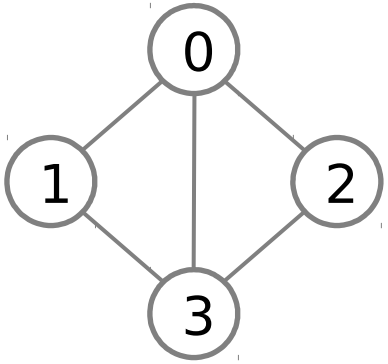
Estruturas de Dados Avançadas (INF1010)

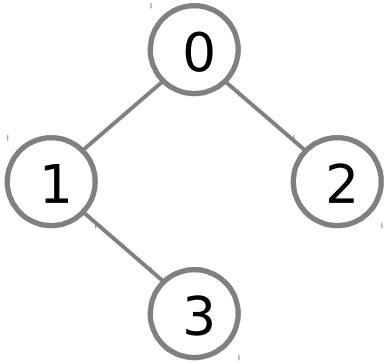
Grafos: árvore geradora mínima
estruturas de união e busca

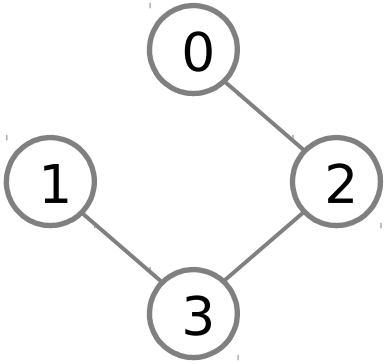
Árvore Geradora de Custo Mínimo

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

- árvore geradora de G: subgrafo **acíclico** com todos os vértices de G e caminhos entre quaisquer dois vértices
- árvore geradora de custo mínimo: a soma dos pesos das arestas é mínima

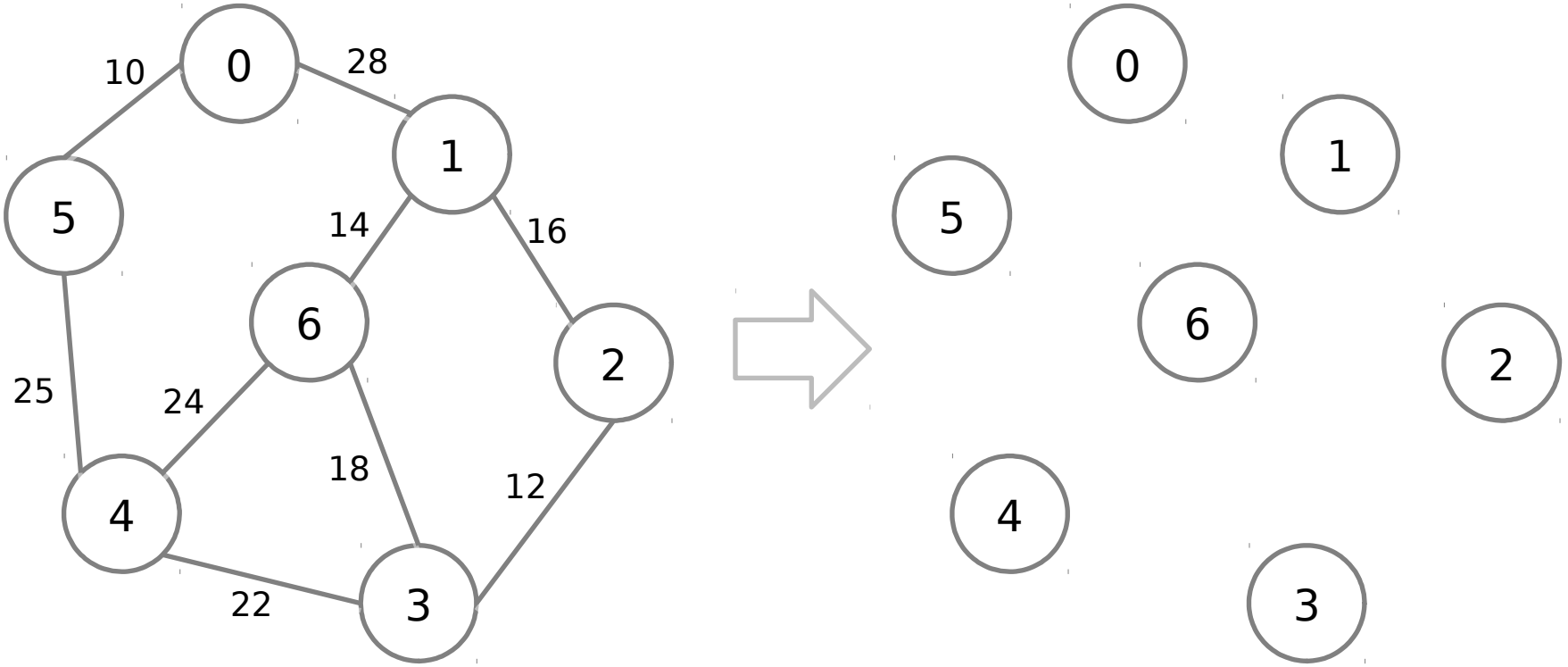






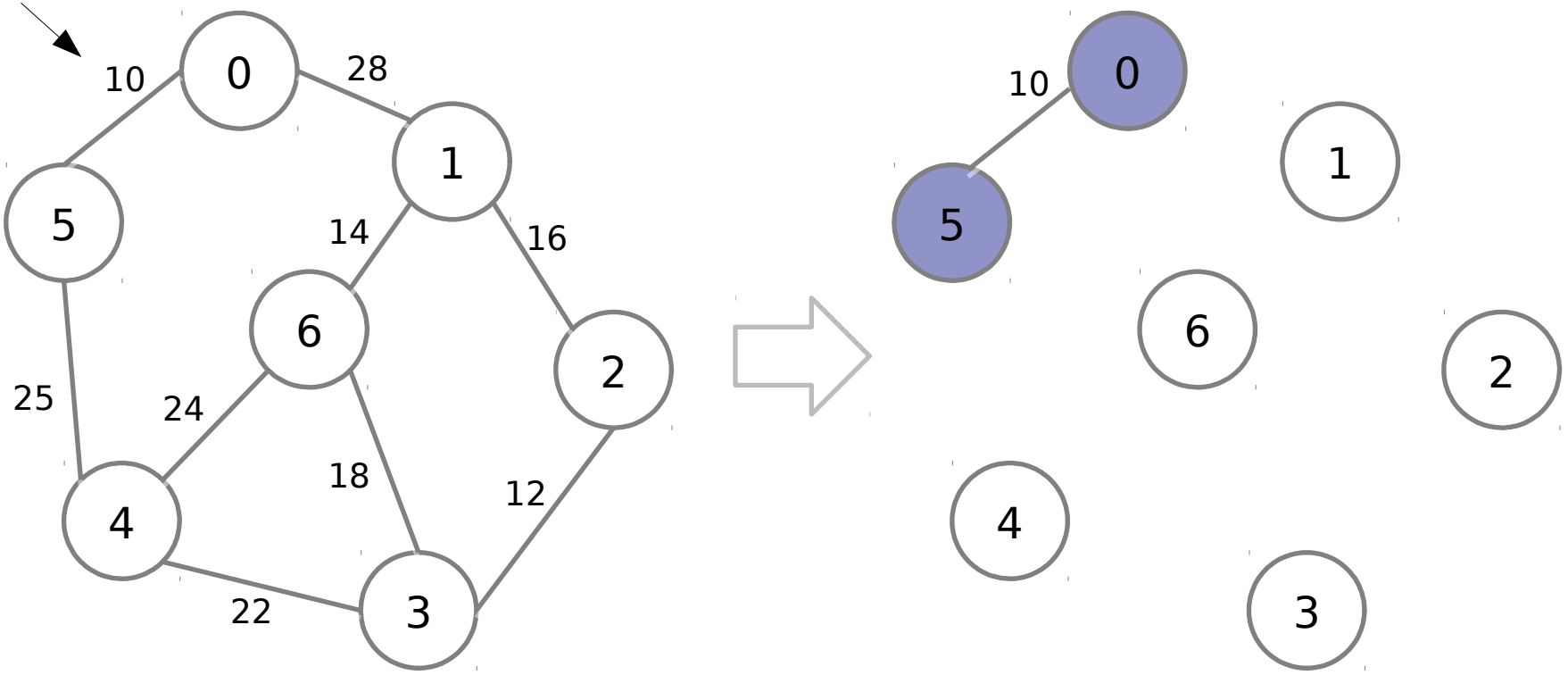
Algoritmo de Kruskal

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



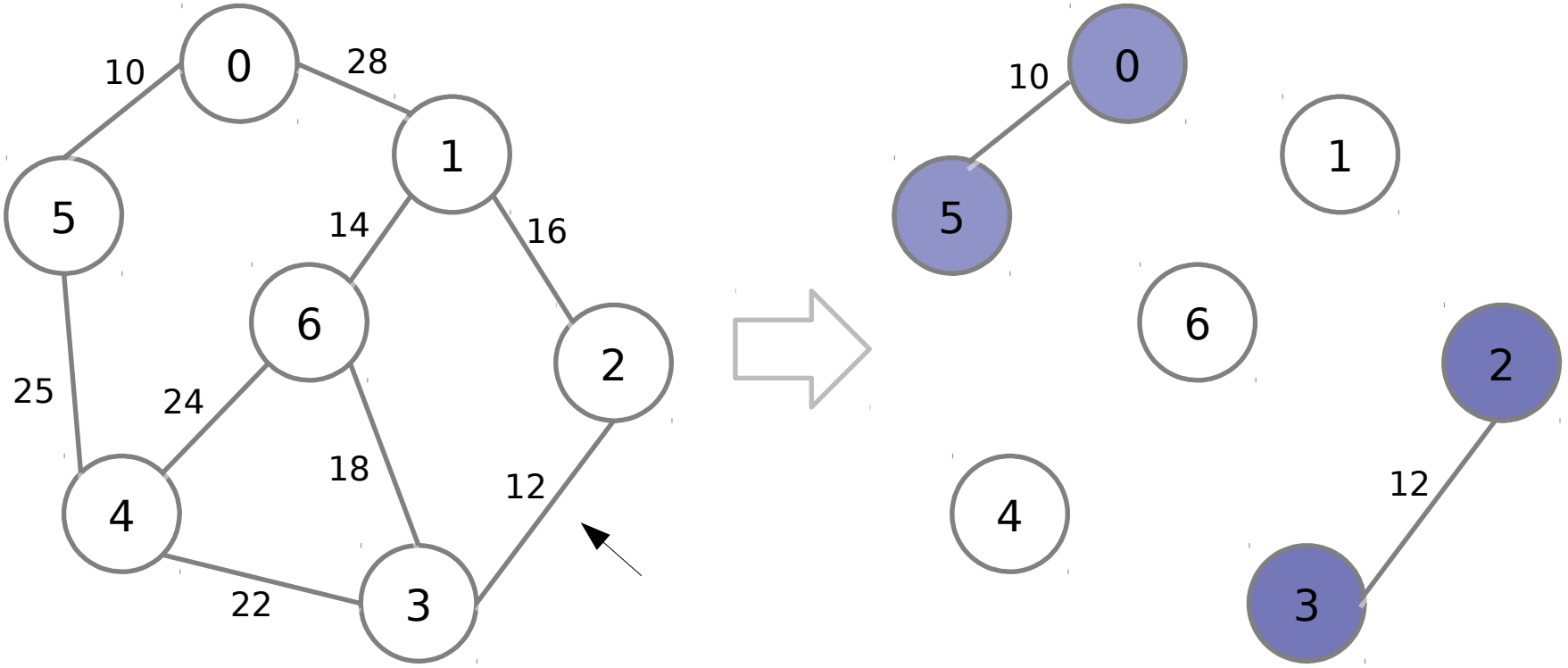
Algoritmo de Kruskal

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



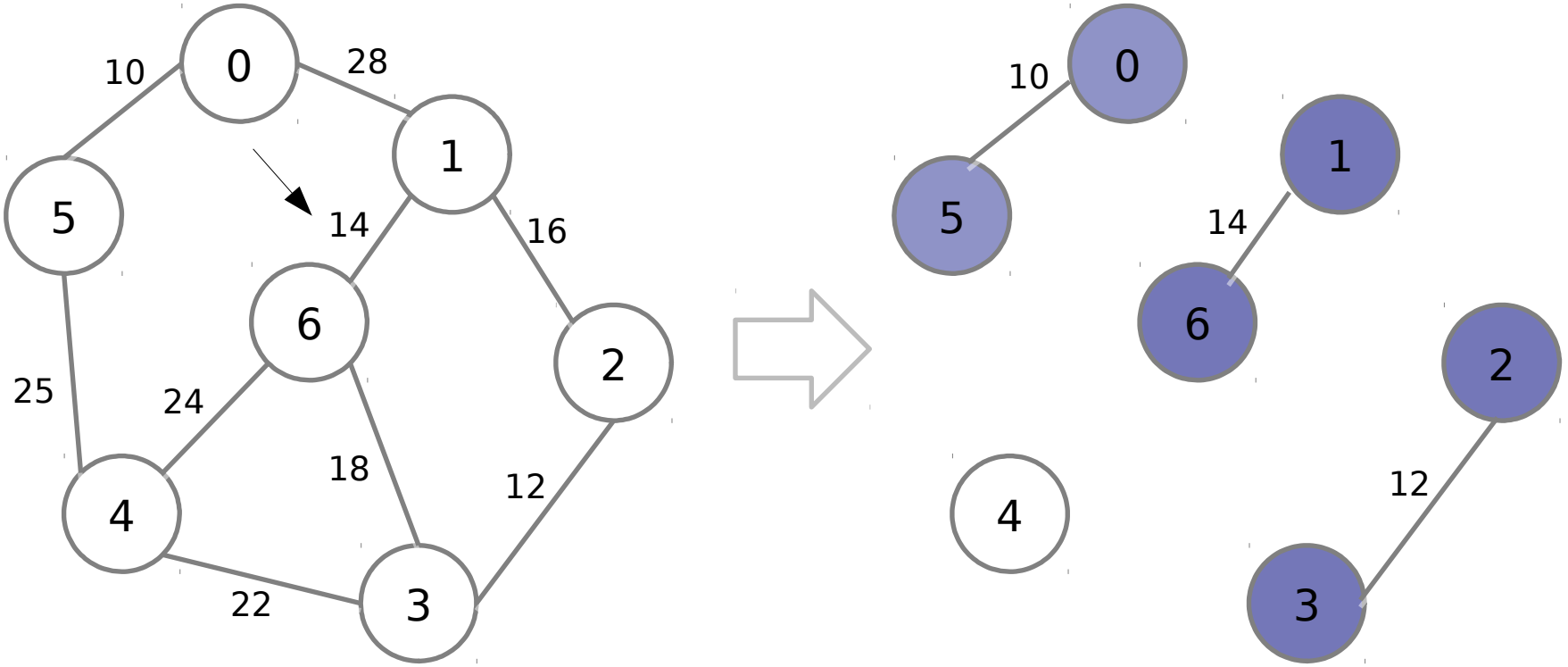
Algoritmo de Kruskal

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



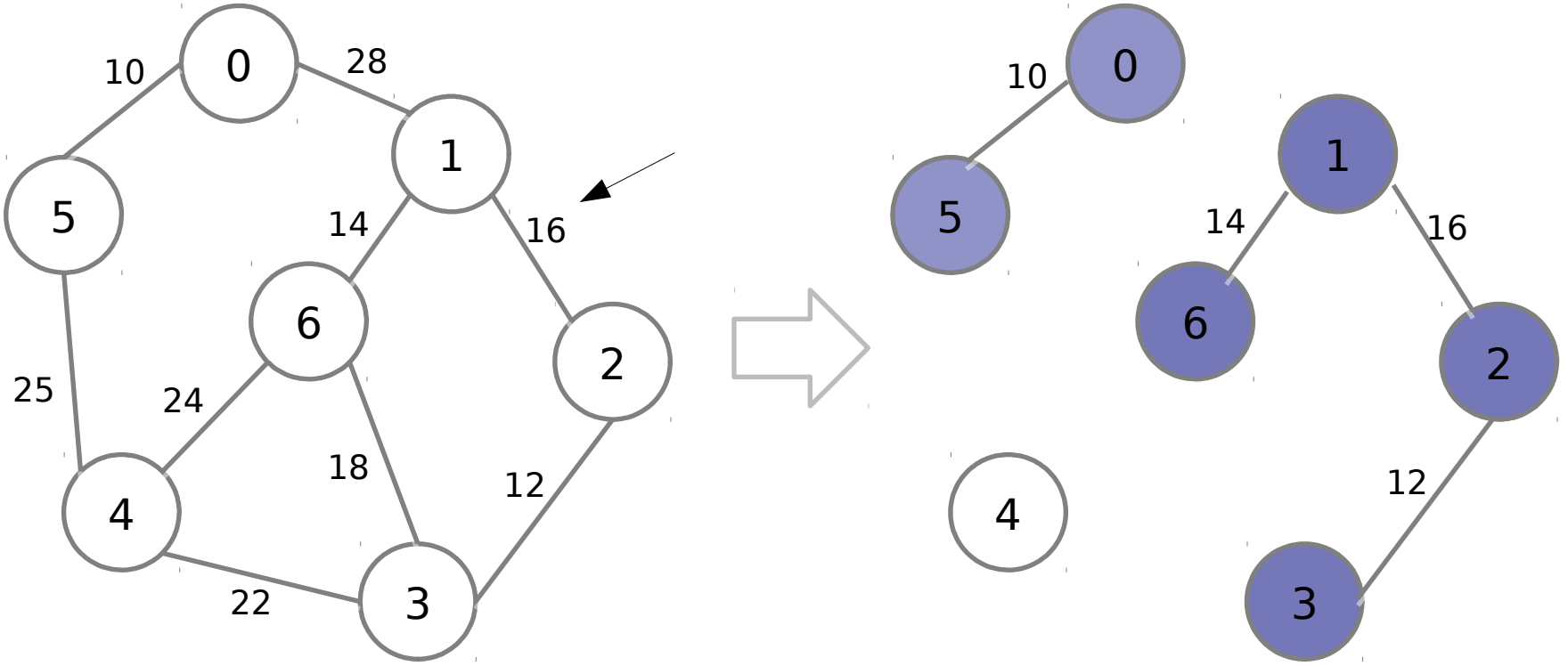
Algoritmo de Kruskal

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



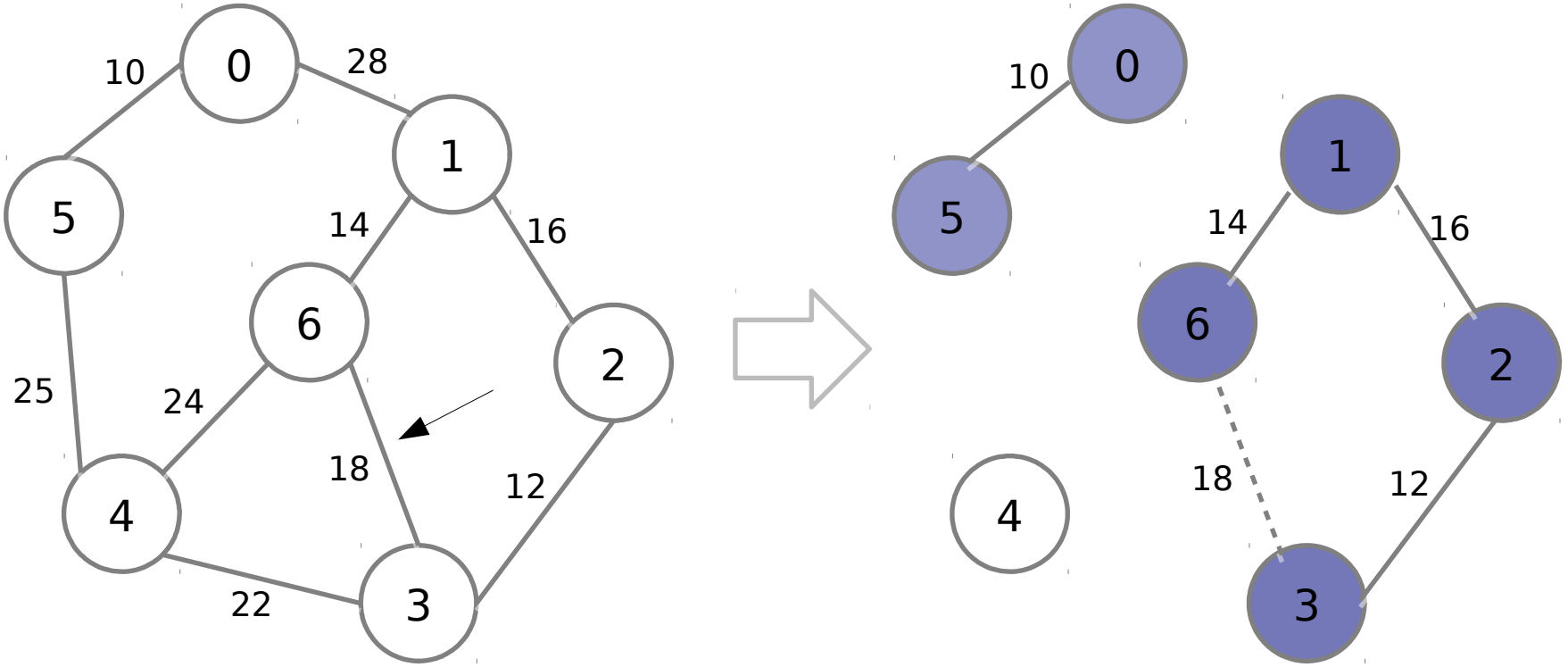
Algoritmo de Kruskal

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



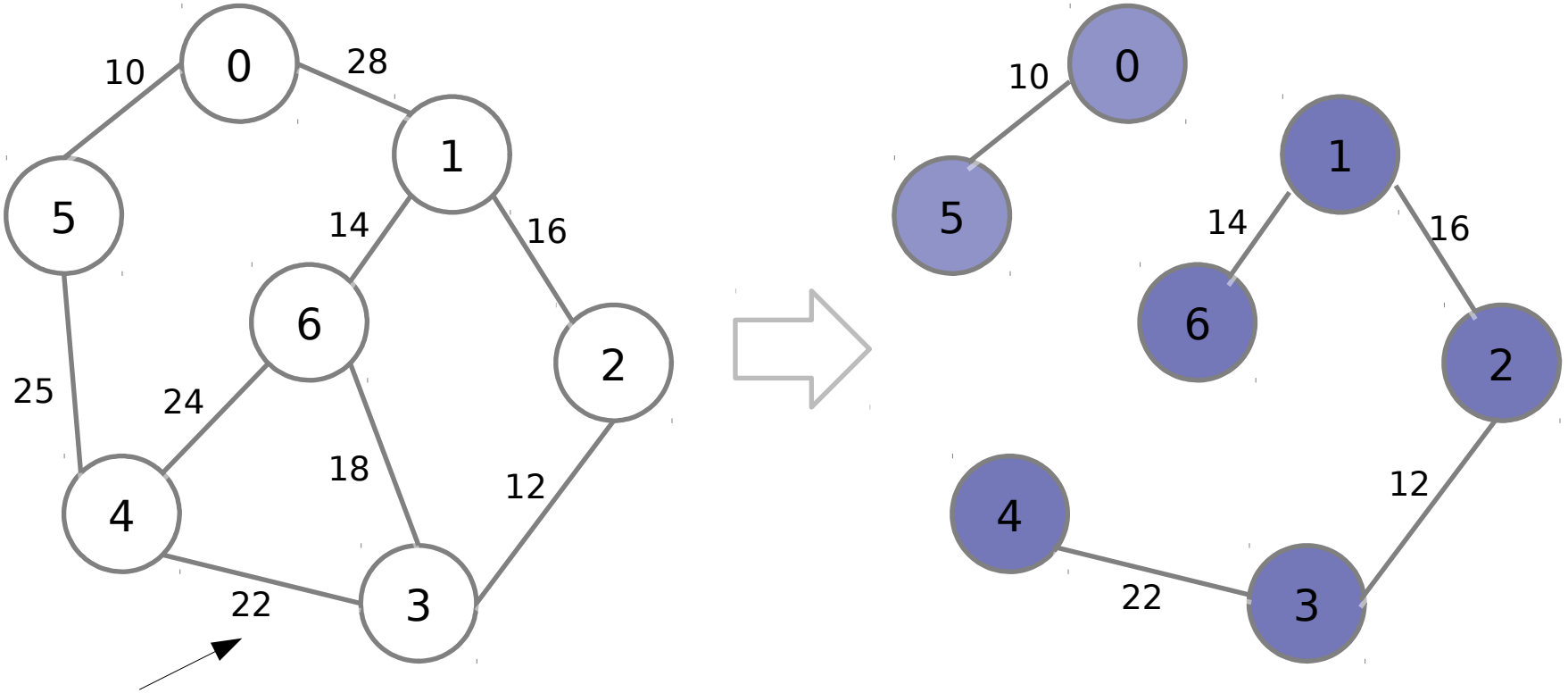
Algoritmo de Kruskal

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



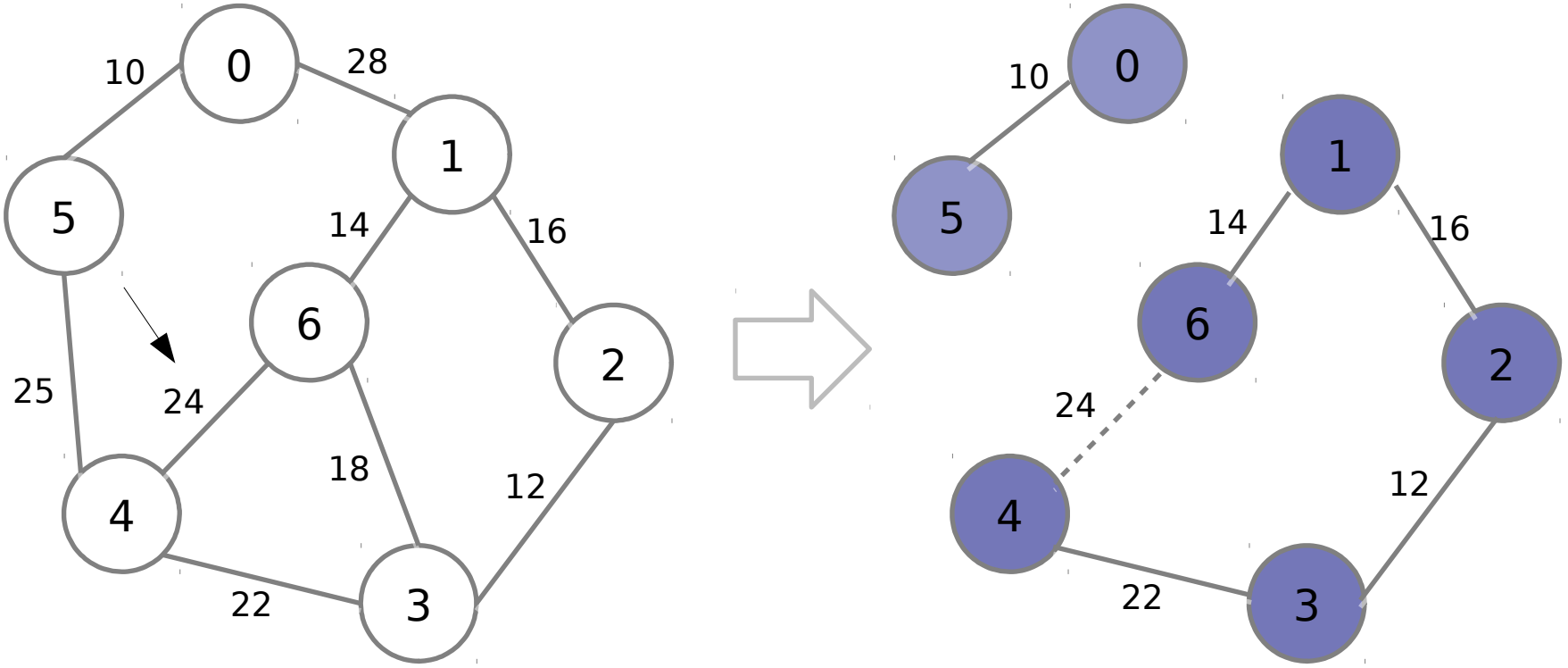
Algoritmo de Kruskal

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



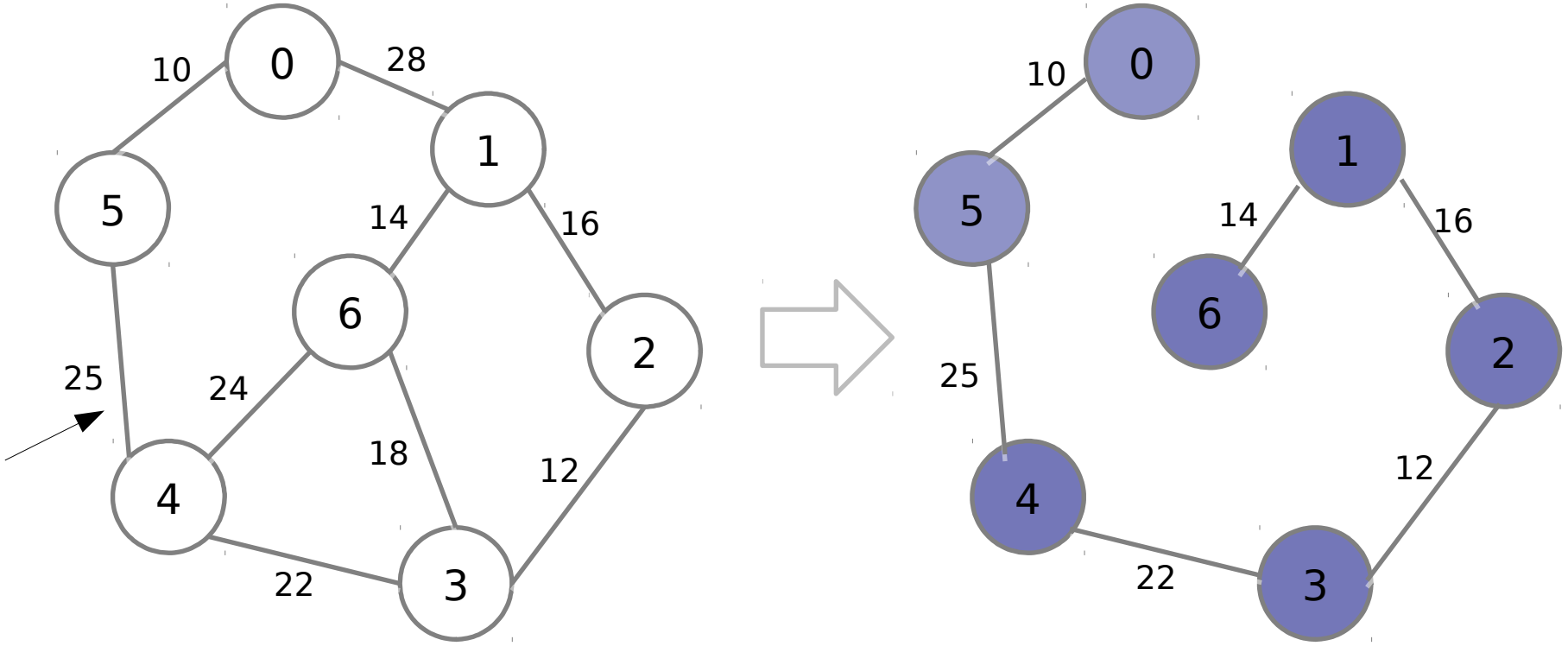
Algoritmo de Kruskal

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

→ o grafo continua representado como antes

→ heap de prioridade para arestas

→ Min Heap para remover aresta de menor custo

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

Implementação

→ o grafo continua representado como antes

→ heap de prioridade para arestas

→ Min Heap para remover aresta de menor custo

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

estrutura de união e busca

Definições

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

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

→ $S_i \subseteq U$ (todos os conjuntos estão contidos no universo)

→ $S_1 \cup \ldots \cup S_k = U$ (cobertura: a união de todos os conjuntos representa o universo)

→ $S_i \cap S_j = \emptyset$, para $i \neq j$ (os conjuntos são **disjuntos**)

Cada conjunto S_i é identificado por um elemento $x \in S_i$ (**representante** de S_i)

Estruturas de União e Busca

Estruturas de Dados para manipulação de **partições** de conjuntos

Operações básicas:

cria()

cria uma partição do conjunto original

busca(x)

informa a qual conjunto (parte) um elemento pertence, retornando o elemento que representa o conjunto

→ útil para determinar se dois elementos estão no mesmo conjunto

união(x,y)

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

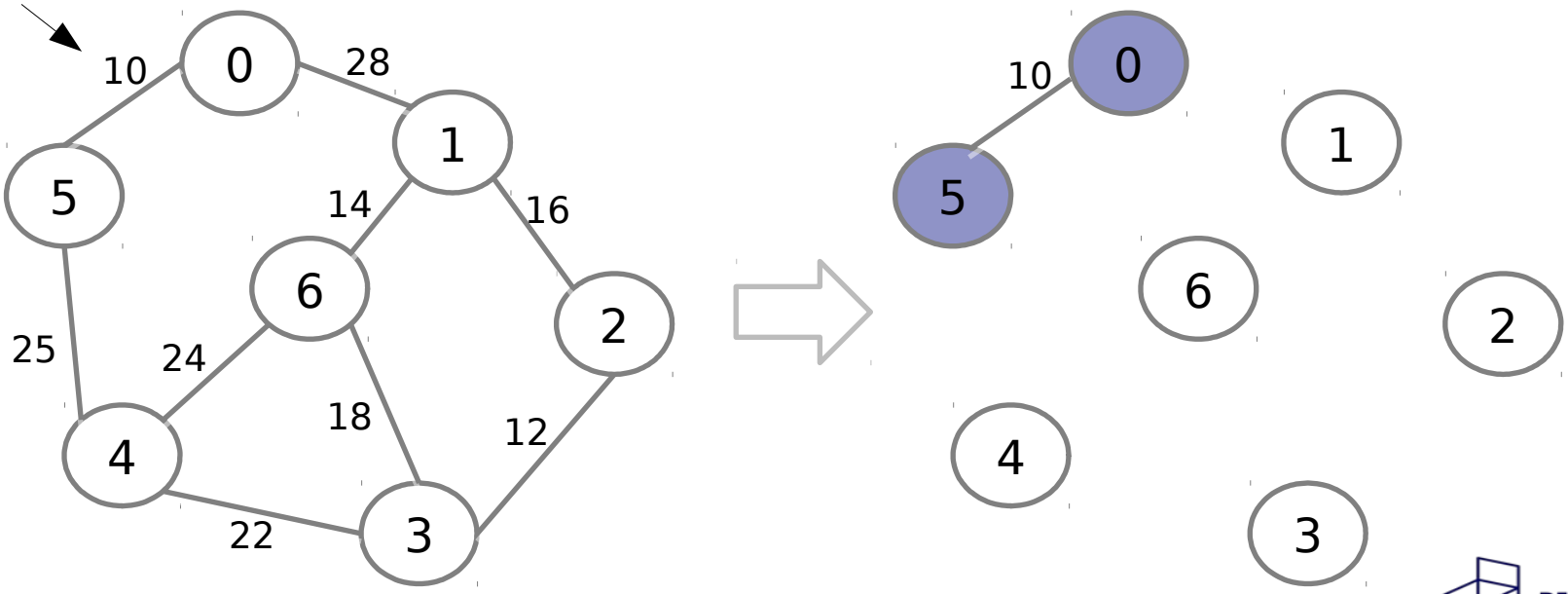
União e Busca para Kruskal

Verificar se arestas unem dois componentes não conexos

→ inicialmente cria-se uma partição com “conjuntos unitários”

→ ao inserir arestas, faz-se a união das partes correspondentes

→ operação de busca indica se vértices já estão conectados

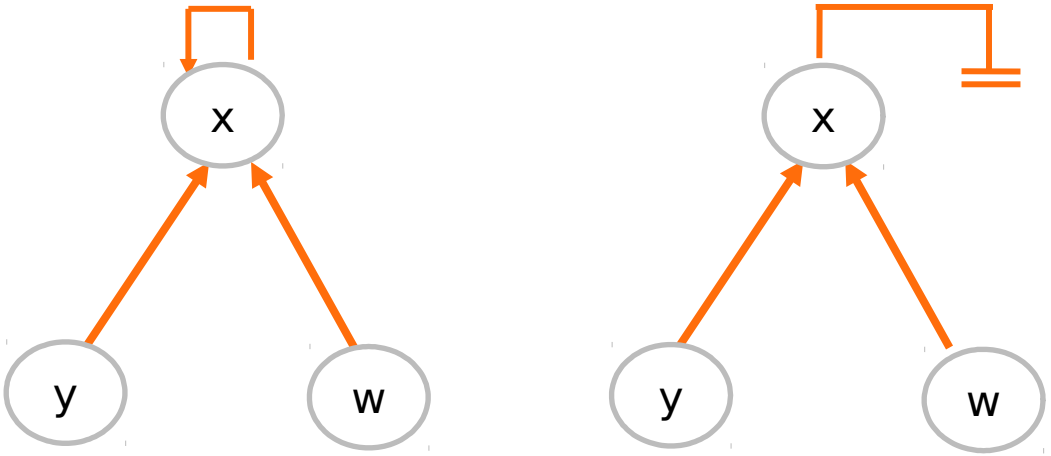


Representação por Árvores Reversas

Cada nó aponta para seu pai

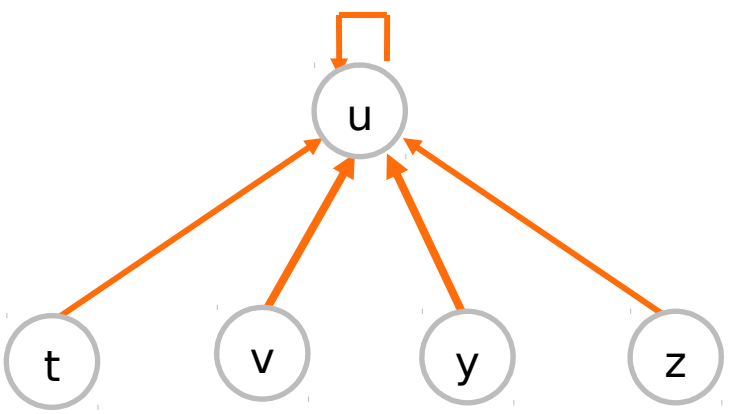
A raiz é o representante do conjunto

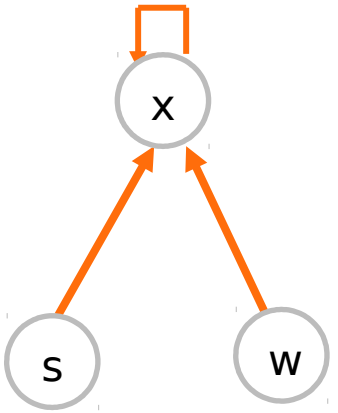
→ aponta para si mesma (ou NULL)



Representação por Árvores Reversas

$$U = \{s,t,u,v,w,x,y,z\}$$





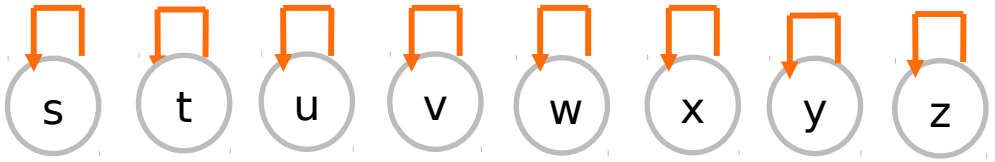
$S_1 = \{t,\underline{\mathbf{u}},v,y,z\}$

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

Operação cria

cria(n) → $\{S_0, \ldots, S_{n-1}\}$

→ cria uma estrutura união-e-busca (partição) com n elementos disjuntos



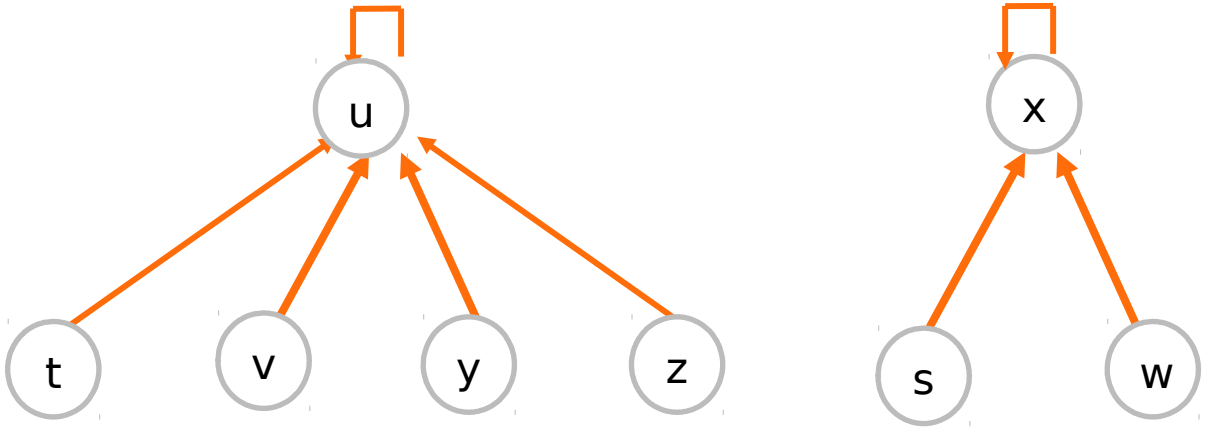
Operação busca

busca(ub, v)

→ identifica o conjunto ao qual o elemento pertence

→ retorna o representante do conjunto

→ caminhar até a raiz



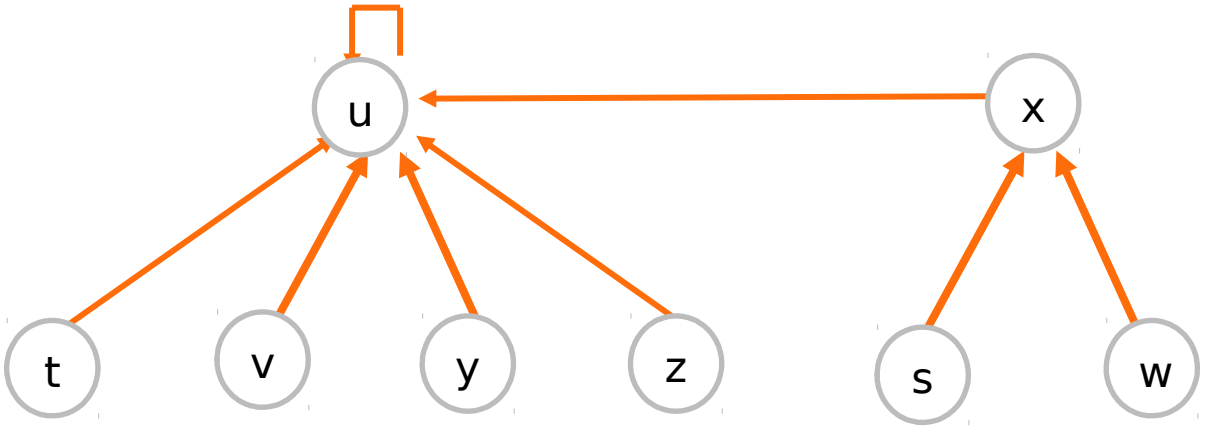
Operação união

união(ub, u, x)

→ une dois conjuntos, fazendo com que a raiz de um aponte para a raiz do outro

→ torna uma árvore uma sub-árvore da outra

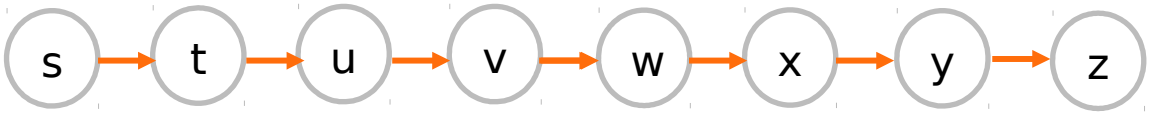
→ se os elementos não são representantes, precisamos achar os representantes (busca)



Complexidade

- busca (e união) → O(h) (altura)
- usando operações de união quaisquer a árvore resultante pode ficar “degenerada”

  → complexidade O(n)



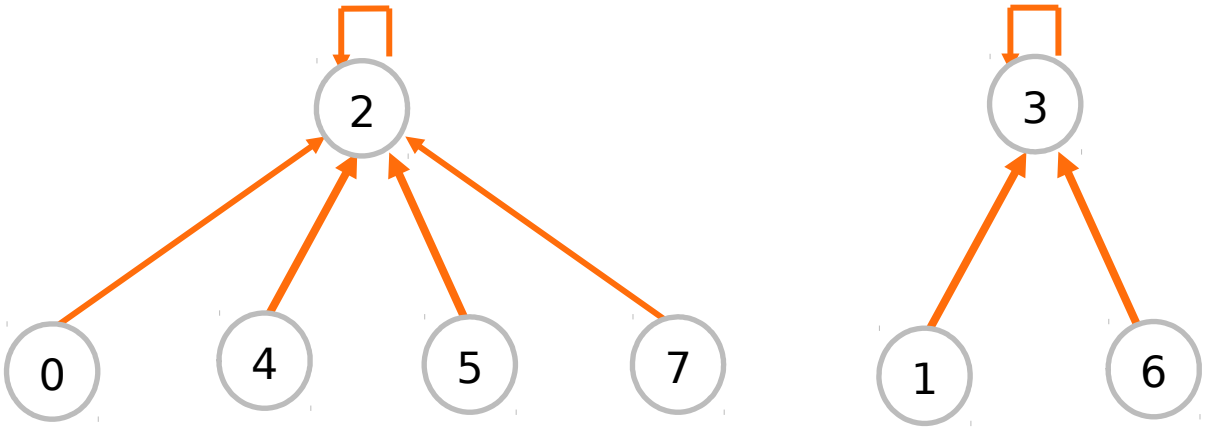
- união por número de nós ou por altura

Representação por Vetor

Cada elemento do vetor representa um elemento do universo (índice = elemento)

→ cada elemento aponta para seu pai (índice do pai)

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



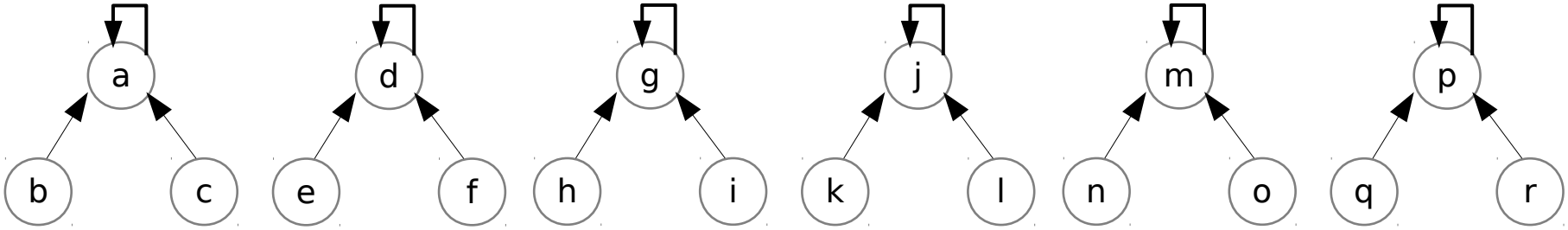
Implementação com Vetor

```
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);
```

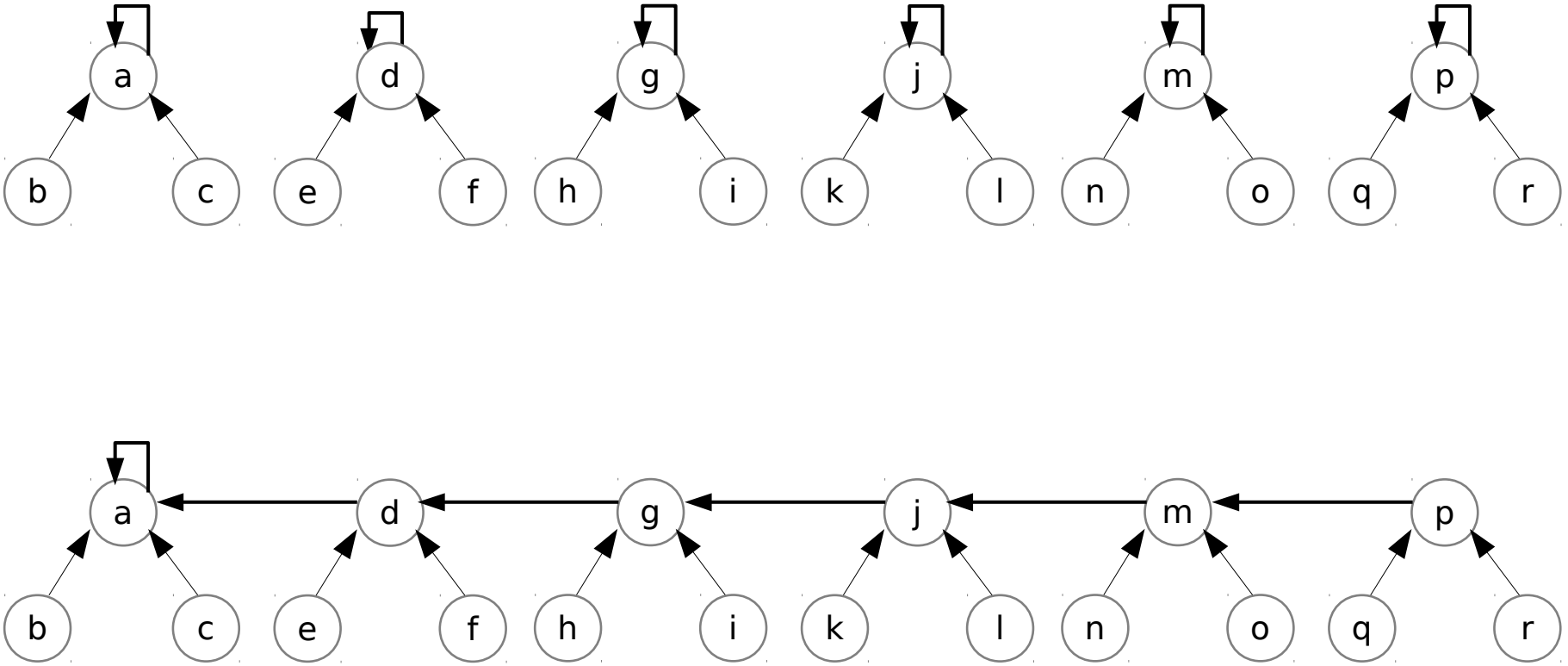
Implementação ineficiente

```
typedef struct uniaobusca UniaoBusca;
struct uniaoBusca {
  int n; int *v;
};

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

União ineficiente



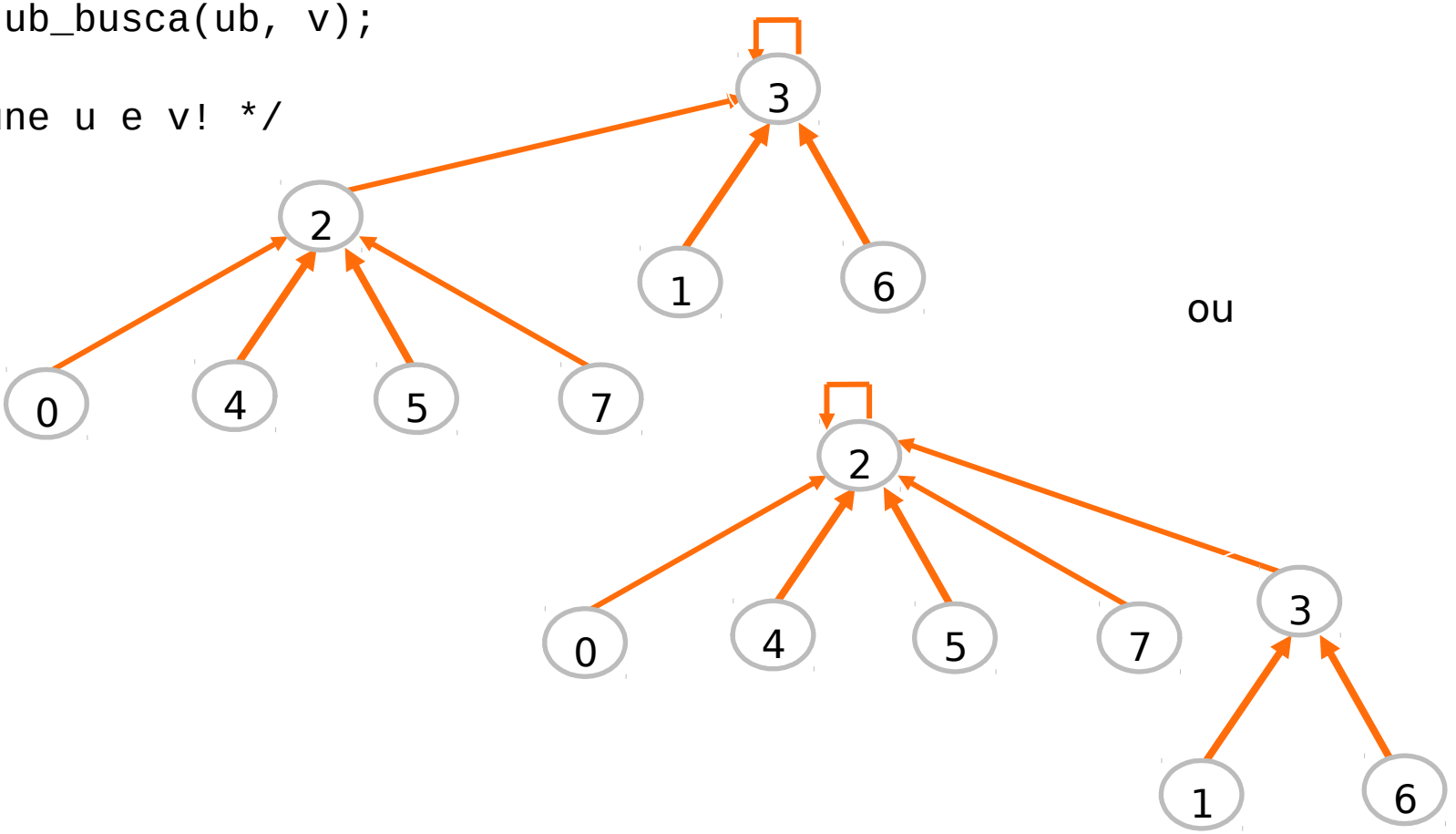
União ineficiente



Operações de busca com tempo linear O(n)

União nas raízes

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



Otimização

"Pendurar" árvore com menor número de nós:

```
int ub_uniao (UniaoBusca* ub, int u, int v){
  u = busca(ub, u);
  v = busca(ub, v);

  /* 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 */
    return v;
  }
  else { /* arvore em v tem menos nós */
    ub->v[v] = u;
    /* atualiza o numero de nós de u */
    return u;
  }
```

Como manter número de nós?

Podemos usar o campo que indica o pai, e nas raizes armazenar -(número de nós) em vez de -1

```
int ub_uniao (UniaoBusca* ub, int u, int v){
  u = busca(ub, u);
  v = busca(ub, v);

  /* uniao com base no numero de nós */
  if (ub->v[u] > ub->v[v] ) { /* negativos: v[u] menor em módulo! */
    ub->v[v] += ub->v[u]; /* atualiza o numero de nós de v */
    ub->v[u] = v;
    return v;
  }
  else { /* arvore em v tem menos nós */
    ub->v[u] += ub->v[v]; /* atualiza o numero de nós de u */
    ub->v[v] = u;
    return u;
  }
```

Otimização

Na busca podemos aproveitar para pendurar nós intermediários diretamente na raiz...

```
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];   /* pai atual */
    ub->v[aux] = u; /* muda para raiz */
  }
  return u;
}
```