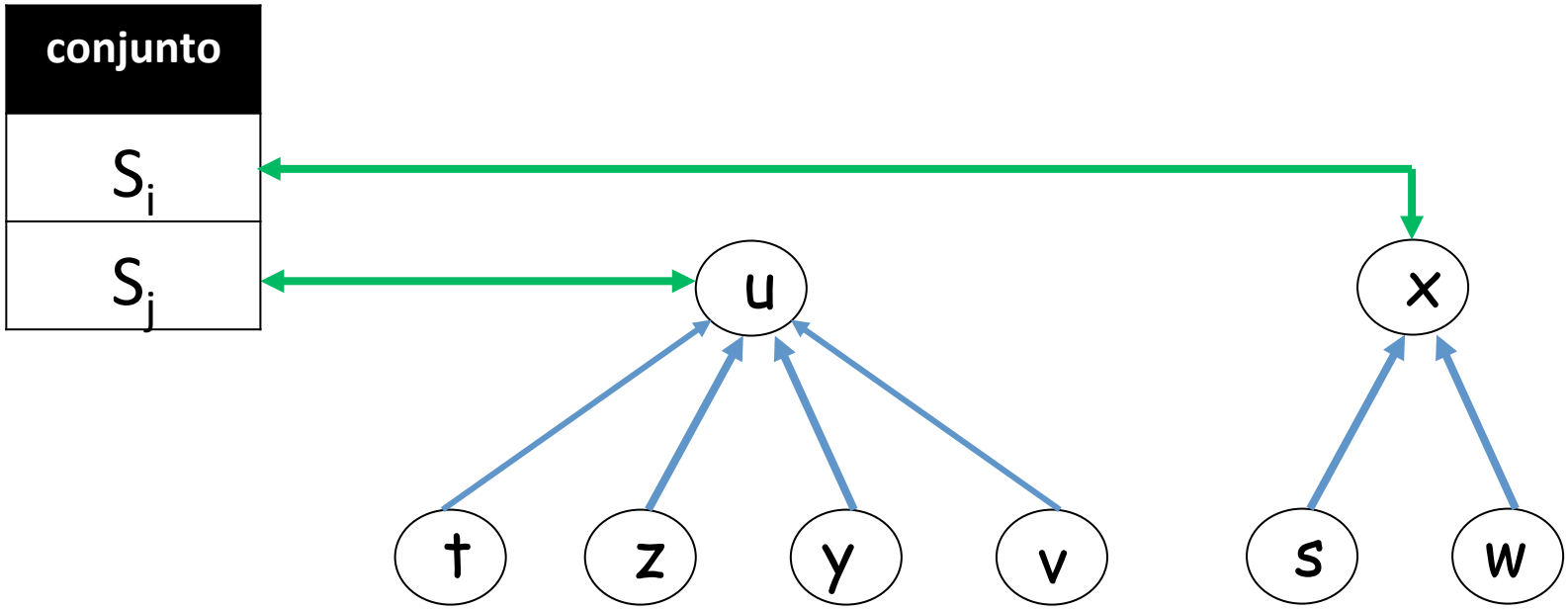
- m OPERAÇÕES

  O (m + n.lg(n))

	REP	NEXT	SIZE
s	**x**	w	1
t	**u**	z	1
u	**u**	t	5
v	**u**	NIL	1
w	**x**	NIL	1
x	**x**	s	3
y	**u**	v	1
z	**u**	y	1

FindPointer

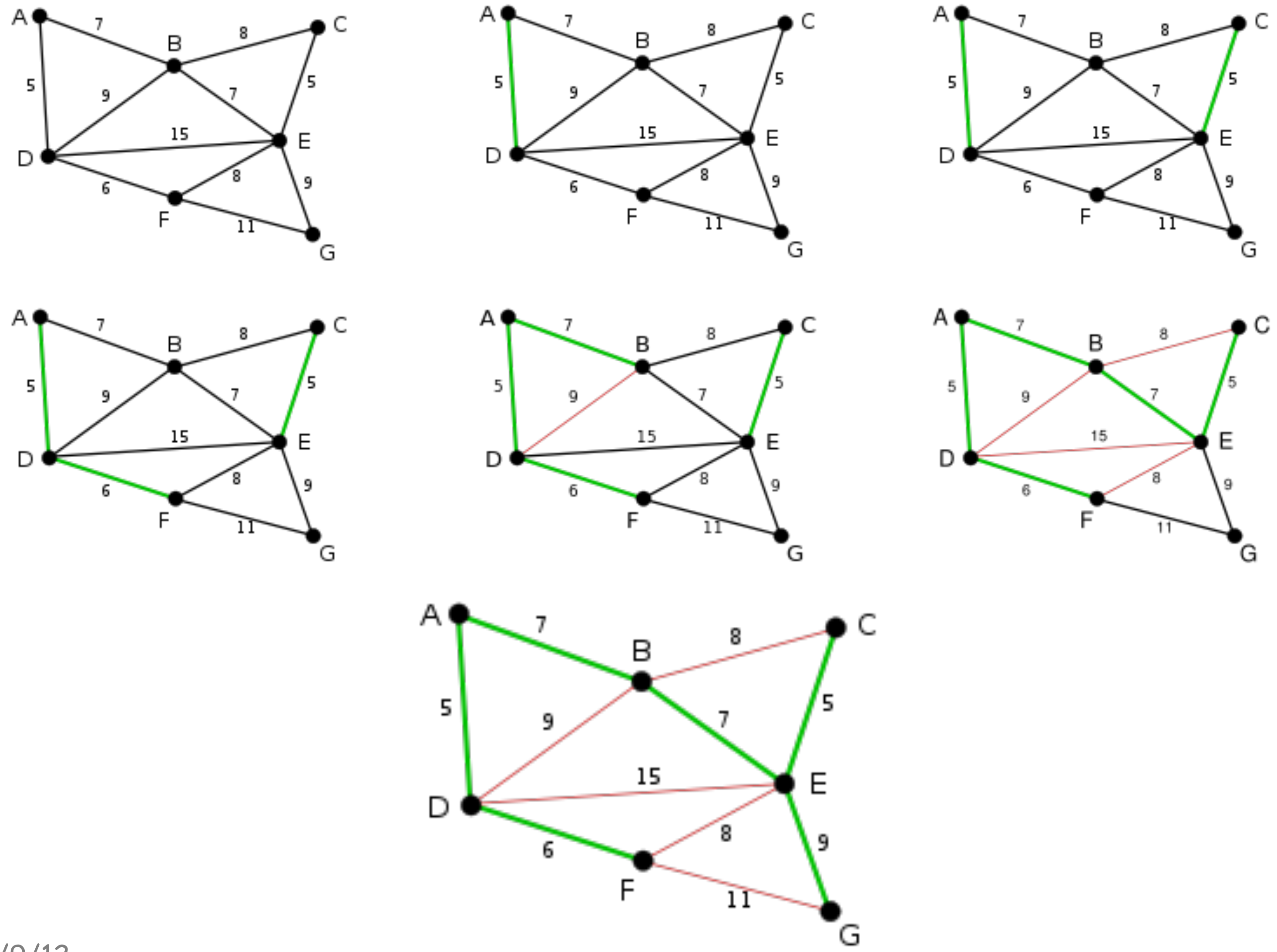
- FindPointer é uma função auxiliar que recebe o nome do conjunto e encontra o representante do conjunto.
- Para isso é criada uma tabela com o nome do conjunto e um ponteiro para um elemento.



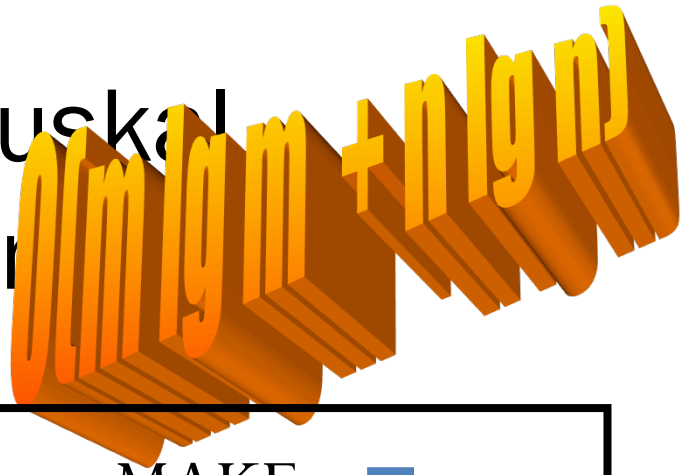
Aplicações

- O algoritmo de Kruskal é um algoritmo em teoria dos grafos que busca uma árvore geradora mínima para um grafo conexo com pesos. Isto significa que ele encontra um subconjunto das arestas que forma uma árvore que inclui todos os vértices, onde o peso total, dado pela soma dos pesos das arestas da árvore, é minimizado. Se o grafo não for conexo, então ele encontra uma floresta geradora mínima (uma árvore geradora mínima para cada componente conexo do grafo).

Exemplo de Kruskal



Algoritmo de Kruskal com União por Tar



Operação	v ≡ w ?	MAKE v ≡ w
Esforço	O(1)	O(lg n) amort.

Ordenação O(m.lg(m))

Inclusão de arco O(n.lg(n))

Não inclusão de arco O(1) até m vezes

$$O(m.lg(m)) + O(n.lg(n)) + O(m)$$

Algoritmo de Kruskal com União por A

Operação	v ≡ w ?	MAKE v ≡ w
Esforço	O(lg n)	O(1)

Ordenação O(m.lg(m))

Inclusão de arco O(n)

Não inclusão de arco O(m.lg(n))

$$O(m.lg(m)) + O(m.lg(n)) + O(n)$$

Exercício: Considere a seguinte sequência de operações:

Inserir (x1) ; ... ; Inserir (x10)

Union (x1,x2) ; Union (x2, x3) ; Union (x3, x4)

Union (x5, x6) ; Union (x7, x8) ; Union (x8, x9)

Union (x2, x6) ; Union (x7, x10)

Find-Set (x2) ; Find-Set (x10)

a) Ilustrar a estrutura de dados resultante usando uma implementação por listas de Union-Find com união por tamanho. Assumir que ao unir dois conjuntos com apenas um elemento cada, aquele com elemento de menor índice se torna representante do conjunto união.

b) Ilustrar novamente a estrutura de dados resultante, porém com uma implementação que use União por altura e compressão de caminho.

Referências

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*. Second edition. The MIT Press, 2001.
- Halim, S, Halim, F. *Competitive Programming*. Lulu, 2010
- Horowitz, E., Sahni, S., Rajasekaran, S. *Computer Algorithms*. Computer Science Press, 1997.
- Weiss, M.K. – *Data Structures and Algorithm Analysis*, Benjamin/ Cummins, 1992.
- en.wikipedia.org/wiki/Disjoint-set_data_structure

dúvidas?