# Capítulo VIII

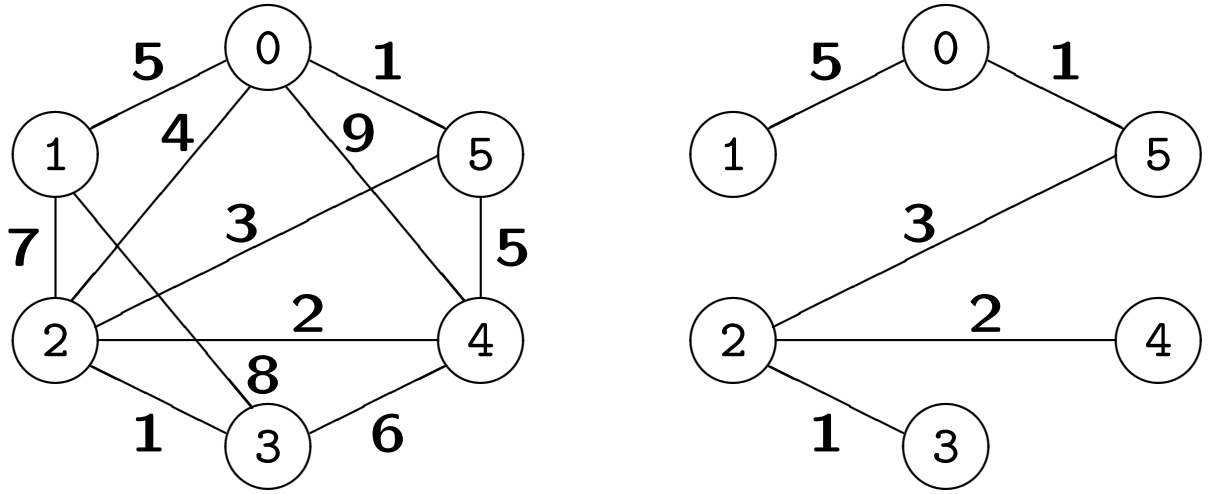
# Árvore Mínima de Cobertura (num grafo não orientado)

—

# Algoritmo de Prim

# Problema

Como ligar um dado equipamento, minimizando o comprimento total da ligação?

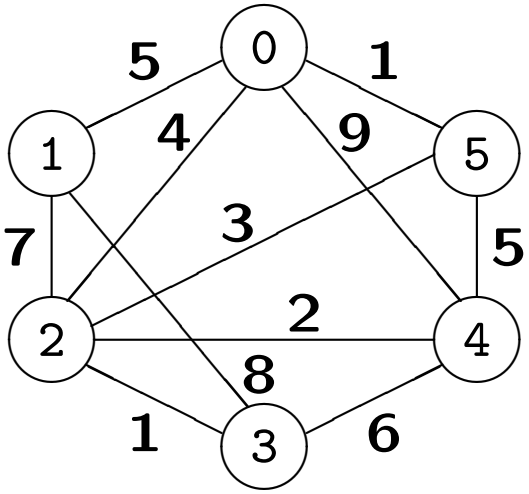


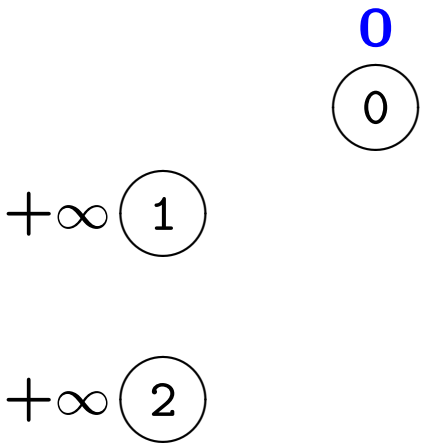

Árvore de Cobertura (sub-grafo acíclico e conexo com todos os vértices) de custo Mínimo (nenhuma árvore de cobertura tem custo menor).

Dado um grafo **não orientado e conexo**, como encontrar uma

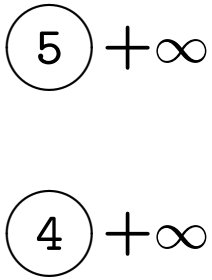
**Árvore Mínima de Cobertura**?

# Algoritmo de Prim [1957]

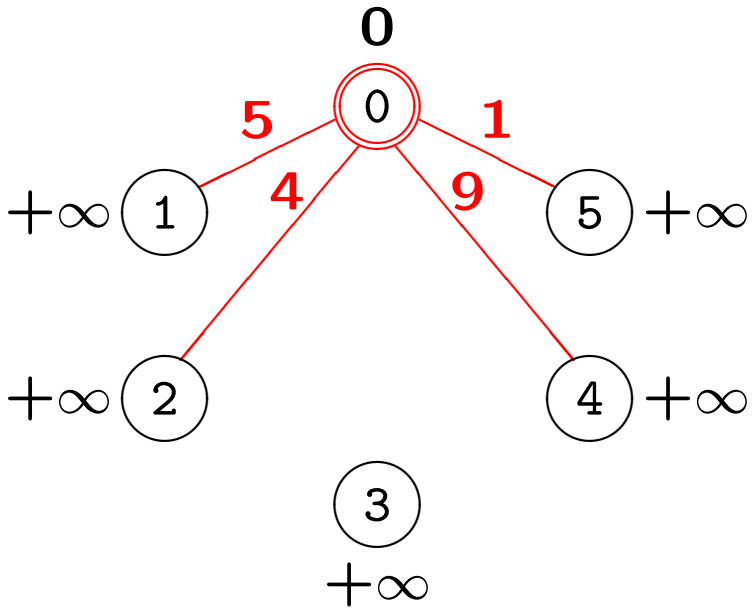


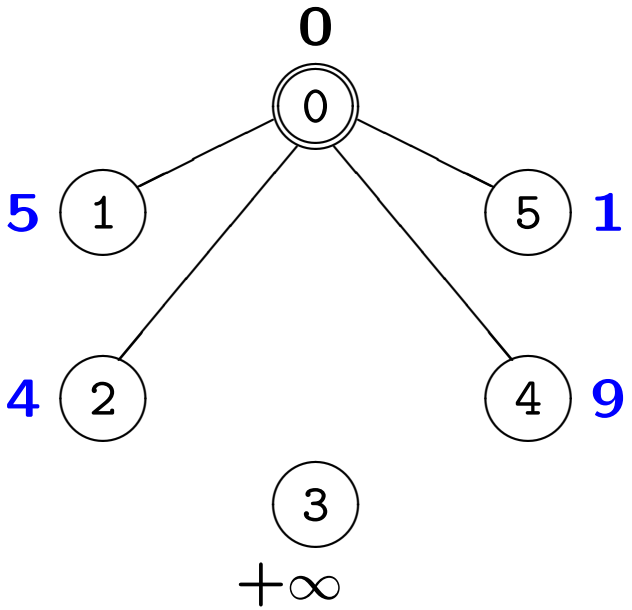




**Inicialização**

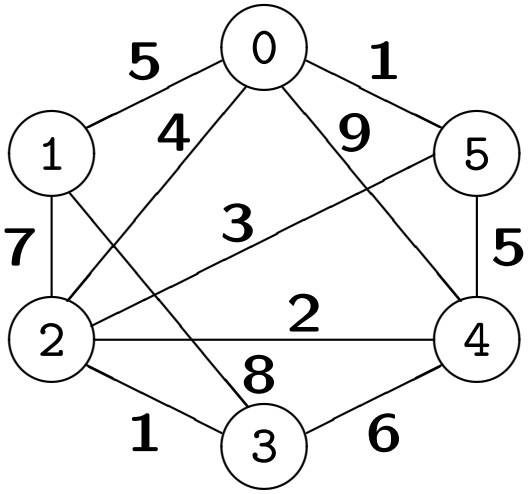


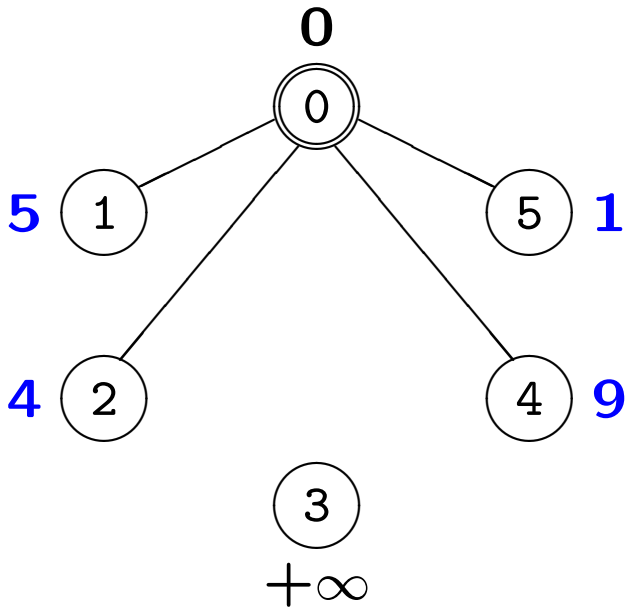

**Seleção de 0**



# Algoritmo de Prim (2)

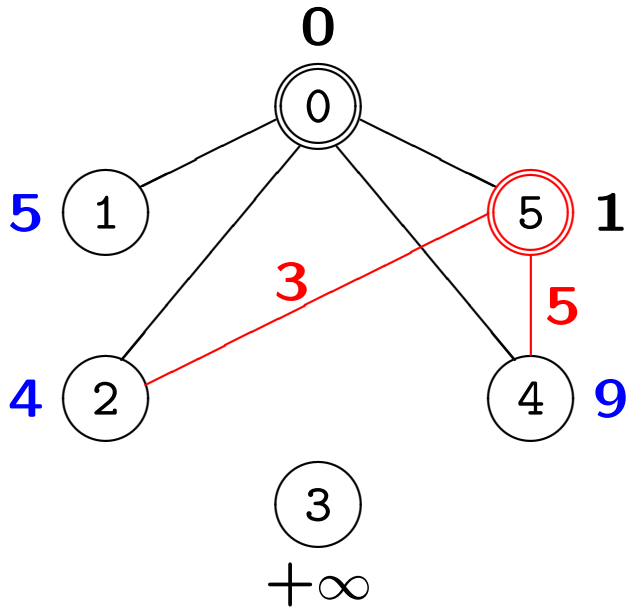


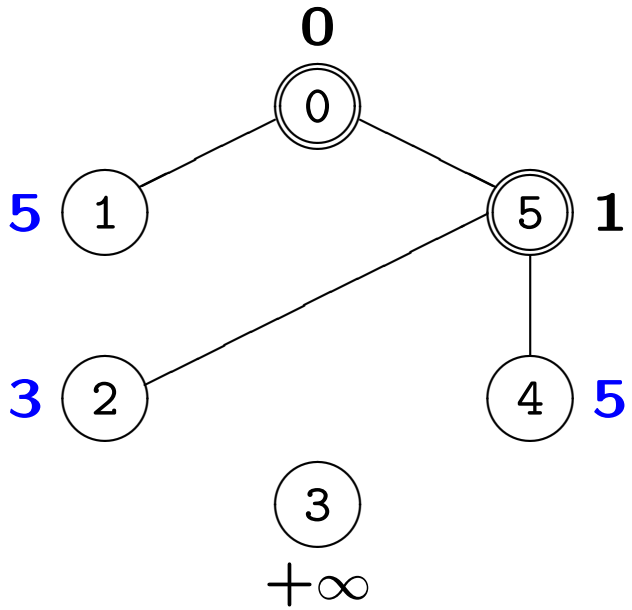




**Situação**

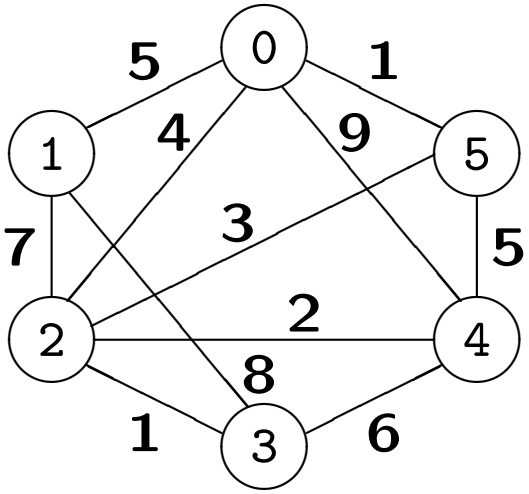
**Corrente**

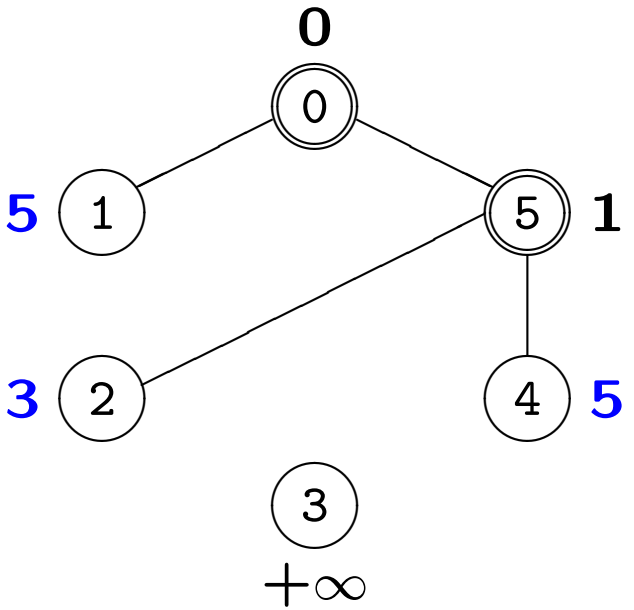
**Seleção de** 5



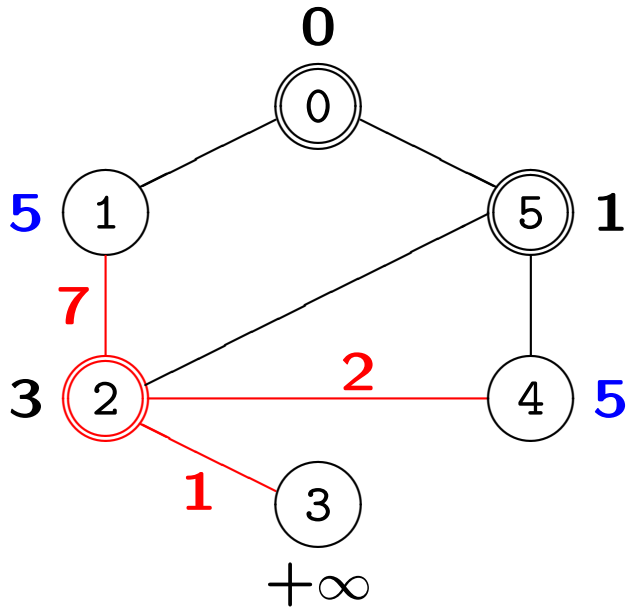
# Algoritmo de Prim (3)

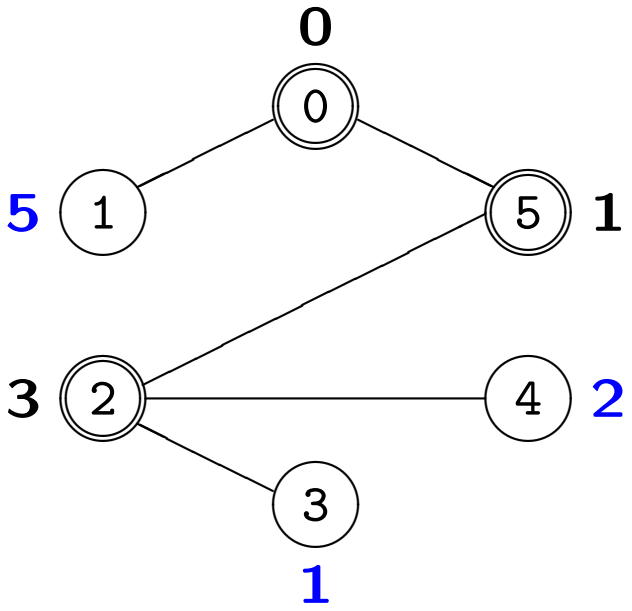


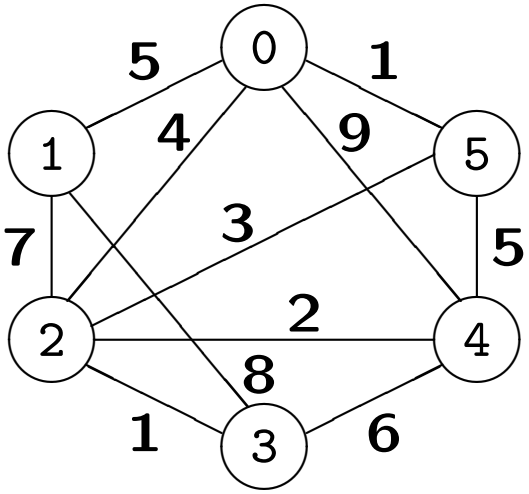

**Situação Corrente**

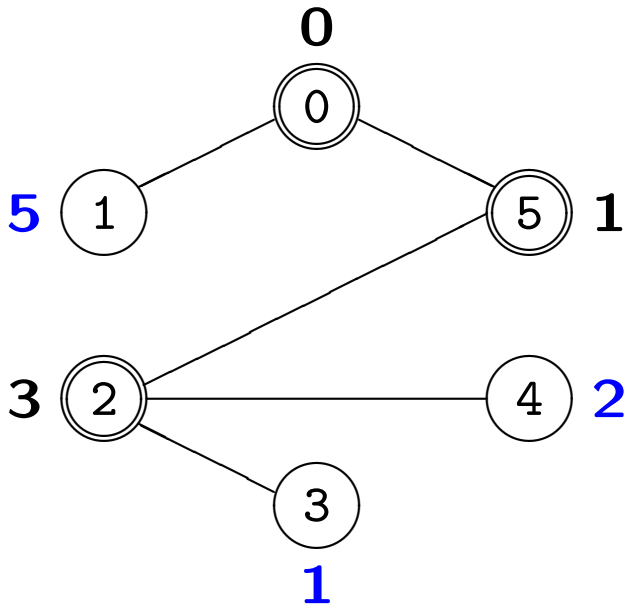
**Seleção de 2**



# Algoritmo de Prim (4)



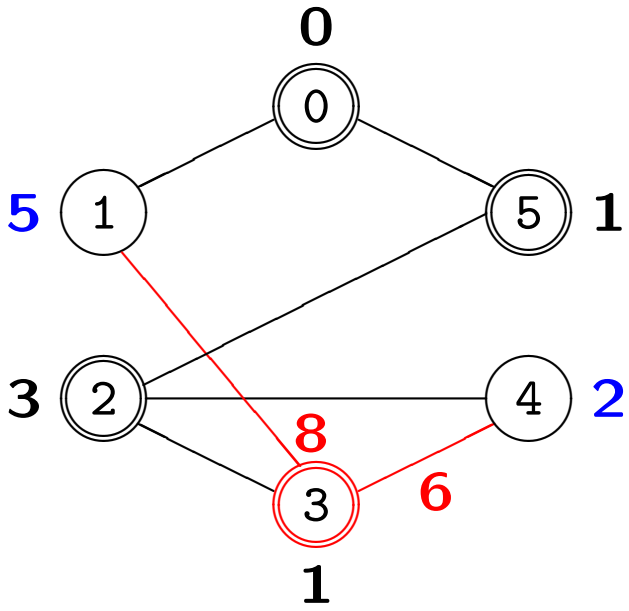



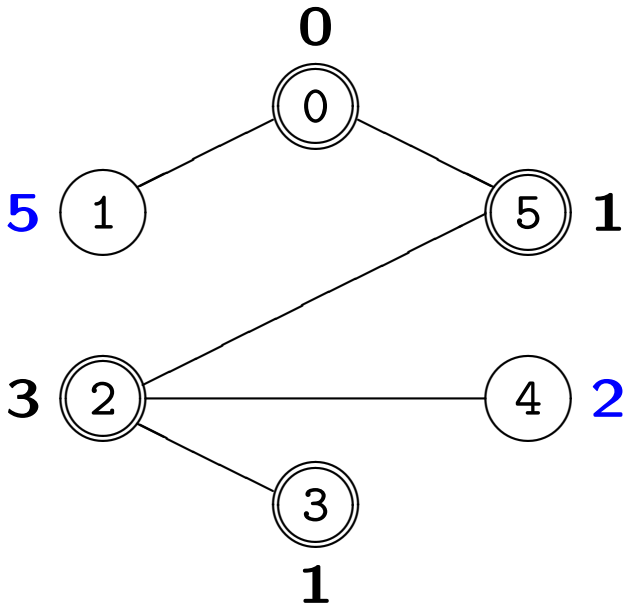

**Situação**

**Corrente**

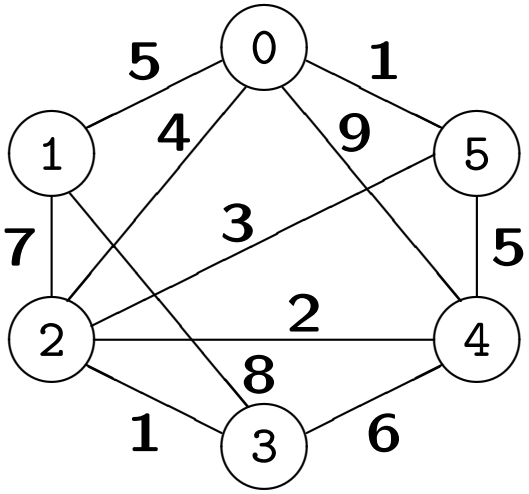
**Seleção**

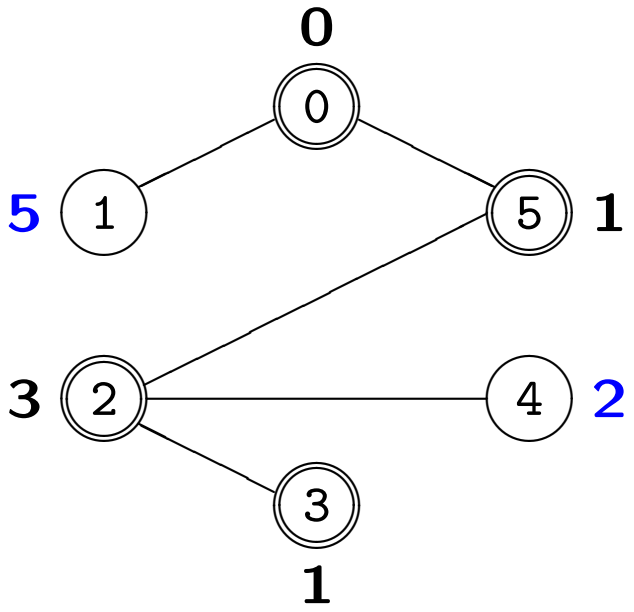
**de** 3



# Algoritmo de Prim (5)



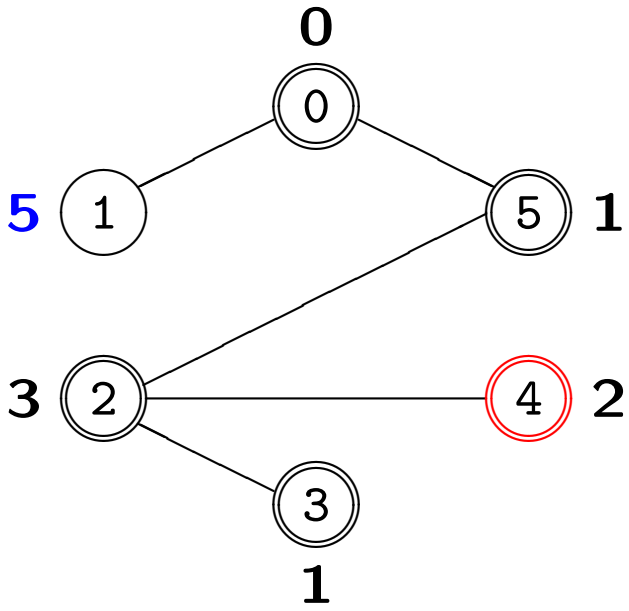



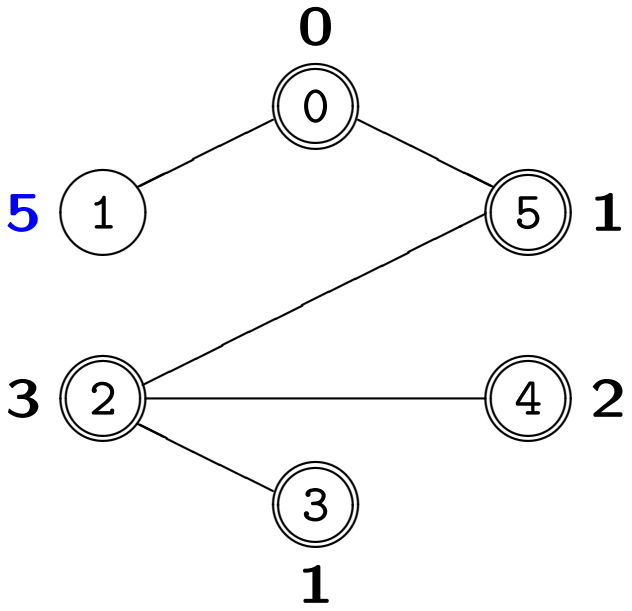

**Situação**

**Corrente**

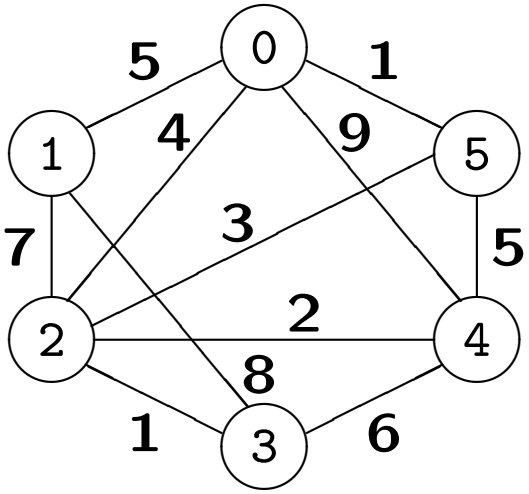
**Seleção**

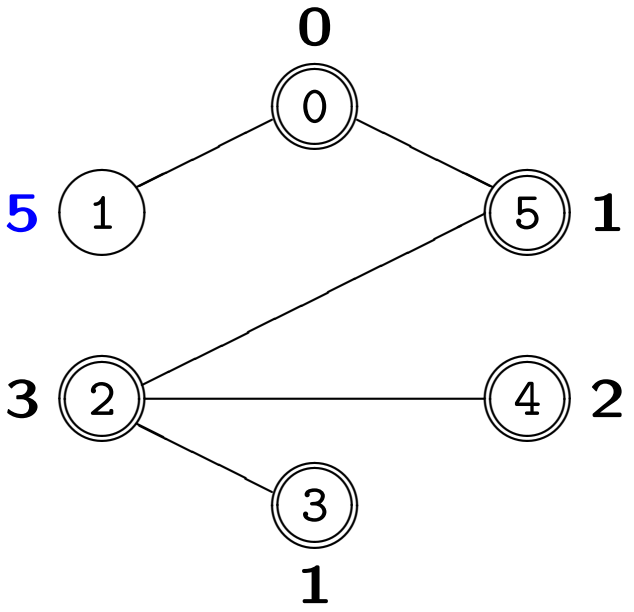
**de** 4



# Algoritmo de Prim (6)

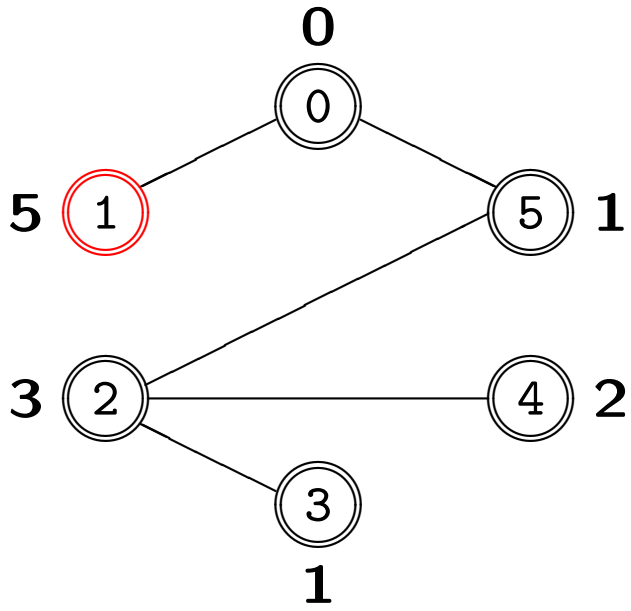


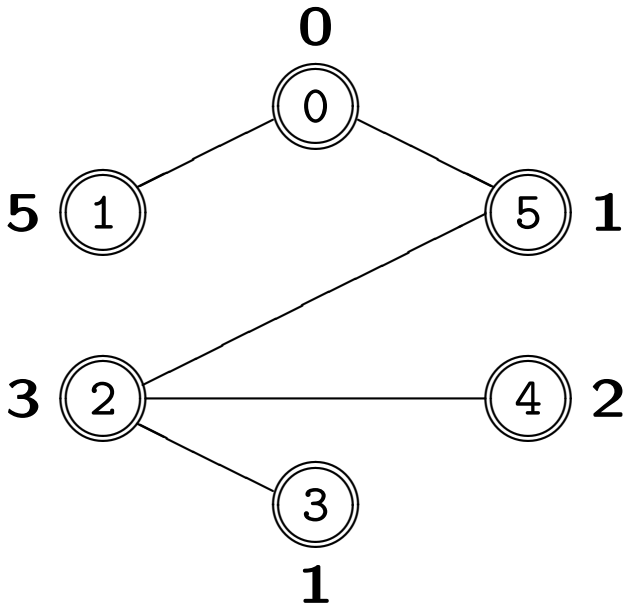




**Situação**

**Corrente**

**Seleção de 1**



# Ideia Geral

Construir a árvore,

partindo de um vértice origem qualquer $o$

e selecionando, em cada passo, um novo vértice

(e um novo arco exceto quando o vértice selecionado é $o$).

# Informação Necessária

**Global: ligados**

Conjunto dos vértices nunca selecionados para os quais já há caminho a partir de $o$.

**Por cada vértice $x$:**

- **boolean** selecionado[$x$]

  Indica se $x$ já foi selecionado, i.e., se já "faz parte da árvore final".

- $\mathbb{R}_0^+ \cup \{+\infty\}$ custo[$x$]

  - **ou** é $+\infty$, quando ainda não há caminho de $o$ para $x$;
  - **ou** é o peso do arco via[$x$], no caso contrário.

- Edge via[$x$]

  Se estiver definido, via[$x$] é um arco de peso mínimo (até ao momento) que liga $x$ à árvore.

# Situação Inicial

**Global: ligados** $= \{o\}$.

**Informação para o vértice** $o$**:**

- selecionado[$o$] = **false**;
- custo[$o$] = 0;
- via[$o$] não está definido.

**Informação para todos os vértices** $x \in V \setminus \{o\}$**:**

- selecionado[$x$] = **false**;
- custo[$x$] = $+\infty$;
- via[$x$] não está definido.

# Em Cada Iteração

Seleciona-se um vértice $x$ de **ligados** tal que custo[$x$] é mínimo.

# Árvore Mínima de Cobertura (1)

```
Iterator<Edge<L>> mstPrim( UndiGraph<L> graph )
{
    boolean[] selected = new boolean[ graph.numVertices() ];

    L[] cost = new L[ graph.numVertices() ];

    Edge<L>[] via = new Edge<L>[ graph.numVertices() ];

    AdaptMinPriQueue<L, Vertex> connected =
        new AdaptMinHeap<L, Vertex>( graph.numVertices() );

    List<Edge<L>> mst = new DoublyLinkedList<Edge<L>>();
```

# Árvore Mínima de Cobertura (2)

```
for every Vertex v in graph.vertices()
{
    selected[v] = false;
    cost[v] = +∞;
}
Vertex origin = graph.aVertex();
cost[origin] = 0;
connected.insert(cost[origin], origin);
```

# Árvore Mínima de Cobertura (3)

```
    do {
        Vertex vertex = connected.removeMin().getValue();
        selected[vertex] = true;
        if ( vertex != origin )
            mst.addLast( via[vertex] );
        exploreVertex(graph, vertex, selected, cost, via, connected);
    }
    while ( !connected.isEmpty() );

    return mst.iterator();
}
```

```
void exploreVertex( UndiGraph<L> graph,   Vertex source,
    boolean[] selected,   L[] cost,   Edge<L>[] via,
    AdaptMinPriQueue<L, Vertex> connected )
{
    for every Edge<L> e in graph.incidentEdges(source)
    {
        Vertex vertex = e.oppositeVertex(source);
        if ( !selected[vertex]   &&   e.label() < cost[vertex] )
        {
            boolean vertexIsInQueue = cost[vertex] < +∞;
            cost[vertex] = e.label();
            via[vertex] = e;
            if ( vertexIsInQueue )
                connected.decreaseKey(vertex, cost[vertex]);
            else
                connected.insert(cost[vertex], vertex);
        }
    }
}
```

# Complexidade do Algoritmo de Prim

## Identificação das Operações

| | |
|---|---|
| criar fila | ? |
| criar lista ligada | $\Theta(1)$ |
| inicializar 2 vetores (selected e cost) | $\Theta(\|V\|)$ |
| inserir origem na fila | ? |
| $\|V\|$ remover mínimo da fila | ? |
| $\|V\| - 1$ inserir à cauda na lista | $\Theta(\|V\|)$ |
| $\|V\|$ percorrer sucessores diretos | $\Theta(\|V\|^2)$ **ou** $\Theta(\|A\|)$ |
| $\leq \|A\|$ inserir na fila ou decrementar chave | ? |

# TAD Fila com Prioridade por Mínimos (K,V)

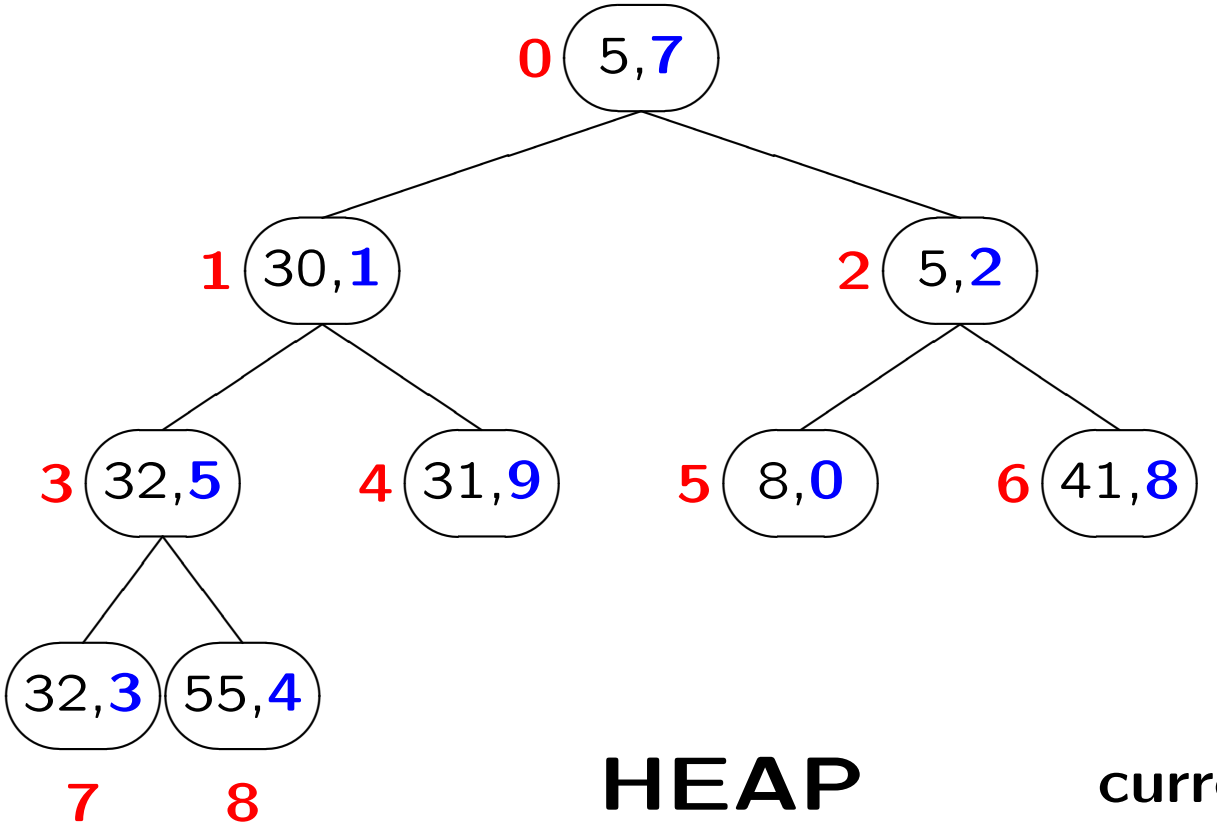
```
public interface MinPriorityQueue<
        K extends Comparable<? super K>, V>
    extends Serializable
{
    // Returns true iff the priority queue contains no entries.
    boolean isEmpty( );

    // Returns the number of entries in the priority queue.
    int size( );

    // Returns an entry with the smallest key in the priority queue.
    Entry<K,V> minEntry( ) throws EmptyQueueException;

    // Inserts the entry (key, value) in the priority queue.
    void insert( K key, V value );

    // Removes an entry with the smallest key from the priority queue
    // and returns that entry.
    Entry<K,V> removeMin( ) throws EmptyQueueException;
}
```

# TAD Fila com Prioridade Adaptável por Mínimos (K,V)

```
public interface AdaptMinPriQueue<
        K extends Comparable<? super K>, V>
    extends MinPriorityQueue<K,V>
{
    // If the priority queue contains an entry with the specified value,
    // returns the associated key and replaces it by the specified key
    // (which is less than the old one). Otherwise, returns null.
    K decreaseKey( V value, K newKey ) throws InvalidKeyException;
}
```

**Nota:** Por questões de eficiência, é usual exigir-se que os valores sejam todos distintos. Nesses casos, o método **insert** levanta uma exceção quando se tenta inserir uma entrada cujo valor já existe na fila.

# Implementação em Heap (K,**V**)

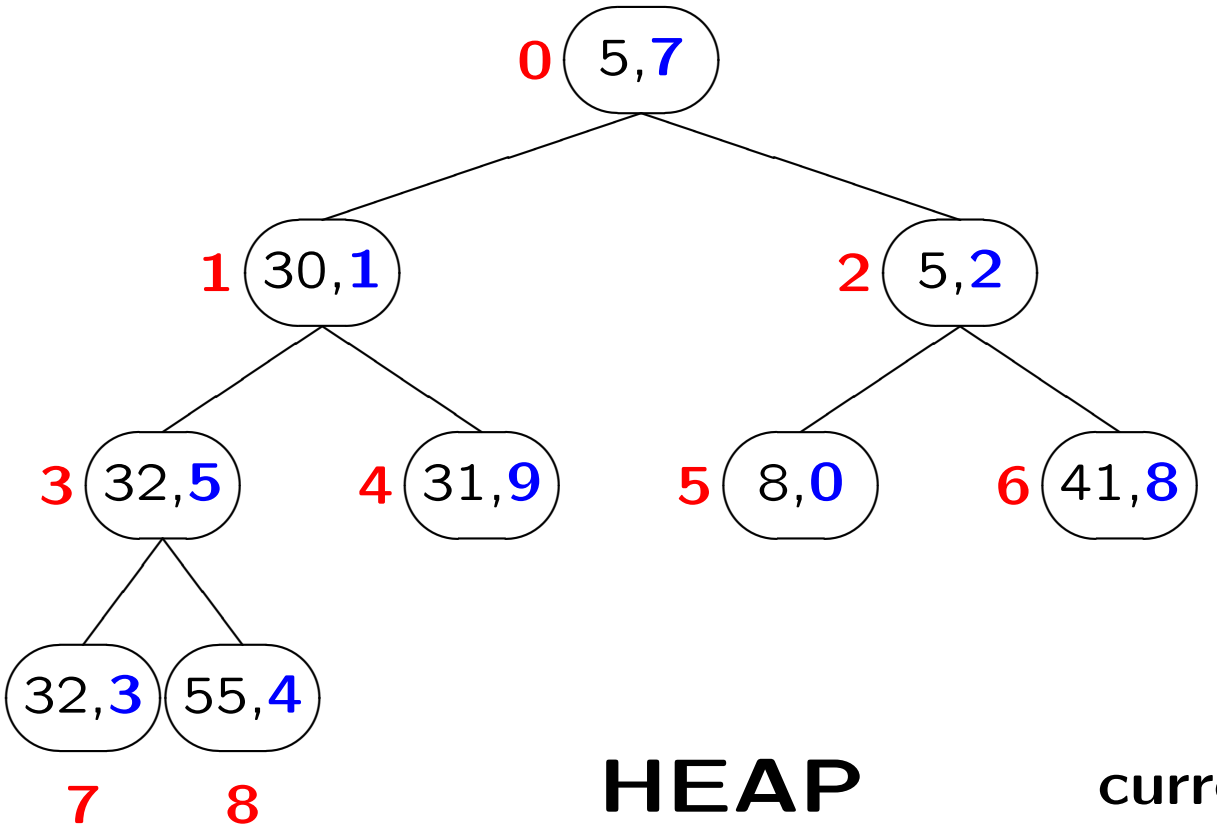



**HEAP**

**currentSize** 9

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 5,7 | 30,1 | 5,2 | 32,5 | 31,9 | 8,0 | 41,8 | 32,3 | 55,4 | |

# Implementação em Heap e Vetor (K,**V**)




**HEAP**

**currentSize** 9

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 5,7 | 30,1 | 5,2 | 32,5 | 31,9 | 8,0 | 41,8 | 32,3 | 55,4 | |

**DIC**

| | |
|---|---|
| 0 | 5 |
| 1 | 1 |
| 2 | 2 |
| 3 | 7 |
| 4 | 8 |
| 5 | 3 |
| 6 | -1 |
| 7 | 0 |
| 8 | 6 |
| 9 | 4 |

# Descrição das Operações com Sucesso (1)

- **void insert**( K key, V value ) **throws** InvalidValueException

  **Dicionário:** Pesquisa-se o valor, para descobrir se já existe.

  **Heap:** Coloca-se a nova entrada no fim do heap e executa-se o borbulhar ascendente a partir dessa posição (a última ocupada).

- Entry<K,V> **removeMin**( ) **throws** EmptyQueueException

  **Heap:** Retorna-se a primeira entrada do heap (posição zero). Coloca-se a última entrada no início do heap e executa-se o borbulhar descendente a partir da posição zero.

Altera-se o dicionário sempre que uma entrada é inserida, é removida ou muda de posição no heap.

# Descrição das Operações com Sucesso (2)

- K **decreaseKey**( V value, K newKey ) **throws** InvalidKeyExcept.

  **Dicionário:** Pesquisa-se o valor, para descobrir a posição da entrada no heap.

  **Heap:** Executa-se o borbulhar ascendente a partir dessa posição.

Altera-se o dicionário sempre que uma entrada muda de posição no heap.

# Complexidades da Fila com Prioridade Adaptável em **Heap** e **Vetor**

($n$ entradas)

| | Melhor Caso | Pior Caso | Caso Esperado |
|---|---|---|---|
| **isEmpty** | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| **size** | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| **minEntry** | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| **insert** | $\Theta(1)$ | $\Theta(\log n)$ | $O(\log n)$ |
| **removeMin** | $\Theta(1)$ | $\Theta(\log n)$ | $O(\log n)$ |
| **decreaseKey** | $\Theta(1)$ | $\Theta(\log n)$ | $O(\log n)$ |

# Complexidades da Fila com Prioridade Adaptável em **Heap** e **Tabela de Dispersão** ($n$ entradas)

| | Melhor Caso | Pior Caso | Caso Esperado |
|---|---|---|---|
| **isEmpty** | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| **size** | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| **minEntry** | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| **insert** | $\Theta(1)$ | $O(n \log n)$ | $O(\log n)$ |
| **removeMin** | $\Theta(1)$ | $O(n \log n)$ | $O(\log n)$ |
| **decreaseKey** | $\Theta(1)$ | $O(n \log n)$ | $O(\log n)$ |

# Complexidade do Algoritmo de Prim

## Fila implementada em Heap e Vetor

| | |
|---|---|
| criar fila | $\Theta(\|V\|)$ |
| criar lista ligada | $\Theta(1)$ |
| inicializar 2 vetores (selected e cost) | $\Theta(\|V\|)$ |
| inserir origem na fila | $\Theta(1)$ |
| $\|V\|$ remover mínimo da fila | $O(\|V\| \times \log \|V\|)$ |
| $\|V\| - 1$ inserir à cauda na lista | $\Theta(\|V\|)$ |
| $\|V\|$ percorrer sucessores diretos | $\Theta(\|V\|^2)$ **ou** $\Theta(\|A\|)$ |
| $\leq \|A\|$ inserir na fila ou decrementar chave | $O(\|A\| \times \log \|V\|)$ |
| **TOTAL** (matriz de adjacências) | |
| **TOTAL** (listas de adjacências) | |

# Complexidade do Algoritmo de Prim

## Fila implementada em Heap e Vetor

| | |
|---|---|
| criar fila | $\Theta(\|V\|)$ |
| criar lista ligada | $\Theta(1)$ |
| inicializar 2 vetores (selected e cost) | $\Theta(\|V\|)$ |
| inserir origem na fila | $\Theta(1)$ |
| $\|V\|$ remover mínimo da fila | $O(\|V\| \times \log \|V\|)$ |
| $\|V\| - 1$ inserir à cauda na lista | $\Theta(\|V\|)$ |
| $\|V\|$ percorrer sucessores diretos | $\Theta(\|V\|^2)$ **ou** $\Theta(\|A\|)$ |
| $\leq \|A\|$ inserir na fila ou decrementar chave | $O(\|A\| \times \log \|V\|)$ |
| **TOTAL** (matriz de adjacências) | $O(\|V\|^2 + \|A\| \times \log \|V\|)$ |
| **TOTAL** (listas de adjacências) | $O(\|A\| \times \log \|V\|)$ |