INF 1010
Estruturas de Dados Avançadas

Grafos

árvores geradoras

suponha G = (V, E, p) um grafo conexo

Uma árvore geradora é um *subgrafo acíclico de G* contendo todos os vértices e *com caminhos entre quaisquer 2 vértices*

árvore geradora mínima (MST)

Árvore geradora de custo mínimo

Dado um grafo ponderado G = (V,E,p),
uma *árvore geradora de custo mínimo* para G
é uma árvore tal que:

V é o conjunto de nós da árvore

A soma dos pesos das arestas é minima
(entre as árvores geradoras)

Algoritmo de Kruskal

Algoritmo de Kruskal

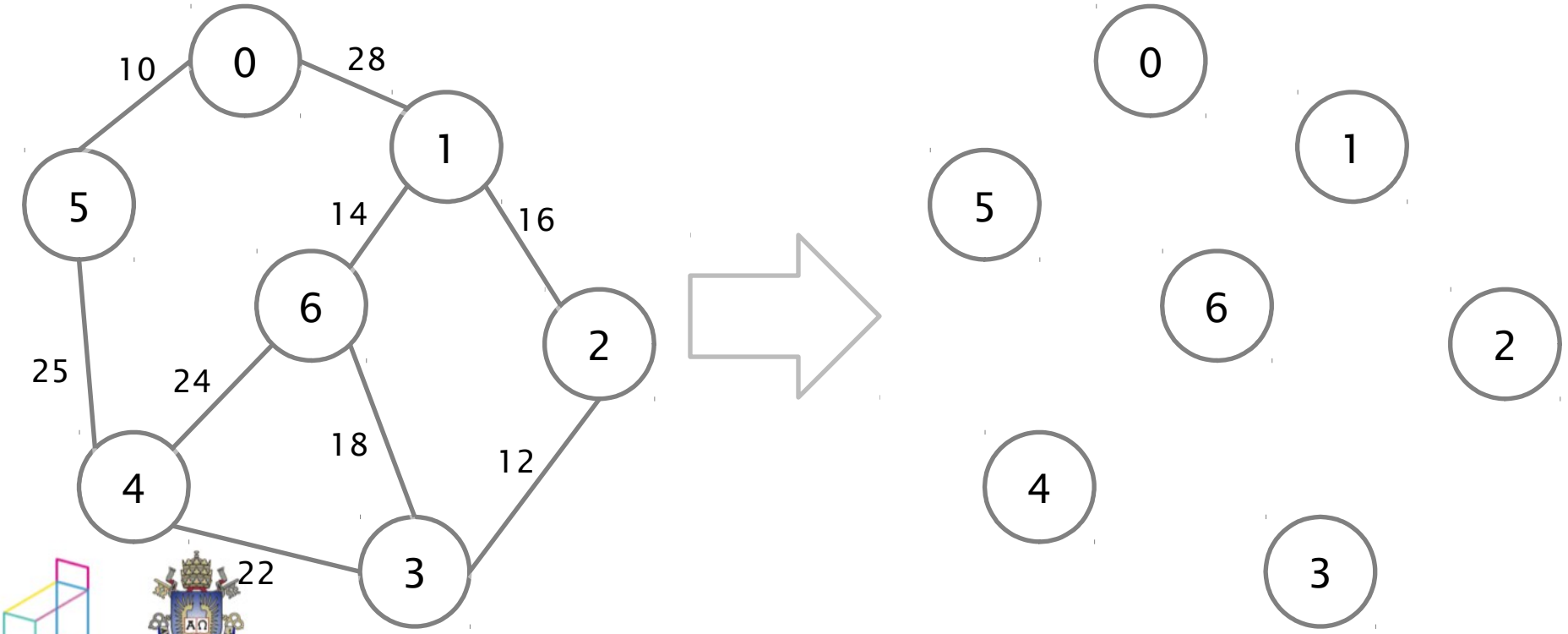
Entrada: Um grafo ponderado G = (V,E,p)

Saída: Árvore geradora de custo mínimo

1. Considere cada nó em V como uma árvore separada (formando uma floresta)
2. Examine a aresta de menor custo. Se ela unir duas árvores na floresta, inclua-a.
3. Repita o Passo (2) até todos os nós estarem conectados.

Algoritmo de Kruskal

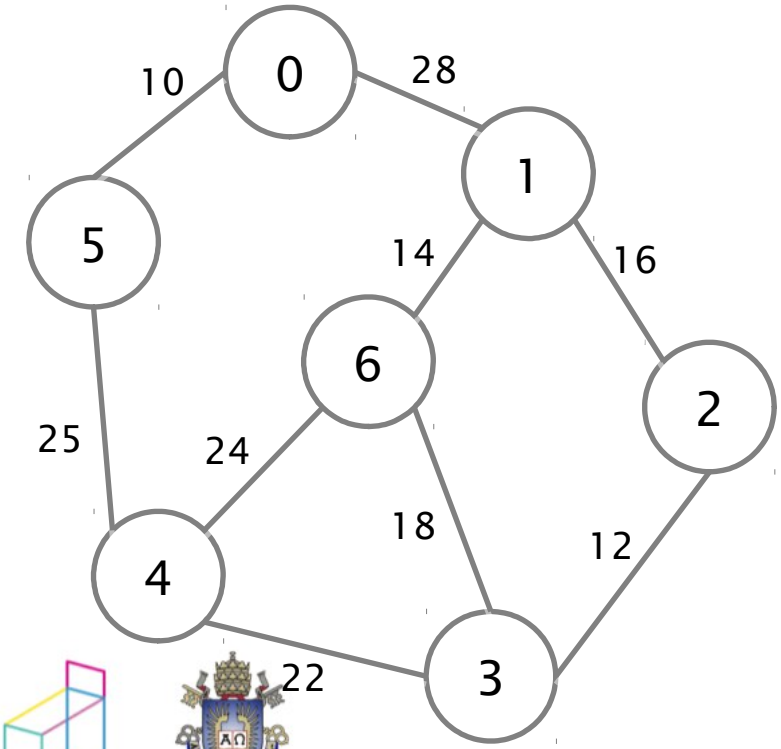
1. Considere cada nó como uma árvore separada (formando uma floresta)

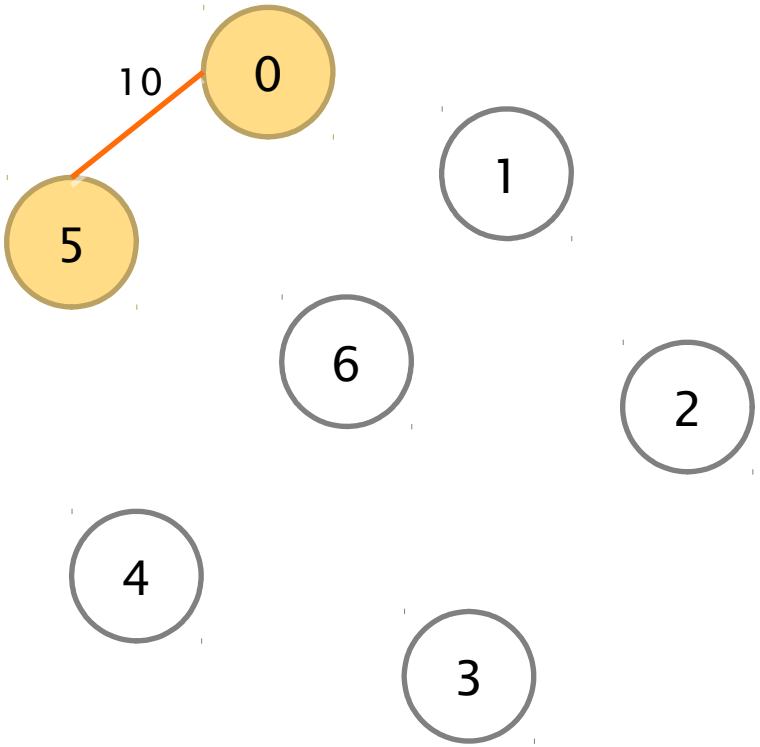


Algoritmo de Kruskal

2. Examine a aresta de menor custo.
Se ela unir duas árvores na floresta, inclua-a.

3. Repita o Passo (2) até todos os nós estarem conectados.

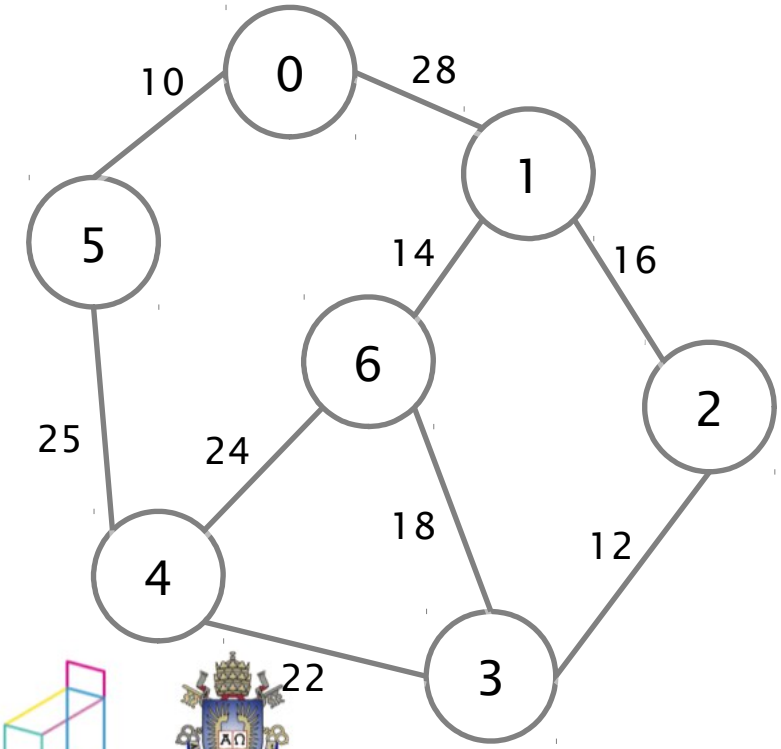


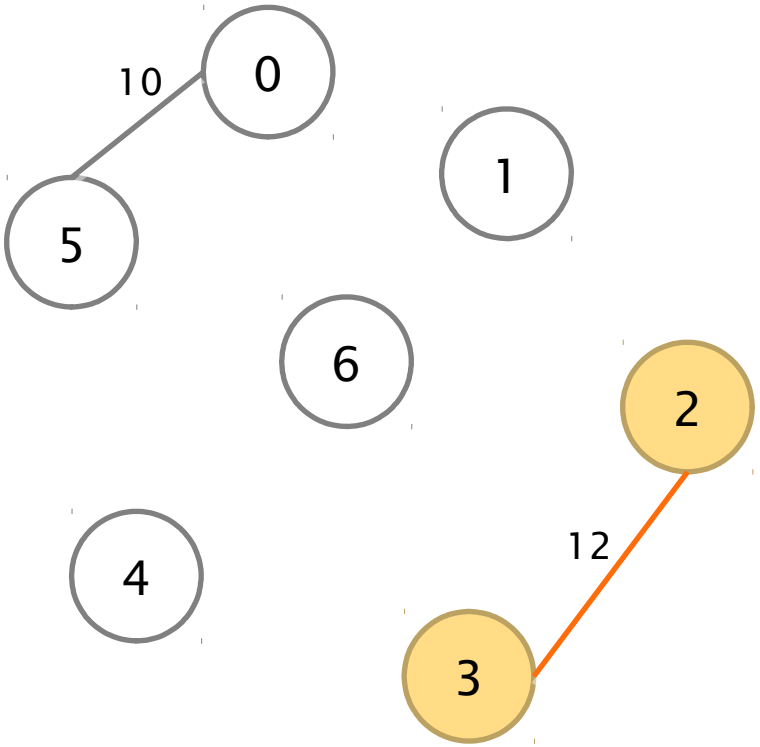


Algoritmo de Kruskal

2. Examine a aresta de menor custo.
Se ela unir duas árvores na floresta, inclua-a.

3. Repita o Passo (2) até todos os nós estarem conectados.

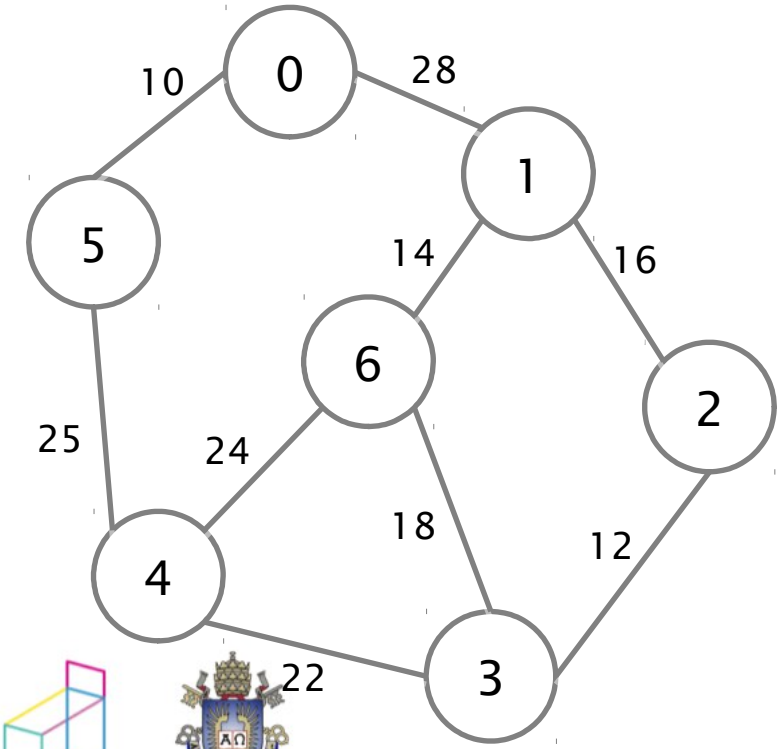


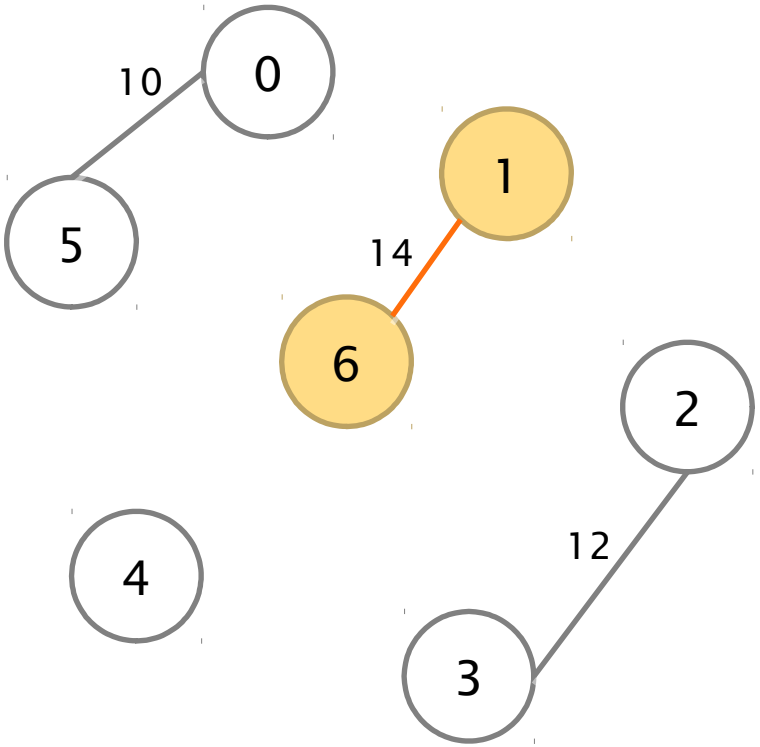


Algoritmo de Kruskal

2. Examine a aresta de menor custo.
Se ela unir duas árvores na floresta, inclua-a.

3. Repita o Passo (2) até todos os nós estarem conectados.

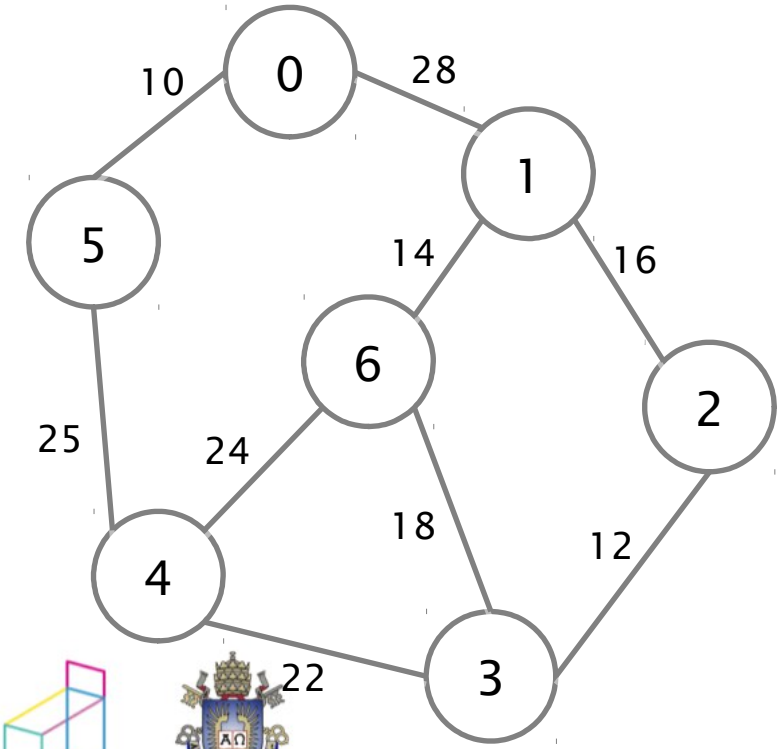


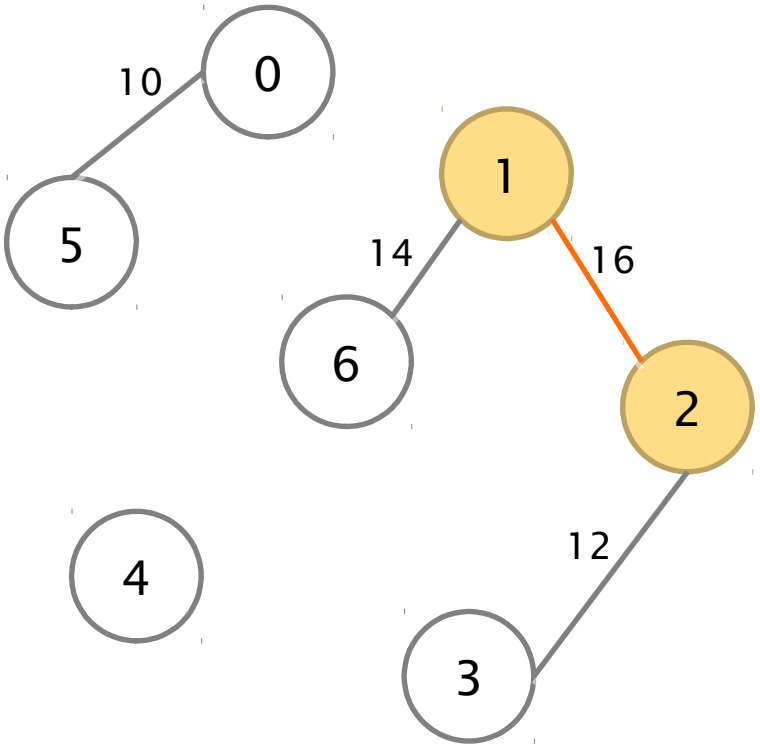


Algoritmo de Kruskal

2. Examine a aresta de menor custo.
Se ela unir duas árvores na floresta, inclua-a.

3. Repita o Passo (2) até todos os nós estarem conectados.

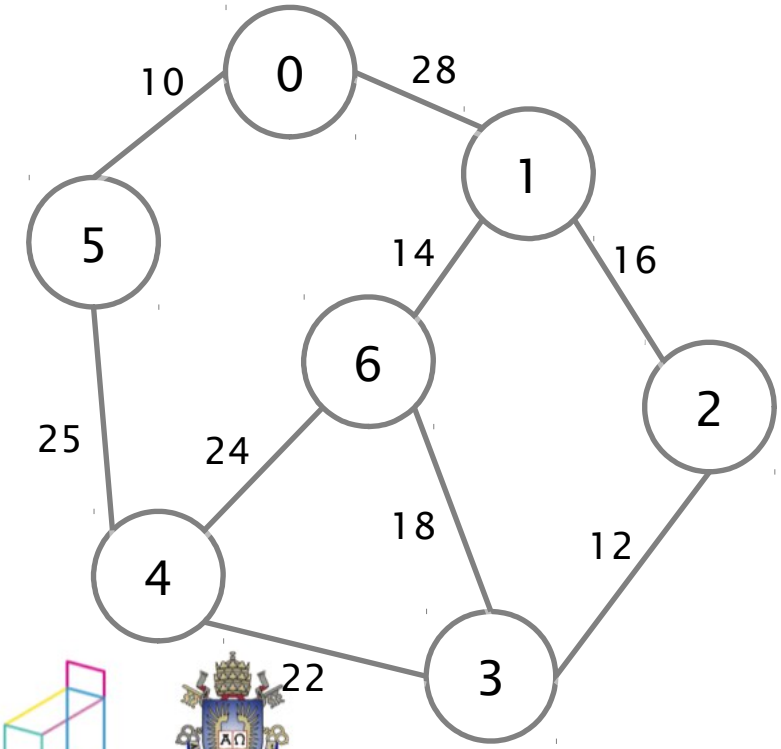


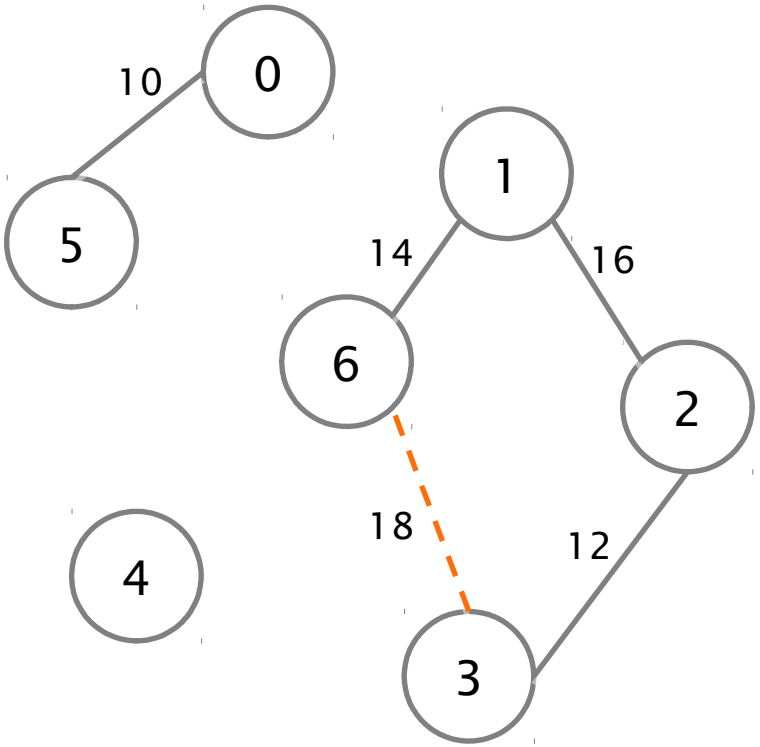


Algoritmo de Kruskal

2. Examine a aresta de menor custo.
Se ela unir duas árvores na floresta, inclua-a.

3. Repita o Passo (2) até todos os nós estarem conectados.

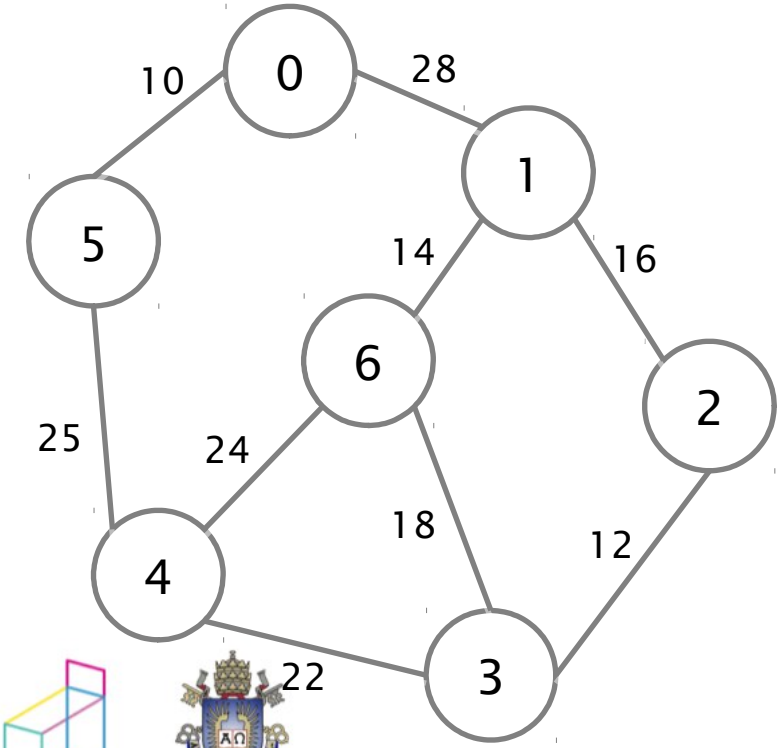


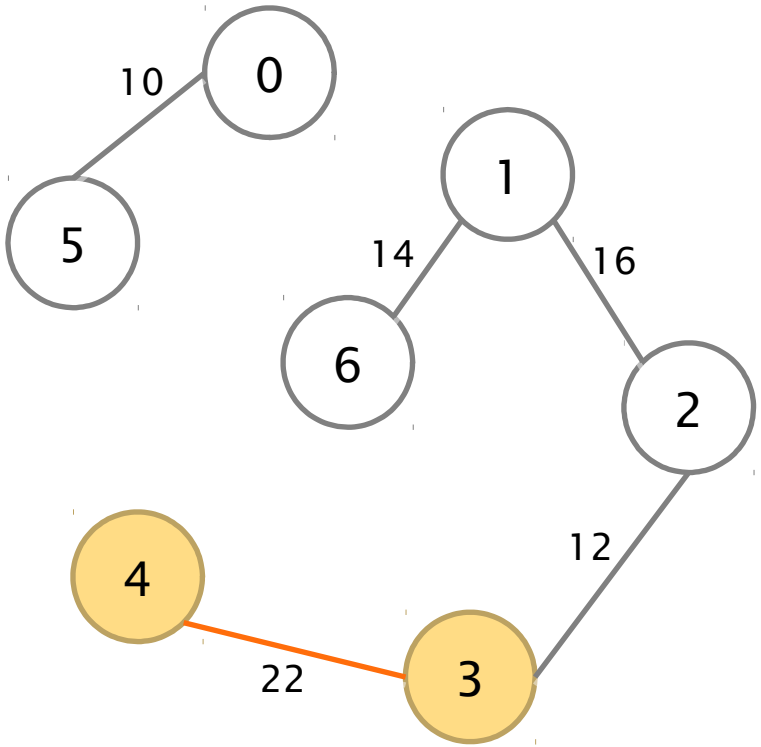


Algoritmo de Kruskal

2. Examine a aresta de menor custo.
Se ela unir duas árvores na floresta, inclua-a.

3. Repita o Passo (2) até todos os nós estarem conectados.

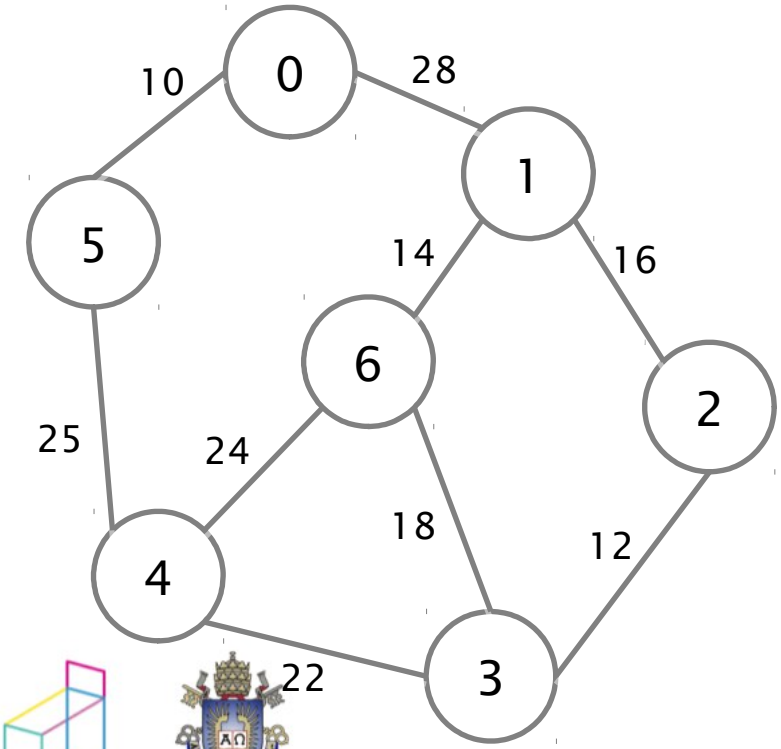


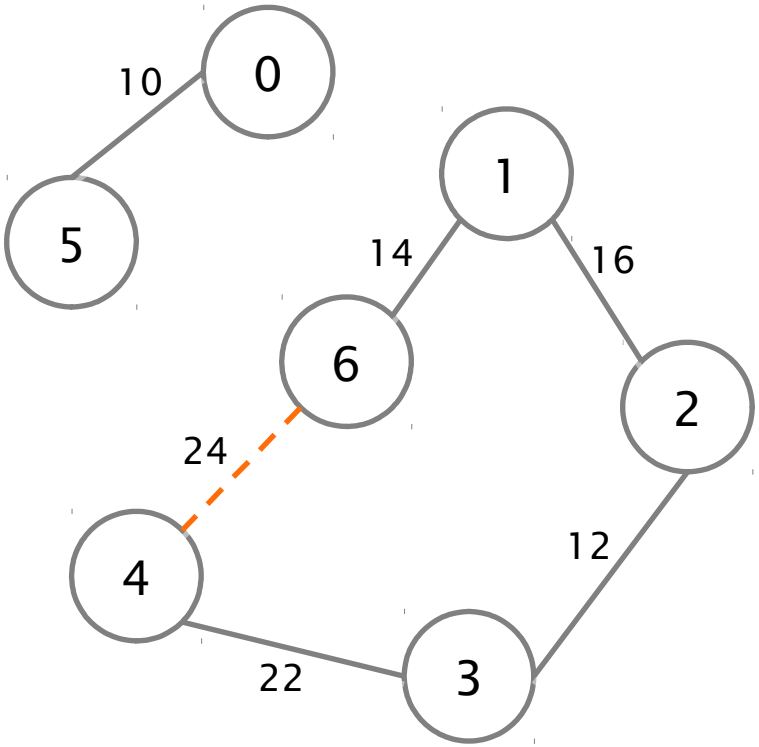


Algoritmo de Kruskal

2. Examine a aresta de menor custo.
Se ela unir duas árvores na floresta, inclua-a.

3. Repita o Passo (2) até todos os nós estarem conectados.

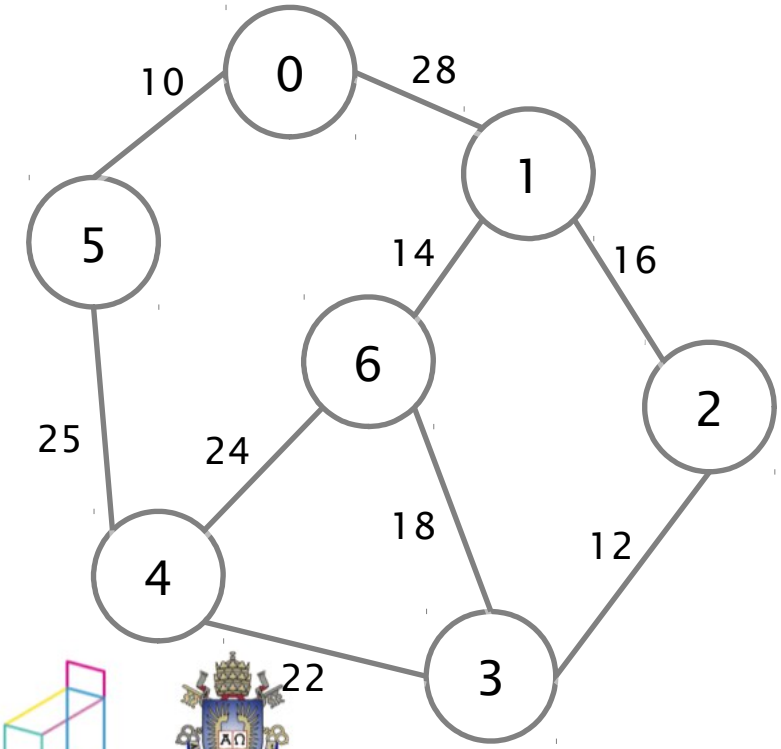


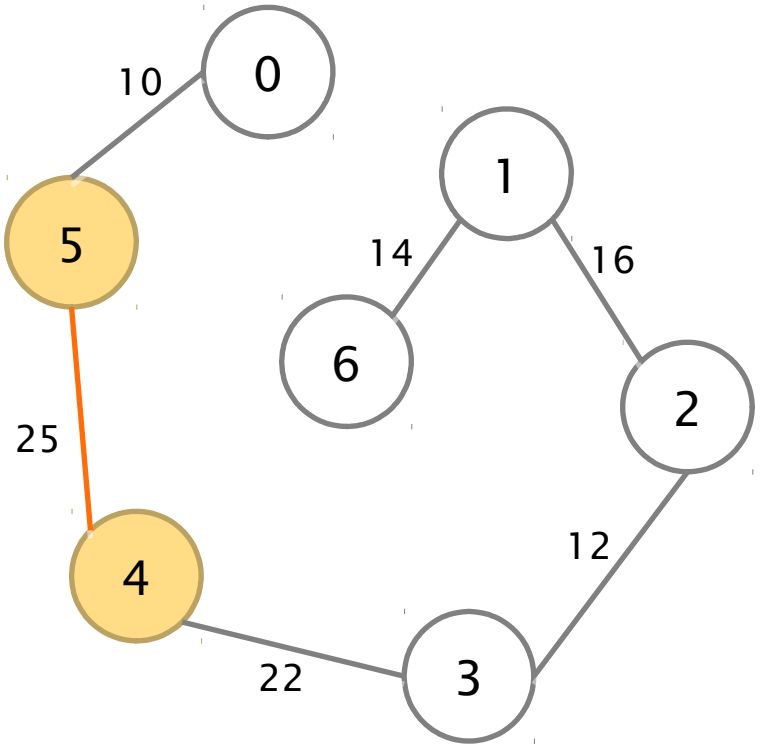


Algoritmo de Kruskal

2. Examine a aresta de menor custo.
Se ela unir duas árvores na floresta, inclua-a.

3. Repita o Passo (2) até todos os nós estarem conectados.





implementação

- grafo continua representado como antes
- heap de prioridades para arestas
  - minheap para remover aresta de menor custo

mas como descobrir se a aresta de menor custo une 2 árvores distintas?

como descobrir se uma aresta “conecta” duas árvores?

- uso de uma estrutura de dados específica chamada união e busca
  - estrutura de dados para manipulação de *partições* de conjuntos

Definições

Universo: $U = \{x_1, x_2, \ldots, x_n\}$

Uma *partição* é uma coleção : $C = \{S_1, \ldots, S_k\}$ de conjuntos disjuntos

$S_i \subseteq U$ para qualquer i

$S_1 \cup \ldots \cup S_k = U$ (cobertura)

$S_i \cap S_j = \varnothing$, para $i \neq j$ (disjunção)

partição S_i identificada por
um de seus elementos $x \in S_i$ (REPRESENTANTE)

Operações Básicas

cria():

cria uma partição do conjunto original

busca(x):

Informa qual partição um elemento faz parte, retornando um elemento que representa a partição

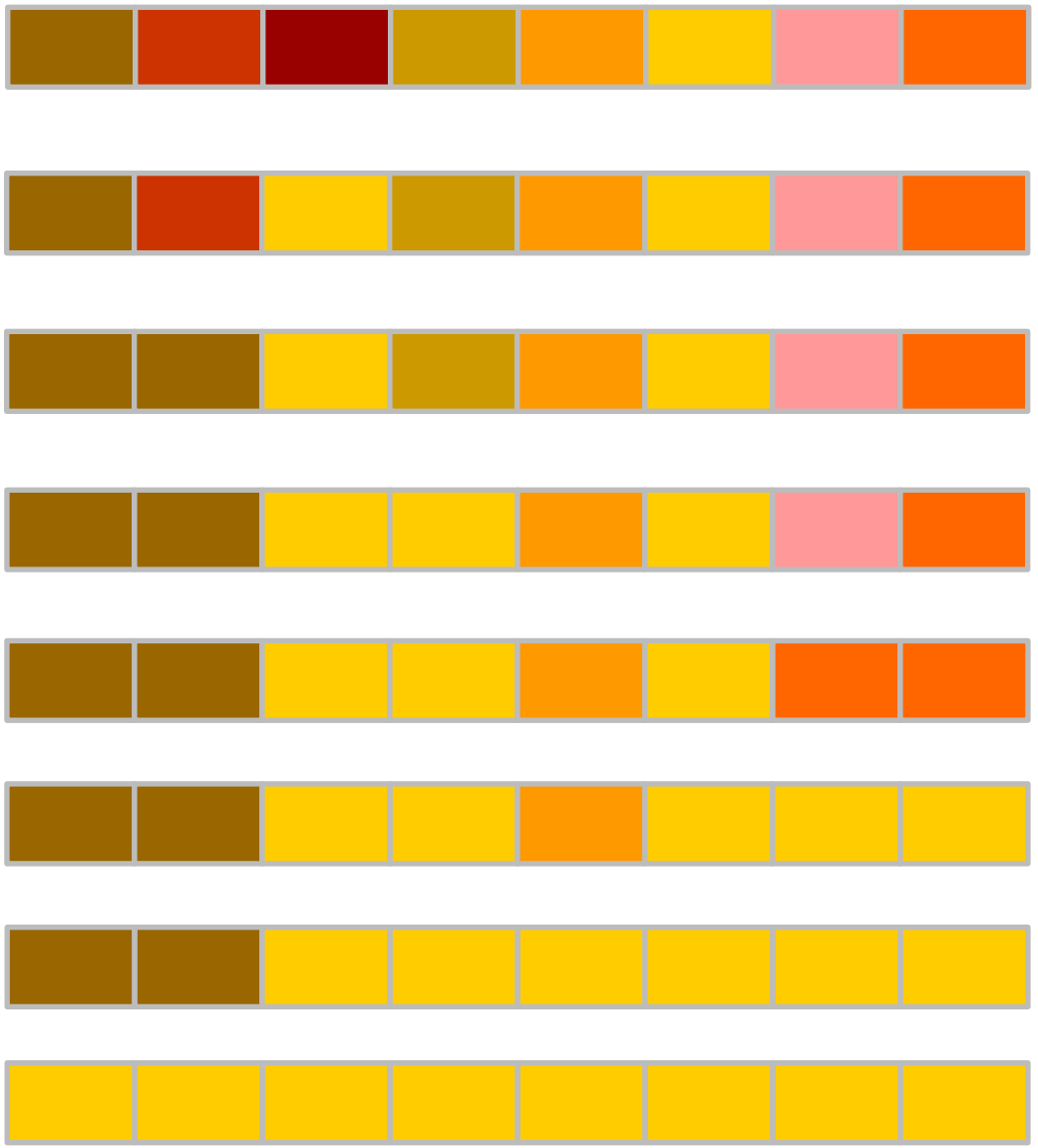
Útil para determinar se dois elementos estão no mesmo conjunto

{k,j,n,o,y,**w**,x,z} -> w

união(x,y):

combina dois conjuntos em um único, substituindo S_i e S_j por um novo conjunto S_k tal que $S_i \cup S_j = S_k$

unindo partes

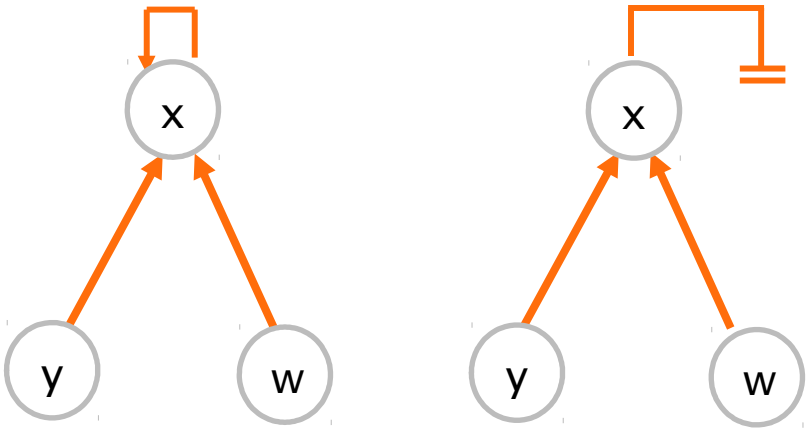


Representação de estruturas de união e busca

Representação por árvores reversas

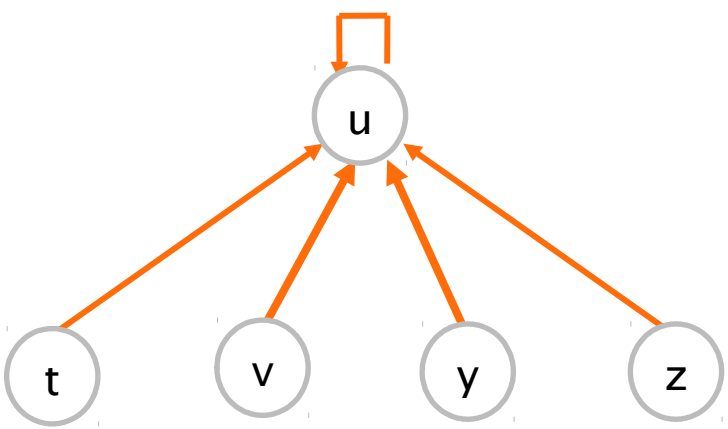
Árvores reversas (Reversed Trees)

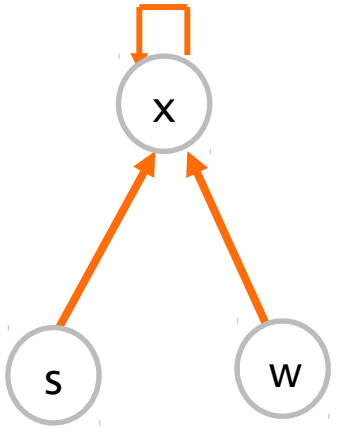
- Cada nó aponta para o seu pai
  - a raiz aponta para si mesma ou aponta para NULL
- Depois de um nó deixar de ser a raiz, um nó nunca pode se tornar uma raiz novamente



Representação por Árvores Reversas

Exemplo





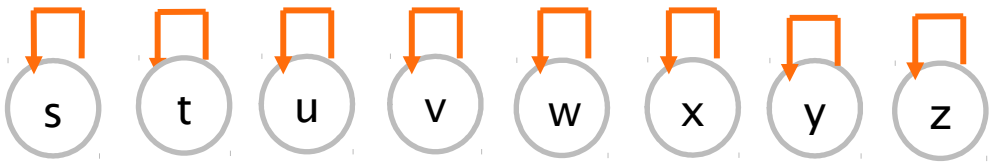
$S_1 = \{t,\underline{u},v,y,z\}$

$S_2 = \{s,w,\underline{x}\}$

Operações

cria(n) = $\{S_0, \ldots, S_{n-1}\}$

cria uma estrutura uniao-e-busca com n elementos disjuntos

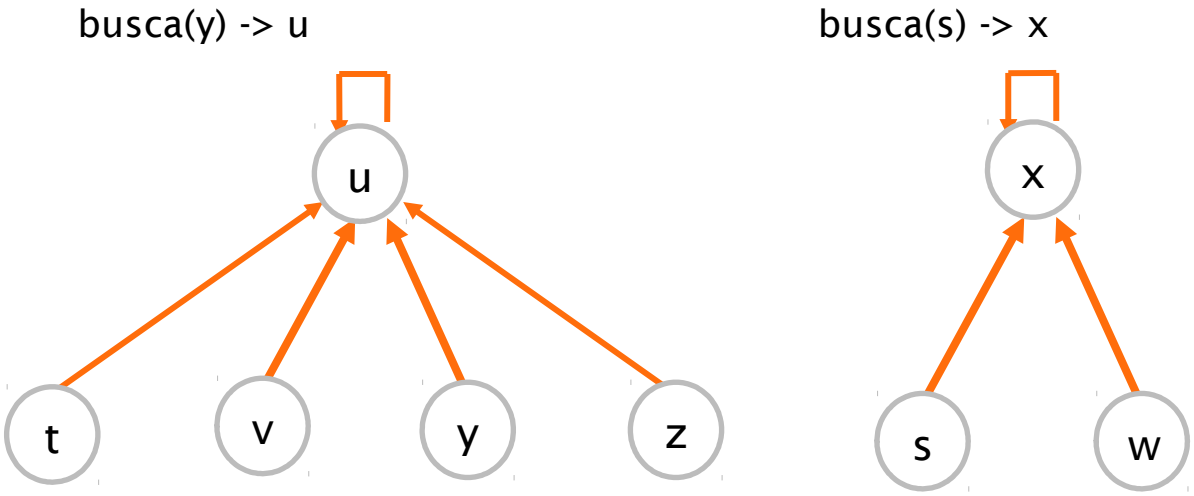


Operações

busca(ub, x)

retorna o elemento que representa o conjunto

busca a raiz da árvore que contém o elemento



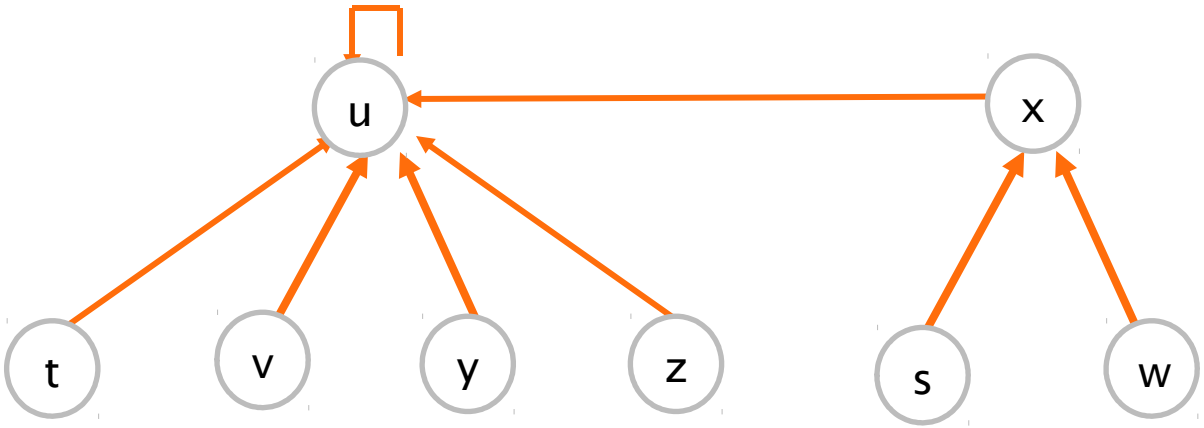
Operações

união(ub, u, x)

une dois grupos, fazendo com que a raiz de um grupo aponte para a raiz do outro grupo

torna uma árvore uma sub-árvore da outra

união(ub, u, x) = {t,**u**,v,y,z, s,w,x}



À medida que novas uniões são realizadas, uma árvore completamente degenerada pode ser criada, fazendo com que as operações de Find se realizem em tempo linear O(n)

Representação por Vetor

Representação por Vetor

Implementação de partições:

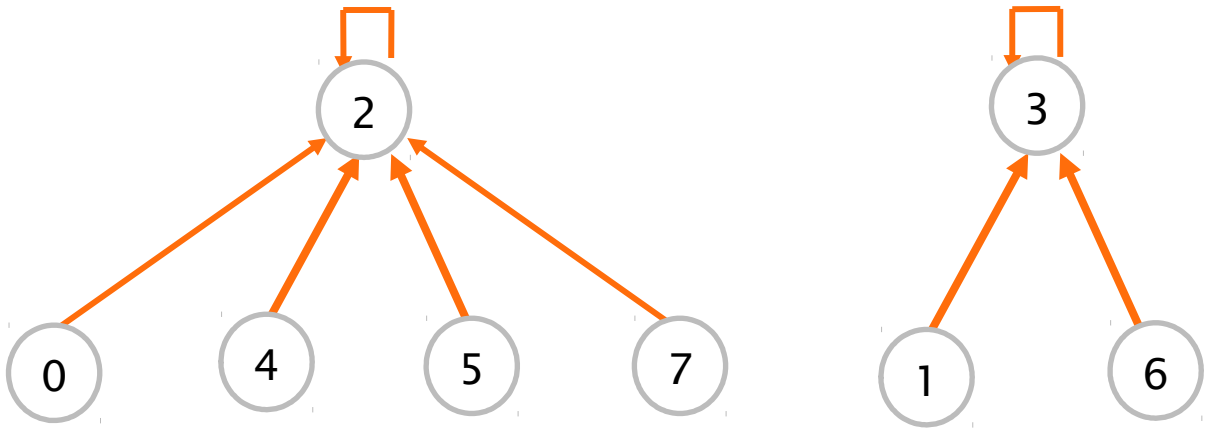
- Através de um vetor
- Cada elemento do vetor representa um elemento do universo
- Cada elemento do vetor aponta para seu pai

Representação por Vetor

Exemplo

(Note que a raiz possui -1 como valor do ponteiro)

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



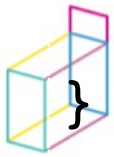
União e Busca para uso no lab 11

```
typedef struct suniaoBusca UniaoBusca;

UniaoBusca* ub_cria(int tam);
/* cria particao de conjunto com tam elementos */
/* cada elemento está inicialmente em parte separada */

int ub_busca (UniaoBusca* ub, int u);
/* retorna o representante da parte em que está u */

int ub_uniao (UniaoBusca* ub, int u, int v);
/* retorna o representante do resultado */

void ub_libera (UniaoBusca* ub);
/* libera a estrutura */

void debug (UniaoBusca *ub);
```

idéia da fç arvoreCustoMinimo

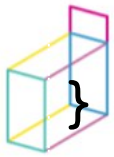
Union-Find p/ Kruskal (implementação simplificada - ineficiente!!!)

```
typedef struct uniaobusca UniaoBusca;

struct uniaoBusca {
  int n; int *v;
};

UniaoBusca* cria(int size) {
   /* aloca novo item com tamanho dado e
      preenche vetor com -1 */
}

int uniao (UniaoBusca* ub, int u, int v) {
  v = busca (ub, v); /* acha raiz de árvore de v *
  u = busca (ub, u);
  ub->v[u] = v;
  return v;
}
```



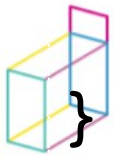
Union-Find p/ Kruskal (implementação simplificada - ineficiente!!!)

```
typedef struct uniaobusca UniaoBusca;

struct uniaoBusca {
  int n; int *v;
};

UniaoBusca* cria(int size) {
   /* aloca novo item com tamanho dado e
      preenche vetor com -1 */
}

int uniao (UniaoBusca* ub, int u, int v) {
  v = busca (ub, v); /* acha raiz de árvore de v *
  u = busca (ub, u);
  ub->v[u] = v;
  return v;
}
```

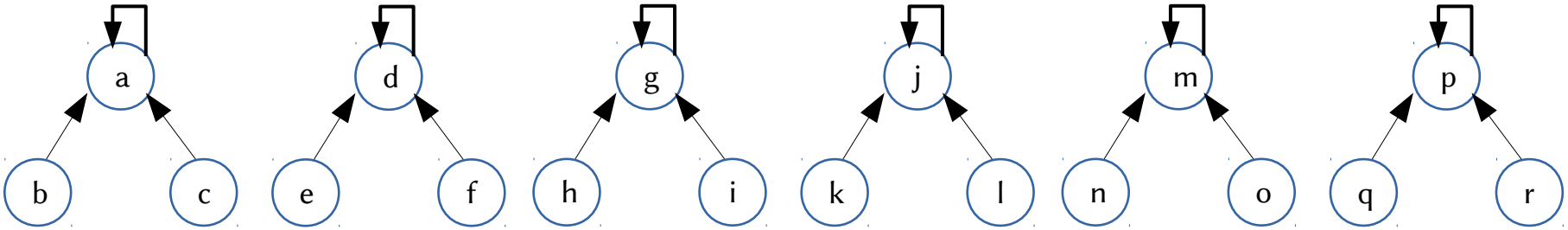


Union-Find p/ Kruskal (implementação simplificada - ineficiente!!!)

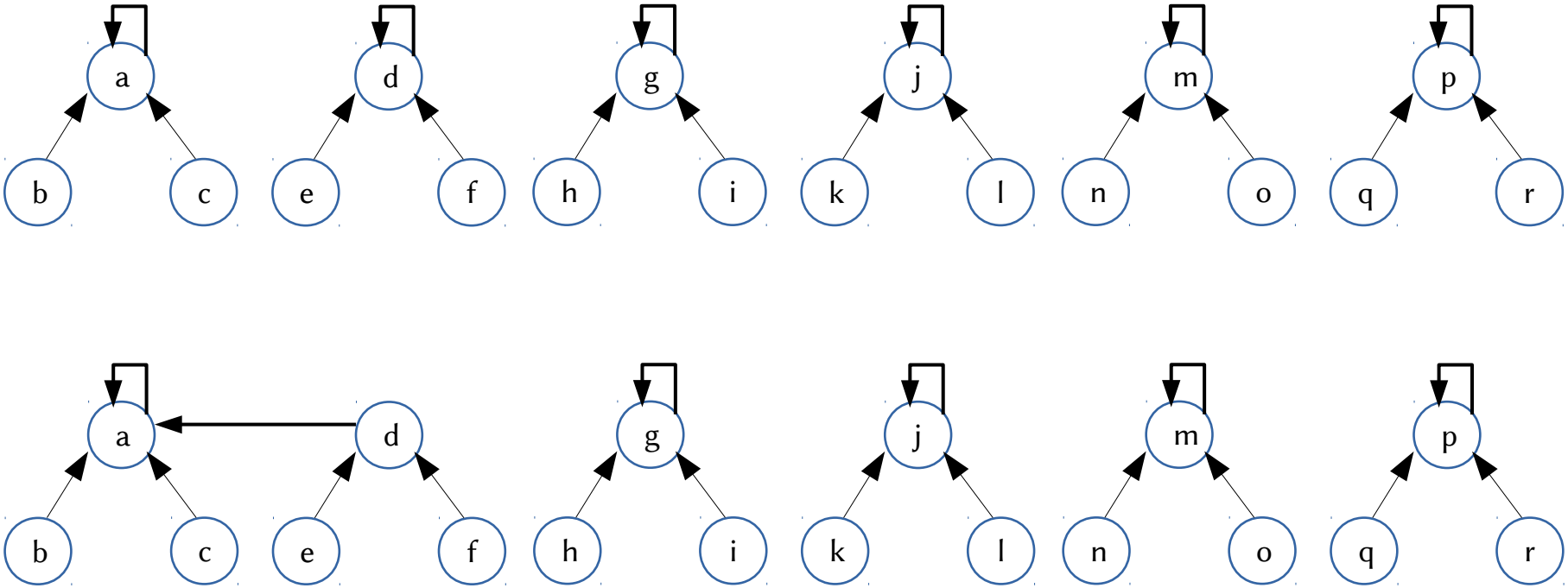
```
typedef struct uniaobusca UniaoBusca;

struct uniaoBusca {int n; int *v;};

UniaoBusca* cria(int size) {
    /* aloca novo item com tamanho dado e
       preenche vetor com -1 */
}
int busca (UniaoBusca* ub, int u){
  while (ub->v[u] >= 0) u = ub->v[u];
  return u;
}
int uniao (UniaoBusca* ub, int u, int v) {
}
```



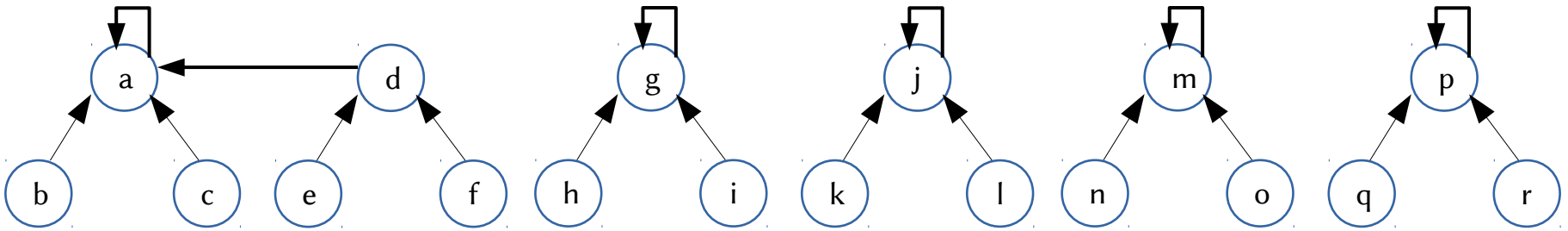
União

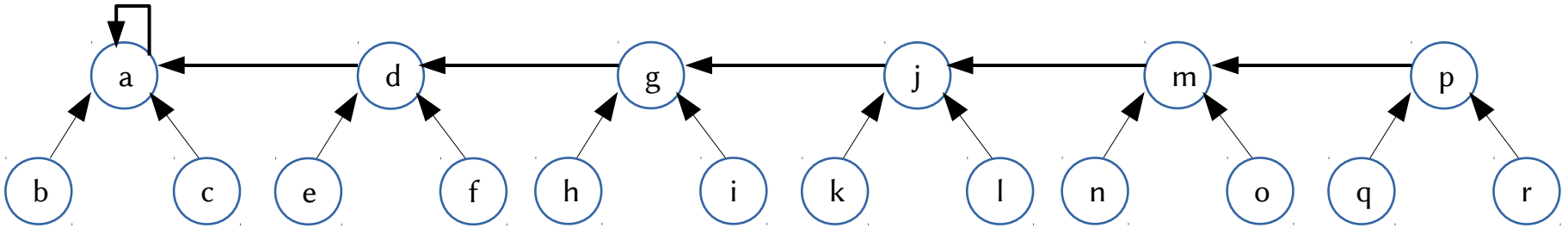


União



União





À medida que novas uniões são realizadas, uma árvore completamente degenerada pode ser criada, fazendo com que as operações de Find se realizem em tempo linear O(n)

Problema

Configuração inicial:

uma floresta com n nós (singletons)

Operações;

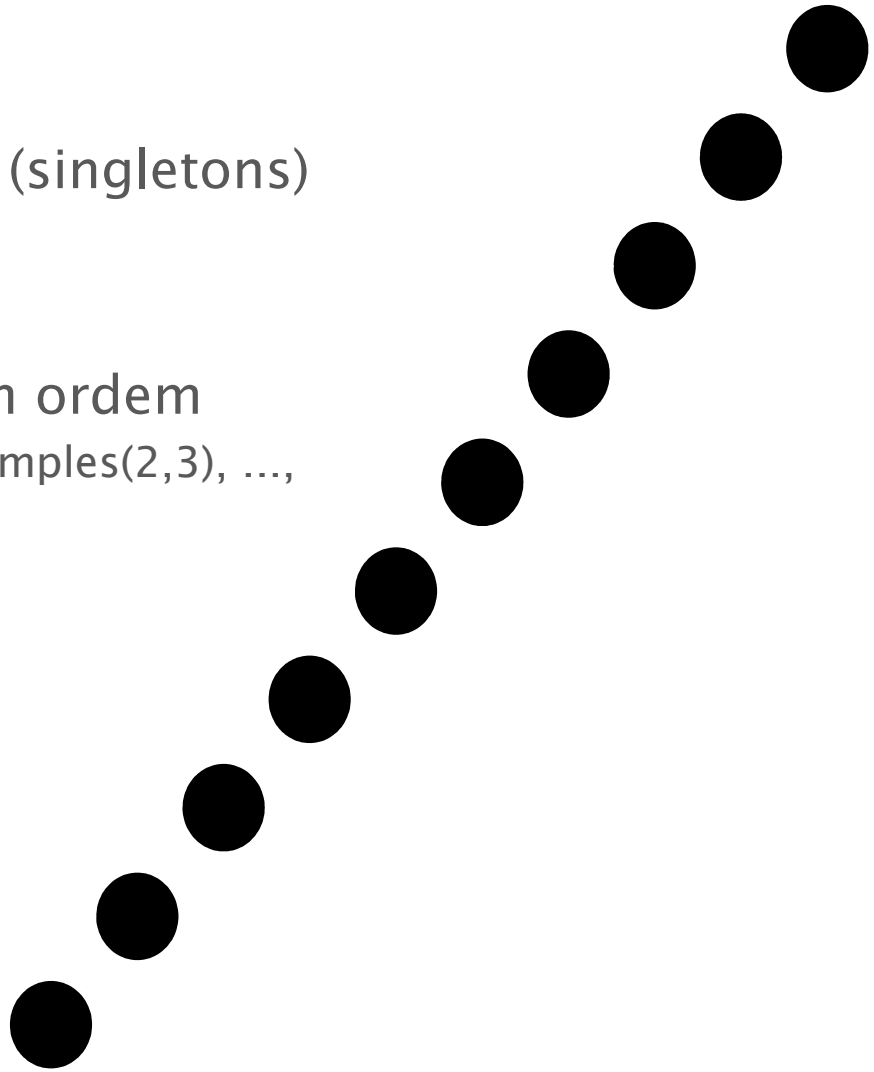
União dos elementos em ordem

uniaoSimples(1,2), uniaoSimples(2,3), ..., uniaoSimples(n-1, n)

Resultado:

uma árvore degenerada

busca muito cara!!!



otimizações

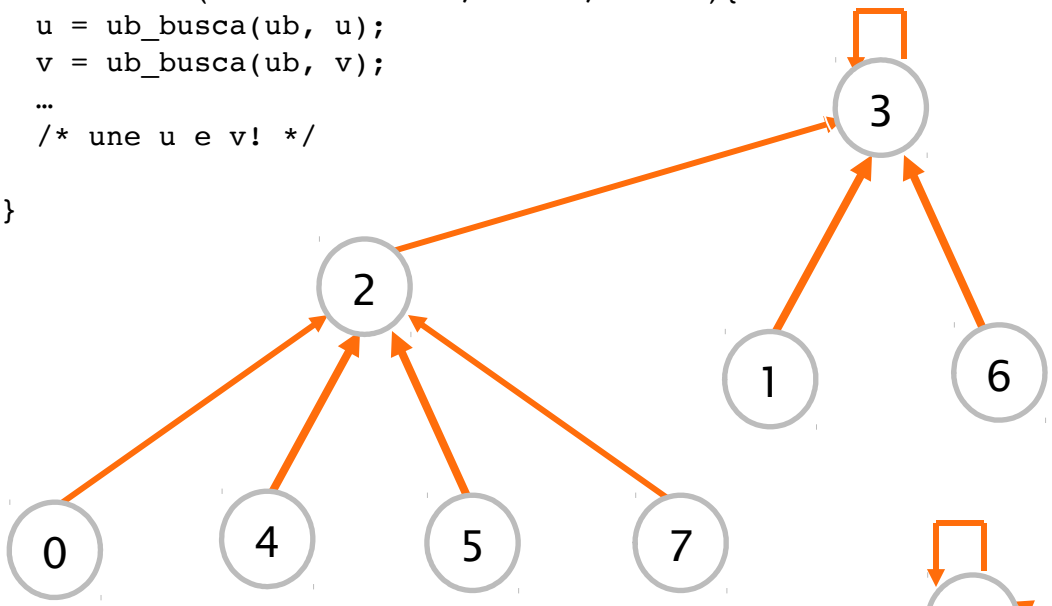
otimizações

- na união busca raizes

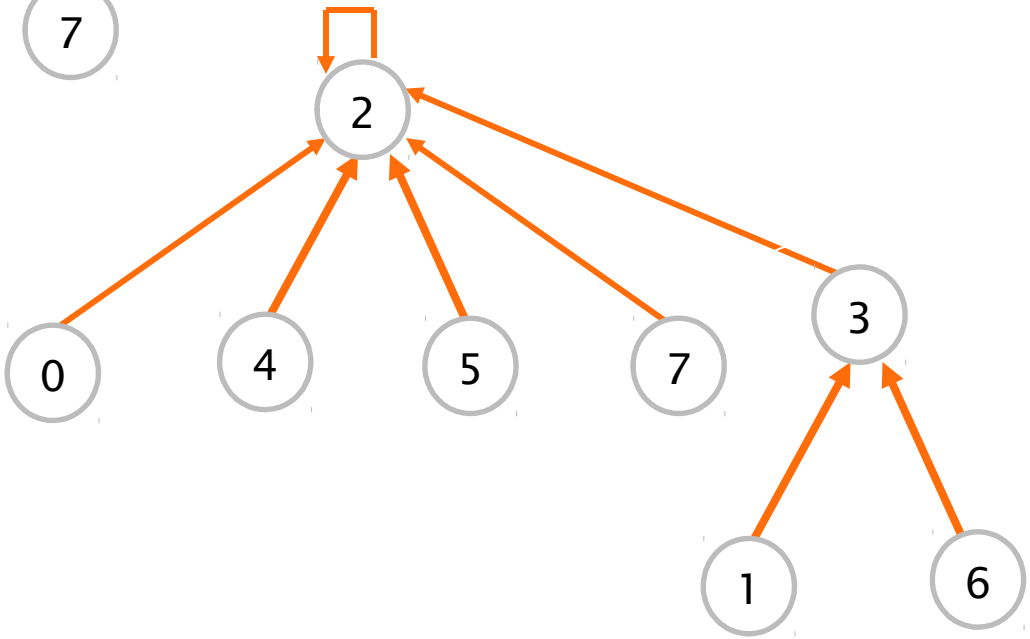
```
void uniao (UniaoBusca* ub, int u, int v){
  u = busca(ub, u);
  v = busca(ub, v);
  …
  /* une u e v! */

}
```

união nas raizes

```
void uniao (UniaoBusca* ub, int u, int v){
  u = ub_busca(ub, u);
  v = ub_busca(ub, v);
  …
  /* une u e v! */

}
```



ou



união nas raizes

- pendurar árvore com menor número de nós

```
void uniao (UniaoBusca* ub, int u, int v){
  u = busca(ub, u);
  v = busca(ub, v);
  if (u==v) return;
  /* uniao com base no numero de nós */
  if /* arvore em u tem menos nós */ {
    ub->v[u] = v;
    /* atualiza o numero de nós de v */
  }
  else /* arvore em v tem menos nós */
}
```

- como manter o número de nós em cada árvore

união nas raizes

- pendurar árvore com menor número de nós

```
void uniao (UniaoBusca* ub, int u, int v){
  u = busca(ub, u);
  v = busca(ub, v);
  if (u==v) return;
  /* uniao com base no numero de nós */
  if /* arvore em u tem menos nós */ {
    ub->v[u] = v;
    /* atualiza o numero de nós de v */
  }
  else /* arvore em v tem menos nós */
}
```

- como manter o número de nós em cada árvore

  podemos usar o próprio campo que indica o pai e em vez de -1 armazenar -(número de nós) em cada raiz

união nas raizes

- ”pendurar” árvore com menor número de nós na outra

```
void uniao (UniaoBusca* ub, int u, int v){
  u = busca(ub, u);
  v = busca(ub, v);
  if (u==v) return;
  /* uniao com base no numero de nós */
  if (ub->v[u] < ub->v[v]) {
    ub->v[v] += ub->v[u];
    ub->v[u] = v;
    return v;
  }
  else /* arvore em v tem menos nós */
}
```

otimizações

- na união busca raizes
  - ☹ mas agora união fica cara
  - como melhorar isso?
  -
-
- na busca podemos “aproveitar” para pendurar nós intermediários diretamente na raiz

otimizações

```
typedef struct uniaobusca UniaoBusca;

struct uniaoBusca {int n; int *v;};

int busca (UniaoBusca* ub, int u){
  while (ub->v[u] >= 0) u = ub->v[u];
  return u;
}
```

otimizações

```
int busca (UniaoBusca* ub, int u){
  int x = u;
  int aux;
  if ((u < 0) || (u > ub->n)) return -1;
  while (ub->v[u] >= 0) u = ub->v[u];
  while (ub->v[x] >= 0) {
    aux = x;
    x = ub->v[x];
    ub->v[aux] = u;
  }
  return u;
}
```