Capítulo XII

Tipo Abstrato de Dados
Fila com Prioridade Adaptável

—

Fila Binomial e Fila de Fibonacci

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)

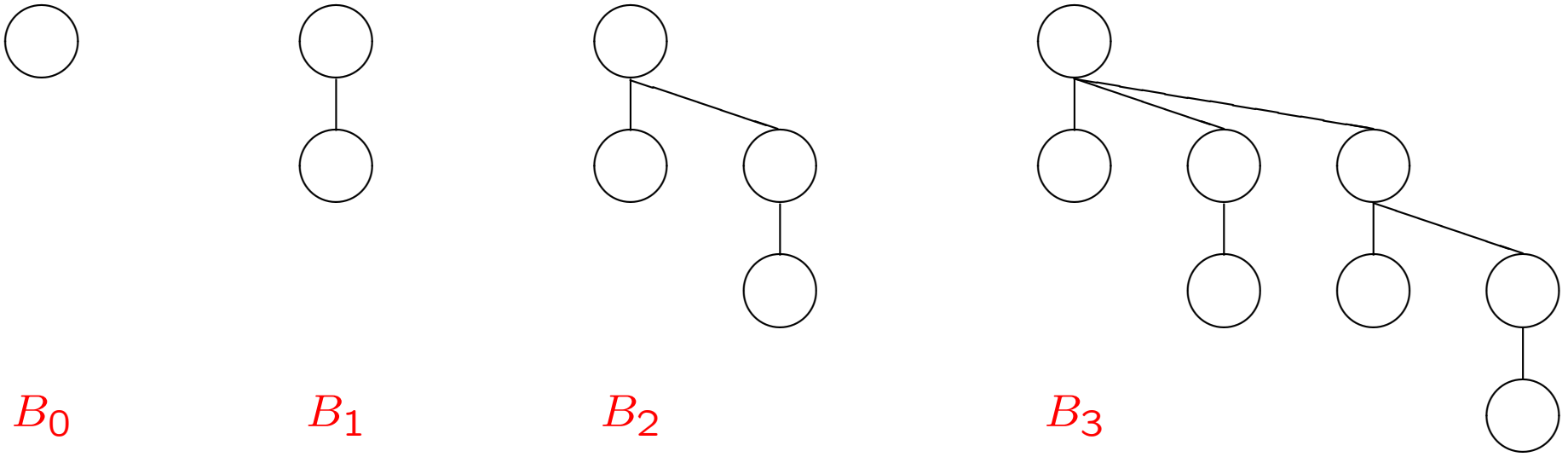
```
public interface AdaptMinPriQueue<
        K extends Comparable<? super K>, V>
    extends MinPriorityQueue<K,V>
{
    // If the priority queue contains an entry with the specified value,
    // returns the associated key and replaces it by the specified key
    // (which is less than the old one).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

Árvore Binomial

Uma **árvore binomial de grau** k, B_k, é uma árvore definida recursivamente.

- B_0 é uma árvore com **1** nó.
- B_k (para $k \geq 1$) é constituída por **2** B_{k-1}, ligadas de forma a que a raiz de uma delas seja o filho mais à direita da raiz da outra.



B_0 B_1 B_2 B_3

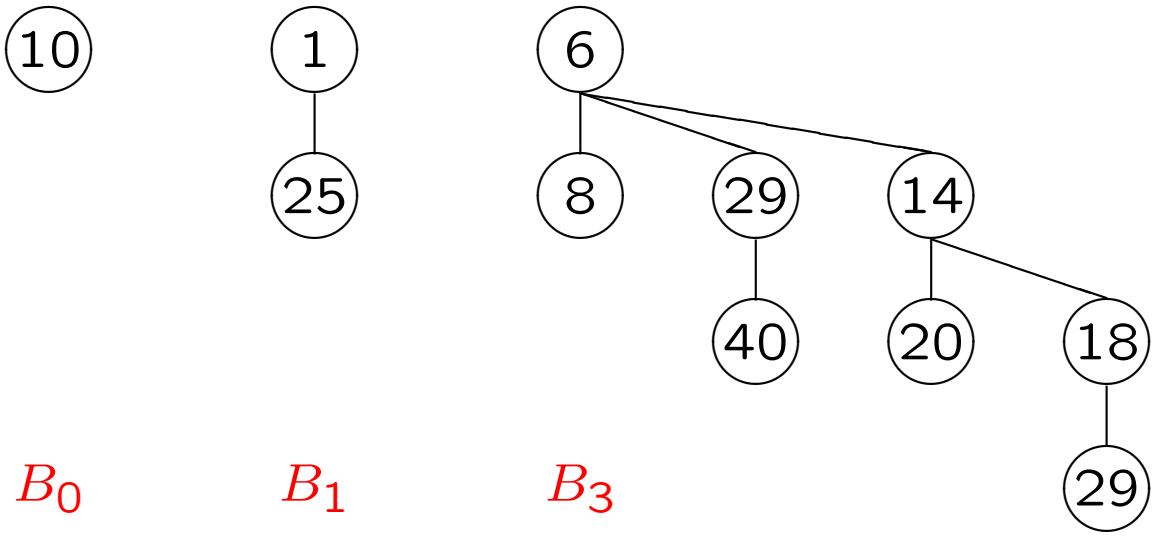
Propriedades das Árvores Binomiais

Seja T uma árvore binomial de grau k (com $k \geq 0$).

- T tem 2^k nós.
- A altura de T é $k + 1$.
- Existem $\binom{k}{i}$ nós à profundidade i, para $i = 0, \ldots, k$.
- A raiz de T tem k filhos.
- Se $A_0, A_1, \ldots, A_{k-1}$ forem as sub-árvores da raiz de T, numeradas da esquerda para a direita, A_j é uma árvore binomial de grau j (para $j = 0, 1, \ldots, k - 1$).

Fila Binomial [Vuillemin 78]

Uma **fila binomial** é uma floresta de árvores binomiais com prioridade de graus distintos.



Propriedades das Filas Binomiais

Seja F uma fila binomial com $n \geq 1$ elementos.

- O mínimo de F encontra-se na raiz de uma árvore de F.
- Se a representação em binário de n for

$$\langle b_{\lfloor \log n \rfloor}, b_{\lfloor \log n \rfloor - 1}, \ldots, b_1, b_0 \rangle,$$

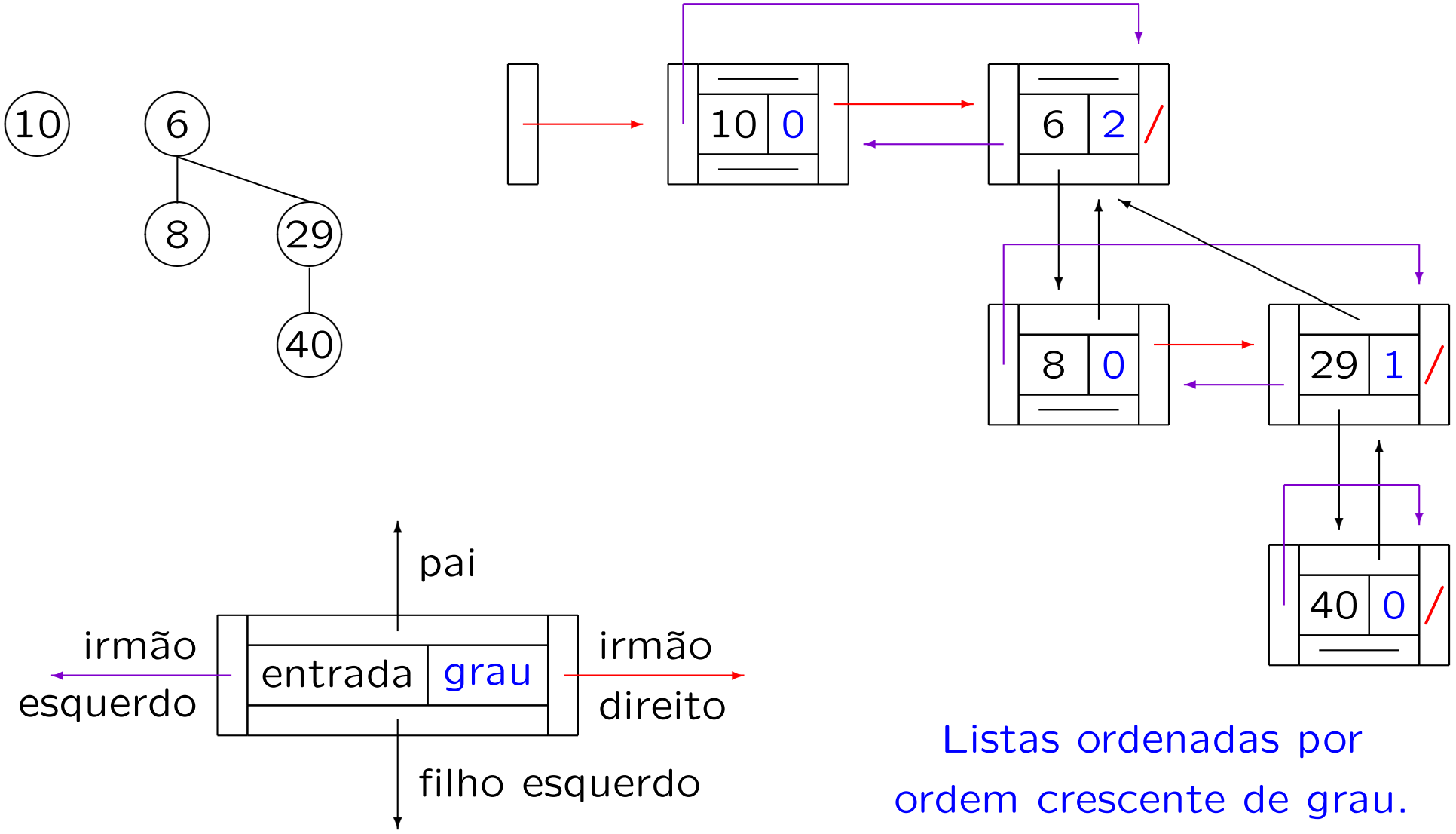
  F tem uma árvore de grau k **sse** $b_k = 1$ (para $k = 0, 1, \ldots, \lfloor \log n \rfloor$).

- O número de árvores de F é, no máximo, $\lfloor \log n \rfloor + 1$.
- A altura da "maior" árvore de F é $\lfloor \log n \rfloor + 1$.

Implementação da Fila Binomial (1)

- A fila é implementada em lista duplamente ligada, com cabeça, **ordenada** por ordem crescente de grau.
- A cabeça da lista (que implementa a fila) aponta para a árvore de menor grau.
- Os filhos de um nó estão implementados em lista duplamente ligada, com cabeça, **ordenada** por ordem crescente de grau.
- A cabeça da lista dos filhos aponta para o filho de grau zero.
- Em qualquer lista, o anterior da cabeça (chamado filho esquerdo) aponta para a cauda da lista.

Implementação da Fila Binomial (2)



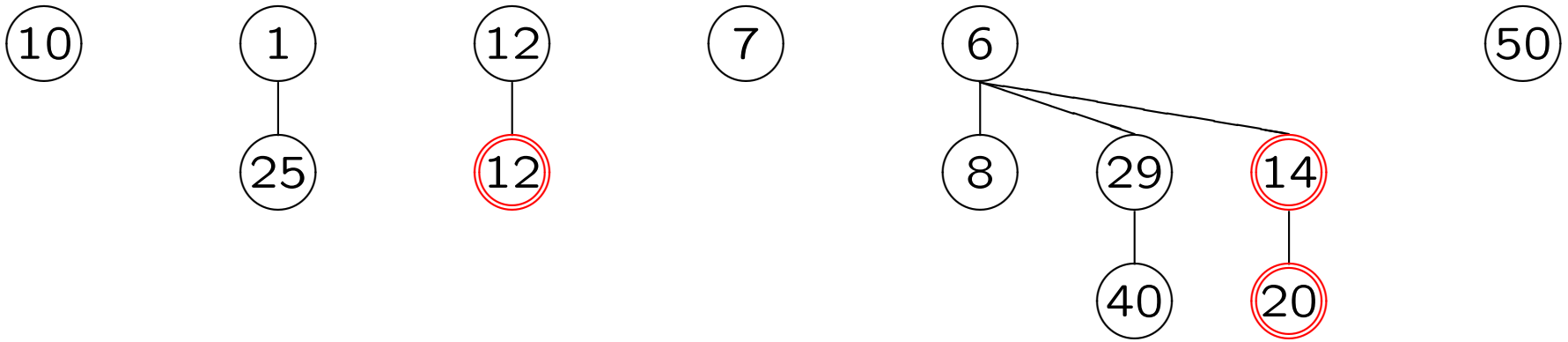
Listas ordenadas por ordem crescente de grau.

Complexidades da Fila com Prioridade Adaptável (n entradas)

	Heap	Fila Binomial
isEmpty	$\Theta(1)$	$\Theta(1)$
size	$\Theta(1)$	$\Theta(1)$
minEntry	$\Theta(1)$	$O(\log n)$
insert	$O(\log n)$	$O(\log n)$
removeMin	$O(\log n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(\log n)$

Fila de Fibonacci [Fredman e Tarjan 87]

Uma **fila de Fibonacci** é uma floresta de árvores com prioridade.



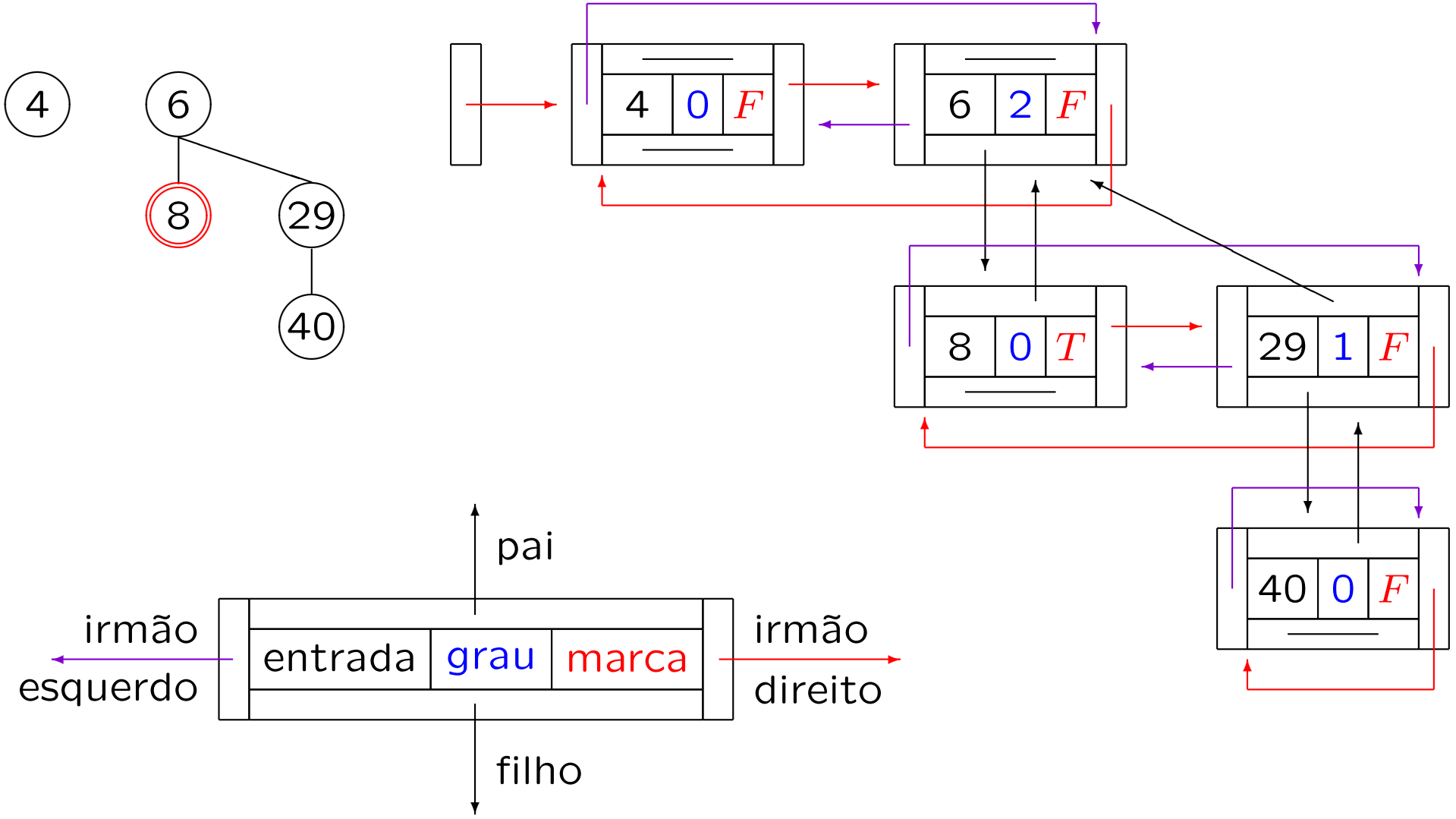
Alguns nós estão **marcados**.

- Um nó está marcado se perdeu um filho desde que é filho do pai atual.
- Uma raiz nunca está marcada.

Implementação da Fila de Fibonacci (1)

- A fila é implementada em lista **circular** duplamente ligada, com cabeça.
- A cabeça da lista (que implementa a fila) aponta para uma árvore cuja raiz tem a chave mínima.
- Os filhos de um nó estão implementados em lista **circular** duplamente ligada, com cabeça.
- A cabeça da lista dos filhos aponta para um filho qualquer.

Implementação da Fila de Fibonacci (2)



Classe Nó de Árvore de Fibonacci (1)

```
class FibNode<K,V>
{
    // Entry stored in the node.
    private EntryClass<K,V> entry;

    // The degree of the node.
    private int degree;

    // (Pointer to) a child.
    private FibNode<K,V> child;

    // (Pointer to) the left sibling.
    private FibNode<K,V> leftSibling;

    // (Pointer to) the right sibling.
    private FibNode<K,V> rightSibling;

    // (Pointer to) the parent.
    private FibNode<K,V> parent;
```

Classe Nó de Árvore de Fibonacci (2)

```
// The node is marked if it has lost a child since
// it has the current parent. Roots are unmarked.
private boolean mark;

public FibNode( K key, V value )
{
    entry = new EntryClass<K,V>(key, value);
    degree = 0;
    child = null;
    leftSibling = this;
    rightSibling = this;
    parent = null;
    mark = false;
}

// Todos os SETs e GETs.
.........................
```

Classe Nó de Árvore de Fibonacci (3)

```
public void incrementDegree( )
{
    degree++;
}

public void decrementDegree( )
{
    degree--;
}

public void makeSingleton( )
{
    leftSibling = this;
    rightSibling = this;
}
```

Classe Nó de Árvore de Fibonacci (4)

```
    public boolean isMarked( )
    {
        return mark;
    }

    public void mark( )
    {
        mark = true;
    }

    public void unmark( )
    {
        mark = false;
    }

}   // End of class FibNode.
```

Criação de uma Fila Vazia

- **FibQueue**( )
  - Inicializa-se **min** a **null**.
  - Inicializa-se **currentSize** a 0.

  Complexidade: $\Theta(1)$ em todos os casos.

Classe **Interna** Fila de Fibonacci

```
class FibQueue<K extends Comparable<? super K>, V>
{
    // (Pointer to) a tree with the smallest key.
    protected FibNode<K,V> min;

    // Number of entries in the priority queue.
    protected int currentSize;


    public FibQueue( )
    {
        min = null;
        currentSize = 0;
    }


    ..........................
}
```

Classe **Pública** Fila de Fibonacci (1)

```
public class FibonacciQueue<K extends Comparable<? super K>, V>
    implements AdaptMinPriQueue<K,V>
{
    // The Fibonacci queue.
    protected FibQueue<K,V> queue;

    // The dictionary.
    protected Map<V, FibNode<K,V>> dict;


    public FibonacciQueue( int capacity )
    {
        queue = new FibQueue<K,V>();
        dict = new HashMap<V, FibNode<K,V>>(capacity);
    }

    ..........................
}
```

Classe **Pública** Fila de Fibonacci (2)

```
// Returns true iff the priority queue contains no entries.
public boolean isEmpty( )
{
    return queue.isEmpty();
}


// Returns the number of entries in the priority queue.
public int size( )
{
    return queue.size();
}
```

Classe **Pública** Fila de Fibonacci (3)

```
// Returns an entry with the smallest key in the priority queue.
public Entry<K,V> minEntry( ) throws EmptyQueueException
{
    if ( this.isEmpty() )
        throw new EmptyQueueException();

    return queue.minEntry();
}
```

Classe **Pública** Fila de Fibonacci (4)

```
// Inserts the entry (key, value) in the priority queue.
public void insert( K key, V value ) throws InvalidValueException
{
    if ( dict.containsKey(value) )
        throw new InvalidValueException(
            "The queue already has an entry with the value.");

    FibNode<K,V> node = queue.insert(key, value);
    dict.put(value, node);
}
```

Classe **Pública** Fila de Fibonacci (5)

```
// Removes an entry with the smallest key from the priority queue
// and returns that entry.
public Entry<K,V> removeMin( ) throws EmptyQueueException
{
    if ( this.isEmpty() )
        throw new EmptyQueueException();

    Entry<K,V> entry = queue.removeMin();
    dict.remove( entry.getValue() );
    return entry;
}
```

Classe **Pública** Fila de Fibonacci (6)

```
// 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.
public K decreaseKey( V value, K newKey ) throws InvalidKeyException
{
    FibNode<K,V> node = dict.get(value);
    if ( node == null )
        return null;

    K oldKey = node.getKey();
    if ( oldKey.compareTo(newKey) <= 0 )
        throw new InvalidKeyException(
            "The specified key is not less than the existent one.");

    queue.decreaseKey(node, newKey);
    return oldKey;
}
```

Métodos Públicos da Classe Interna

- **boolean** **isEmpty**( );
- **int** **size**( );
- Entry<K,V> **minEntry**( );
- FibNode<K,V> **insert**(K key, V value);
- Entry<K,V> **removeMin**( );
- **void** **decreaseKey**(FibNode<K,V> node, K newKey);

Métodos **isEmpty**, **size** e **minEntry**

(fila com n entradas)

- **boolean** **isEmpty**( )
  - Testa-se se **min** é **null**.

  Complexidade: $\Theta(1)$ em todos os casos.

- **int** **size**( )
  - Retorna-se o valor guardado em **currentSize**.

  Complexidade: $\Theta(1)$ em todos os casos.

- Entry<K,V> **minEntry**( )
  - Retorna-se a entrada guardada em **min**.

  Complexidade: $\Theta(1)$ em todos os casos.

Classe **Interna** Fila de Fibonacci

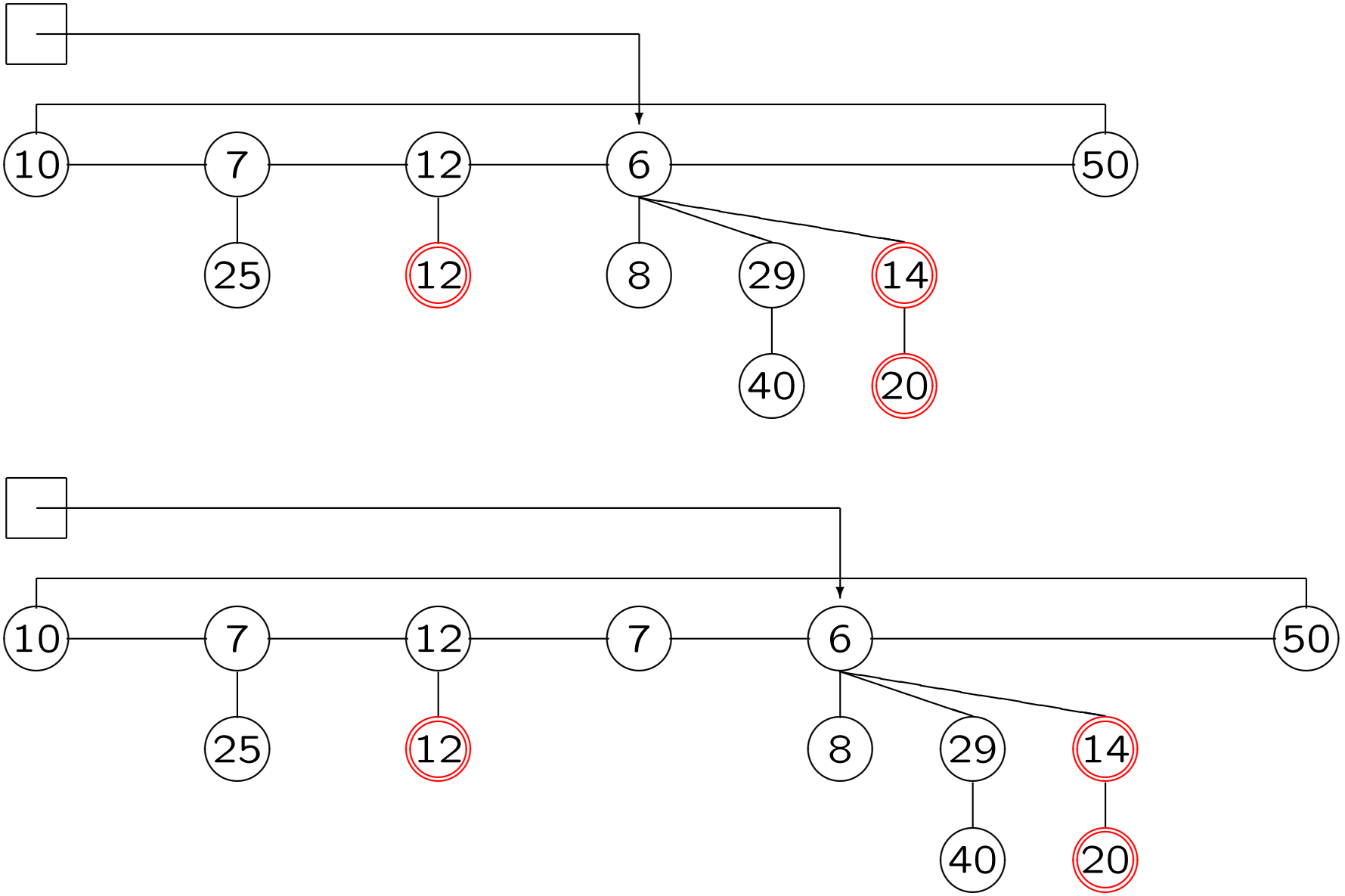
```
// Returns true iff the queue contains no entries.
public boolean isEmpty( )
{
    return min == null;
}

// Returns the number of entries in the priority queue.
public int size( )
{
    return currentSize;
}

// Returns an entry with the smallest key in the queue.
// Pre-condition: the queue is not empty.
public Entry<K,V> minEntry( )
{
    return min.getEntry();
}
```

Inserção de uma Entrada
(fila com n entradas)

- FibNode<K,V> **insert**(K key, V value)
  - Cria-se uma árvore t com a entrada (key, value) — $\Theta(1)$.
  - Insere-se t na fila (por exemplo, à esquerda de **min**) — $\Theta(1)$.
  - Atualiza-se **min**, se key for menor que a chave em **min** — $\Theta(1)$.
  - Retorna-se t — $\Theta(1)$.

Complexidade: $\Theta(1)$ em todos os casos.

Inserir 7



Classe **Interna** Fila de Fibonacci

```
// Inserts the entry (key, value) in the queue and
// returns the node which contains that entry.
public FibNode<K,V> insert( K key, V value )
{
    FibNode<K,V> newTree = new FibNode<K,V>(key, value);
    if ( this.isEmpty() )
        min = newTree;
    else
    {
        this.insertTree(min, newTree);
        // Update min.
        if ( key.compareTo( min.getKey() ) < 0 )
            min = newTree;
    }
    currentSize++;
    return newTree;
}
```

Classe **Interna** Fila de Fibonacci

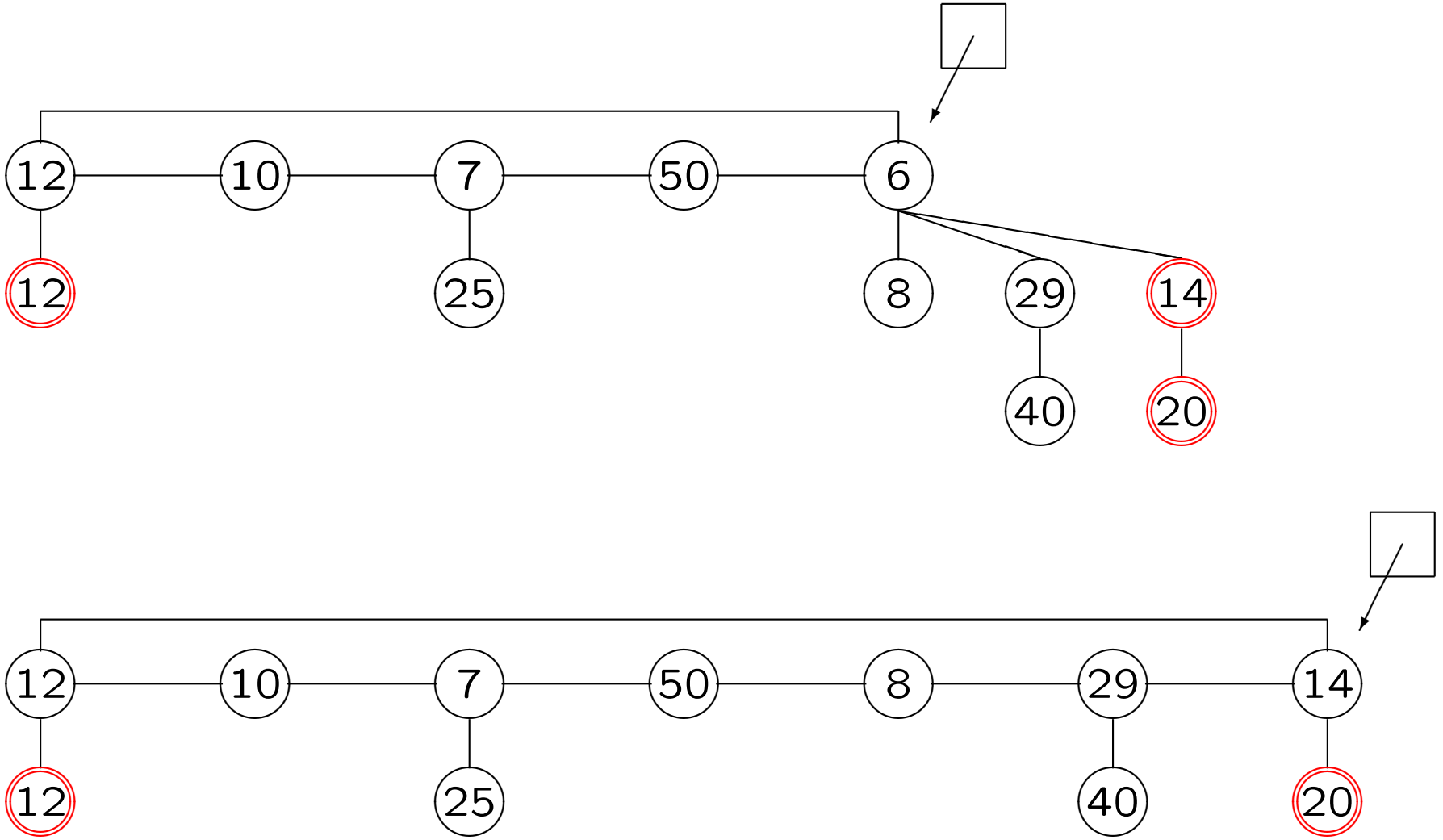
```
// Inserts the specified tree in the list,
// to the left of the specified head.
// Pre-condition: the list is not empty.
protected void insertTree( FibNode<K,V> head,
    FibNode<K,V> newTree )
{
    FibNode<K,V> leftTree = head.getLeftSibling();
    newTree.setLeftSibling(leftTree);
    newTree.setRightSibling(head);
    leftTree.setRightSibling(newTree);
    head.setLeftSibling(newTree);
}
```

Remoção de uma Entrada com Chave Mínima
(fila com n entradas)

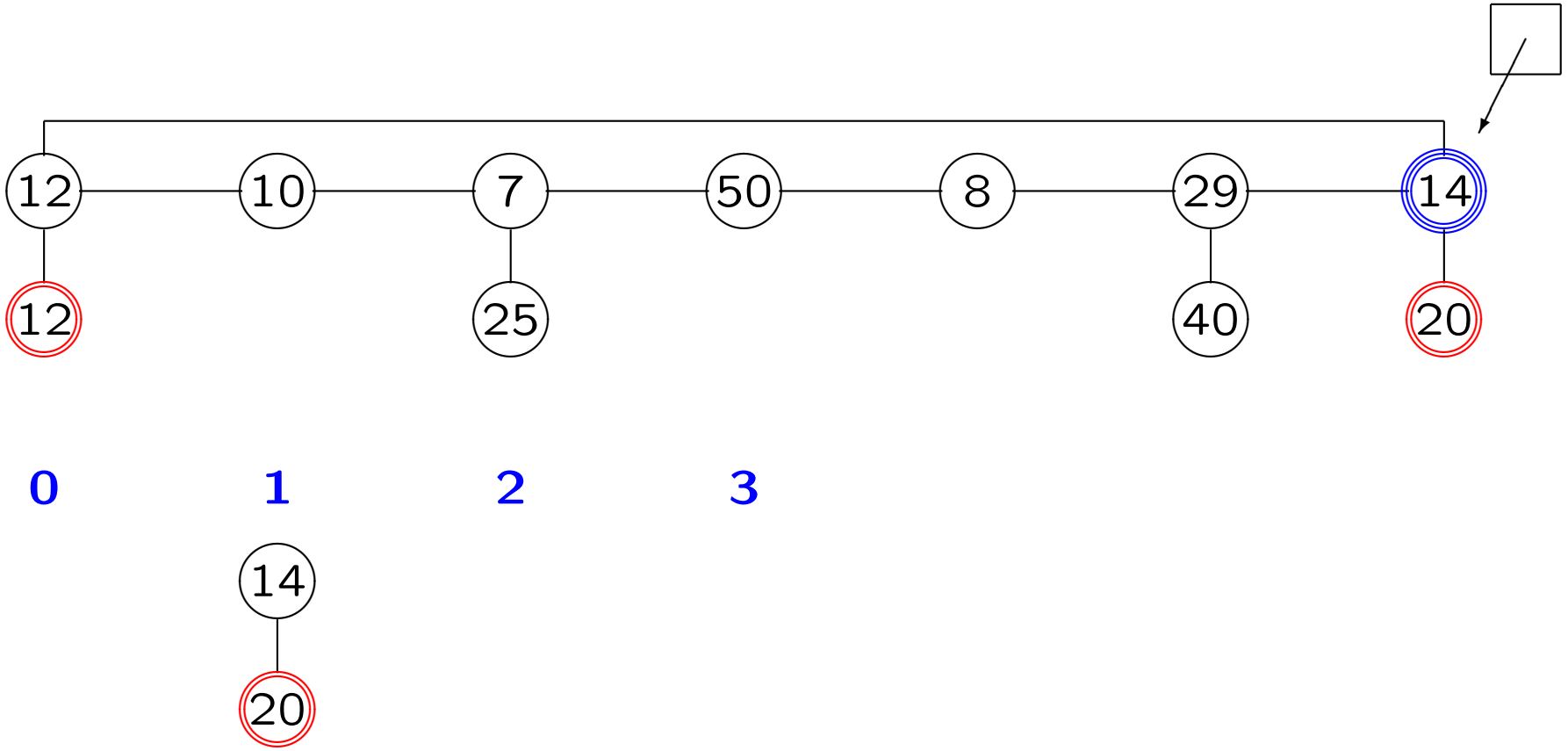
- Entry<K,V> **removeMin**( )
  - Guarda-se a entrada e guardada em **min** — $\Theta(1)$.
  - Para cada filho de **min**, coloca-se a marca a **false**, coloca-se o pai a **null** e insere-se na fila — $\Theta(d)$.
  - Retira-se a árvore **min** da fila, ficando **min** a apontar para uma árvore qualquer (ou a **null**, se a fila ficar vazia) — $\Theta(1)$.
  - Consolida-se a fila — **this**.consolidate().
  - Retorna-se e — $\Theta(1)$.

Complexidade: d + "consolidação", onde d é o grau de **min**.

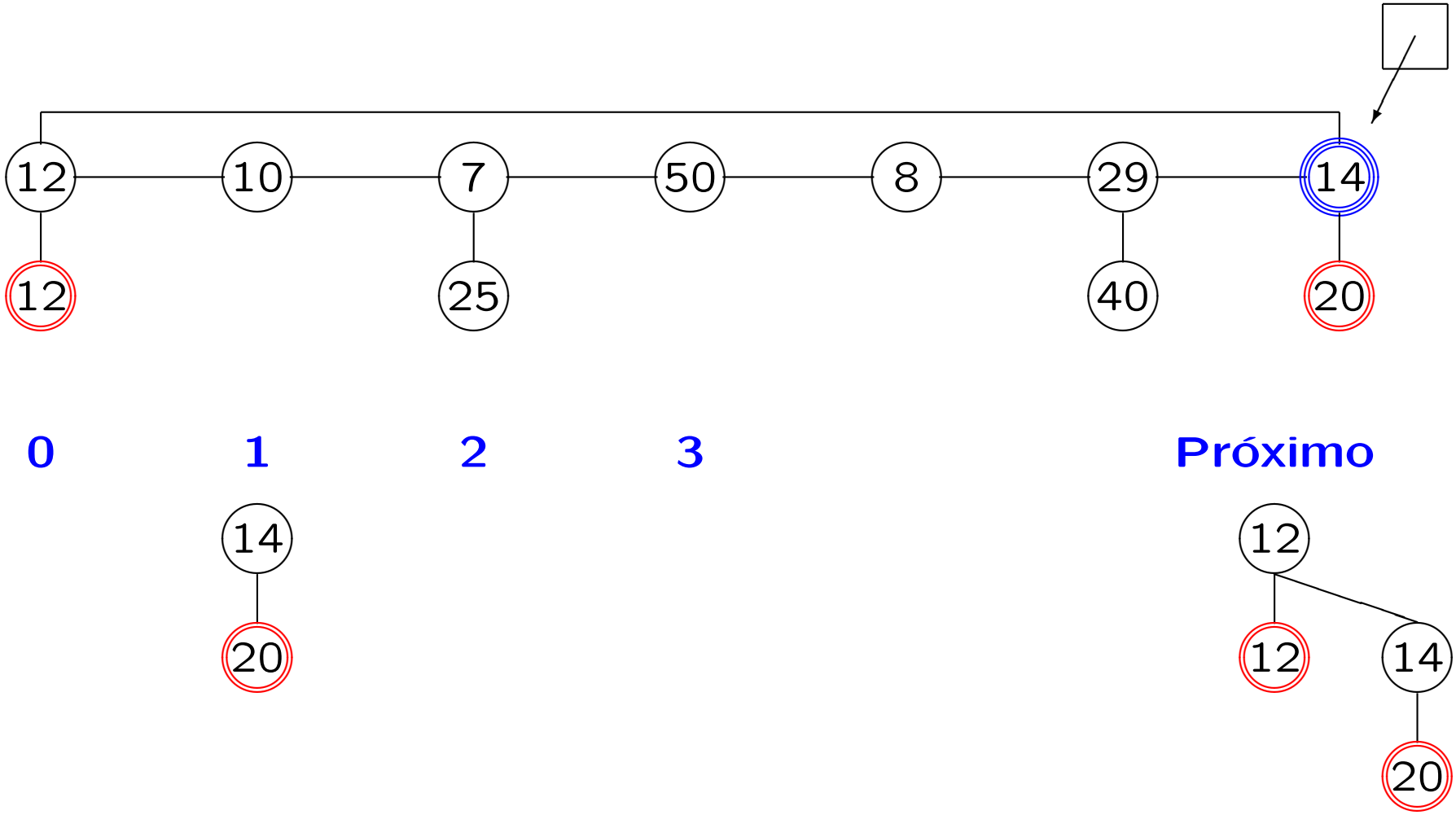
Remover Entrada com Chave Mínima (1)



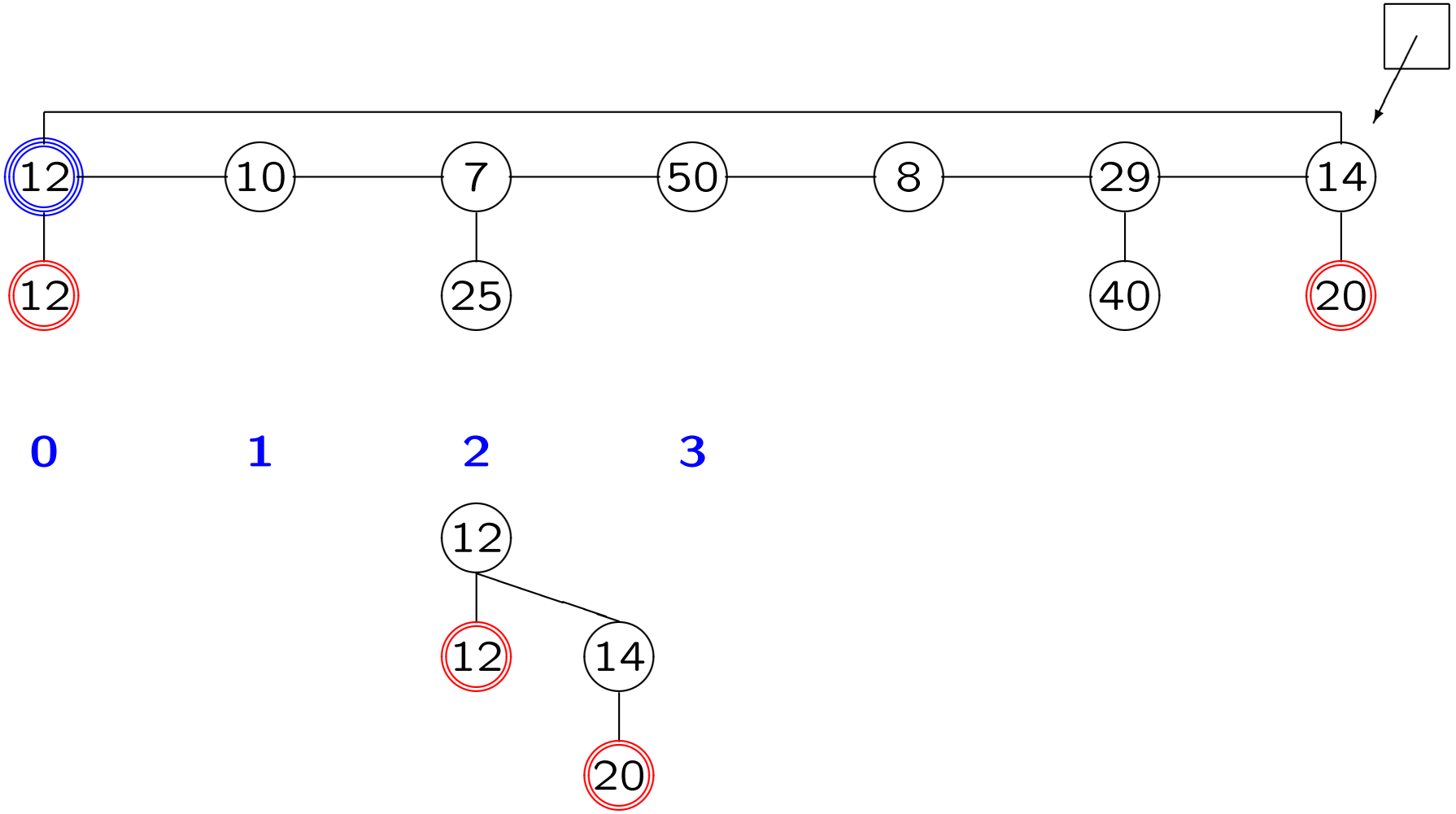
Remover Entrada com Chave Mínima (2)



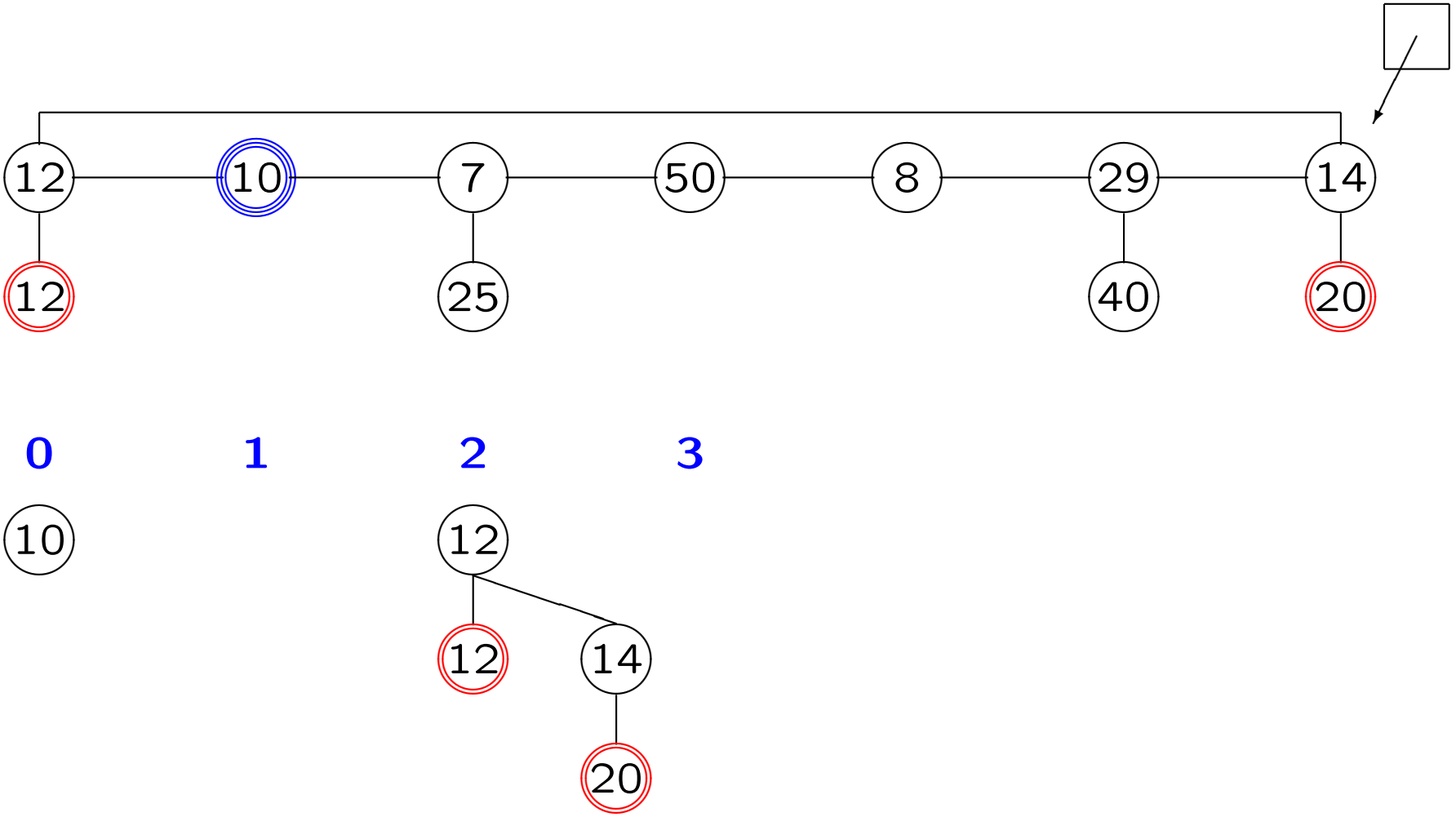
Remover Entrada com Chave Mínima (2)



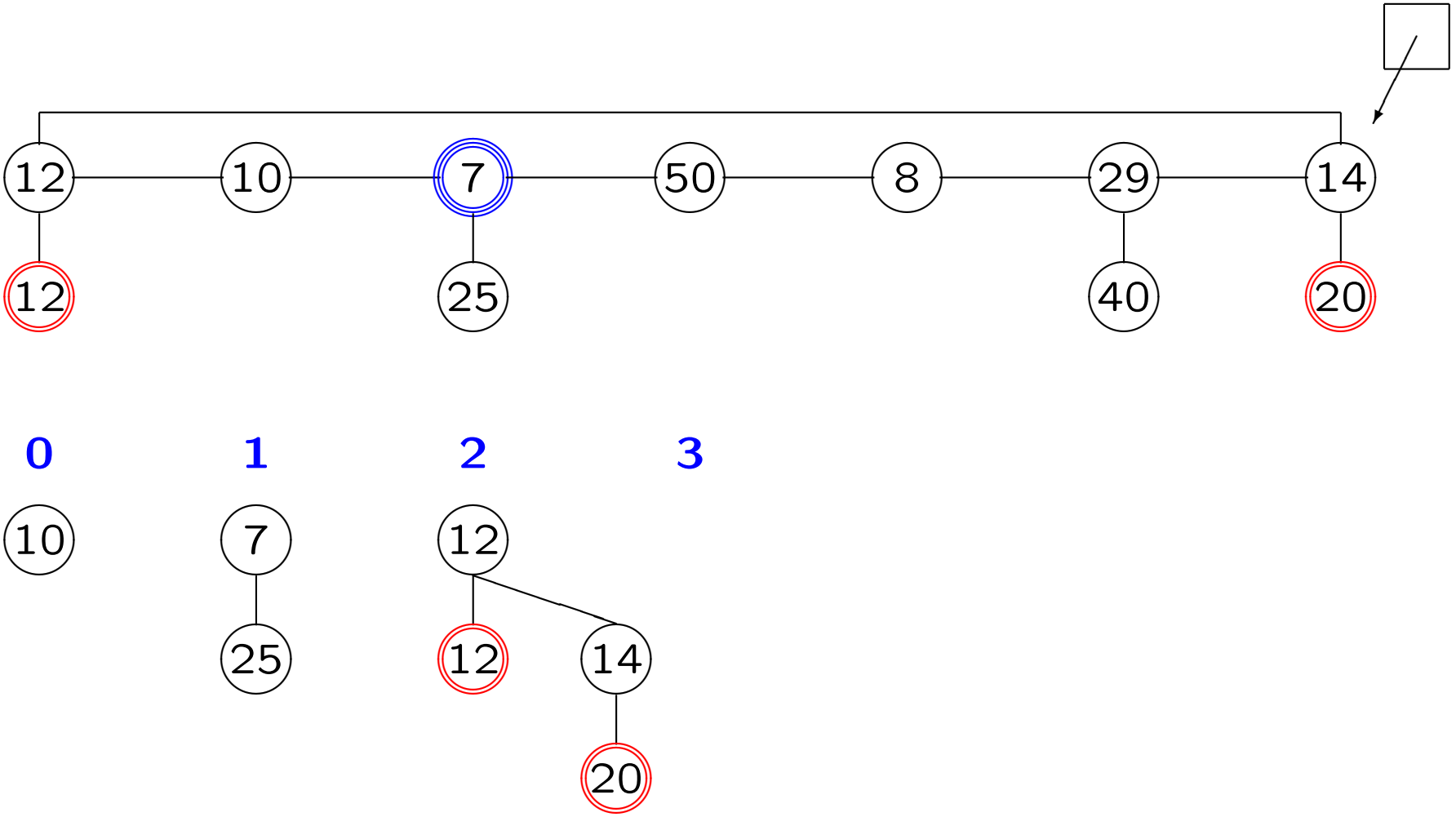
Remover Entrada com Chave Mínima (3)



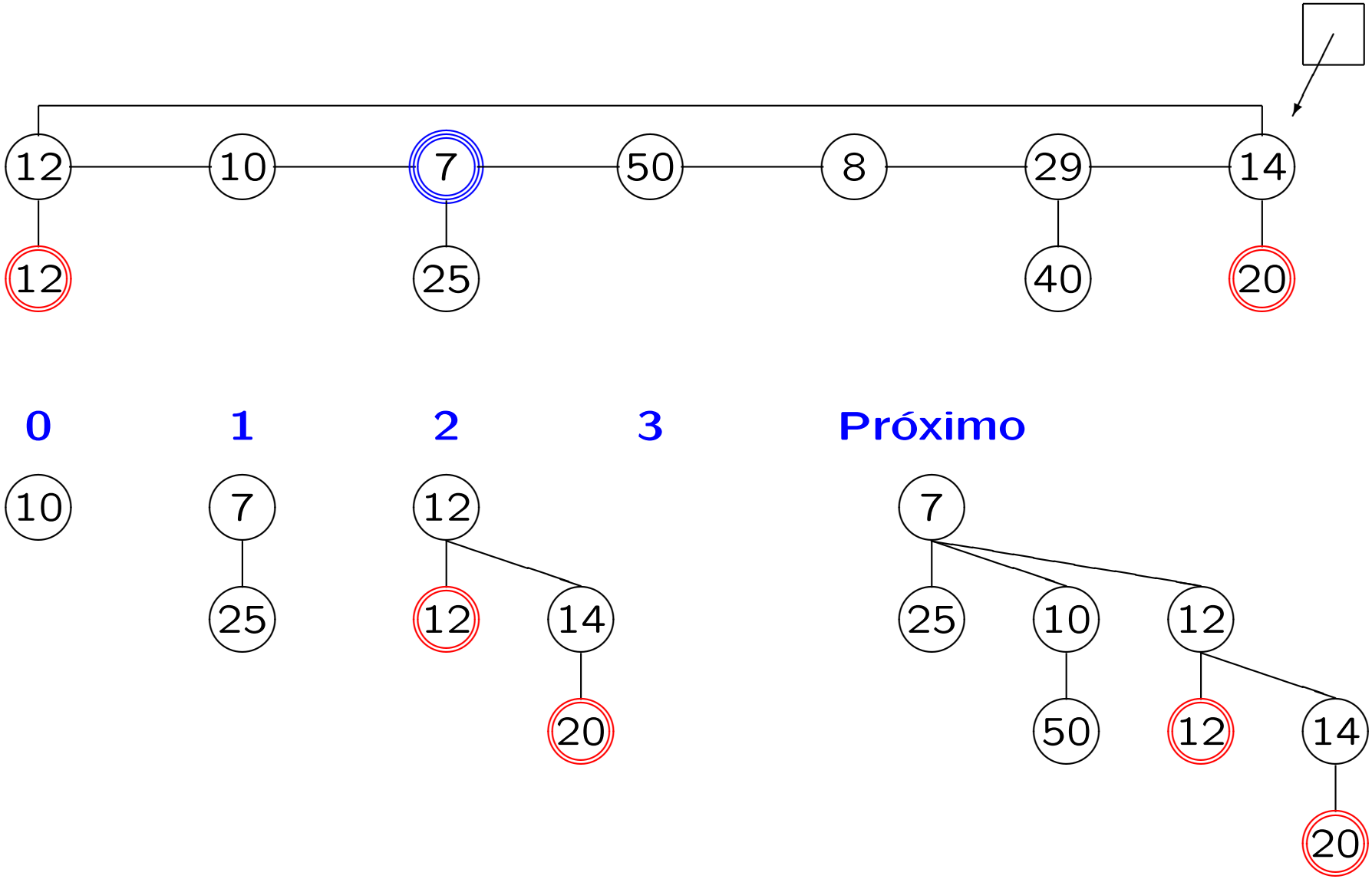
Remover Entrada com Chave Mínima (4)



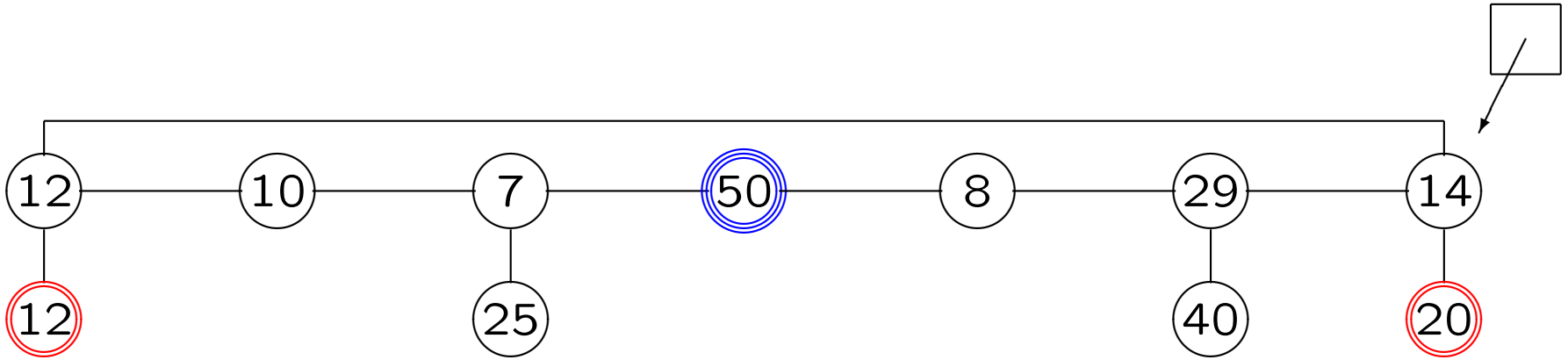
Remover Entrada com Chave Mínima (5)



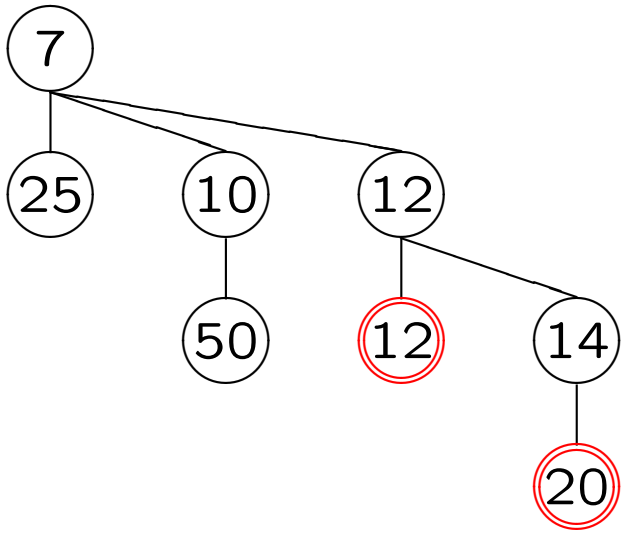
Remover Entrada com Chave Mínima (5)



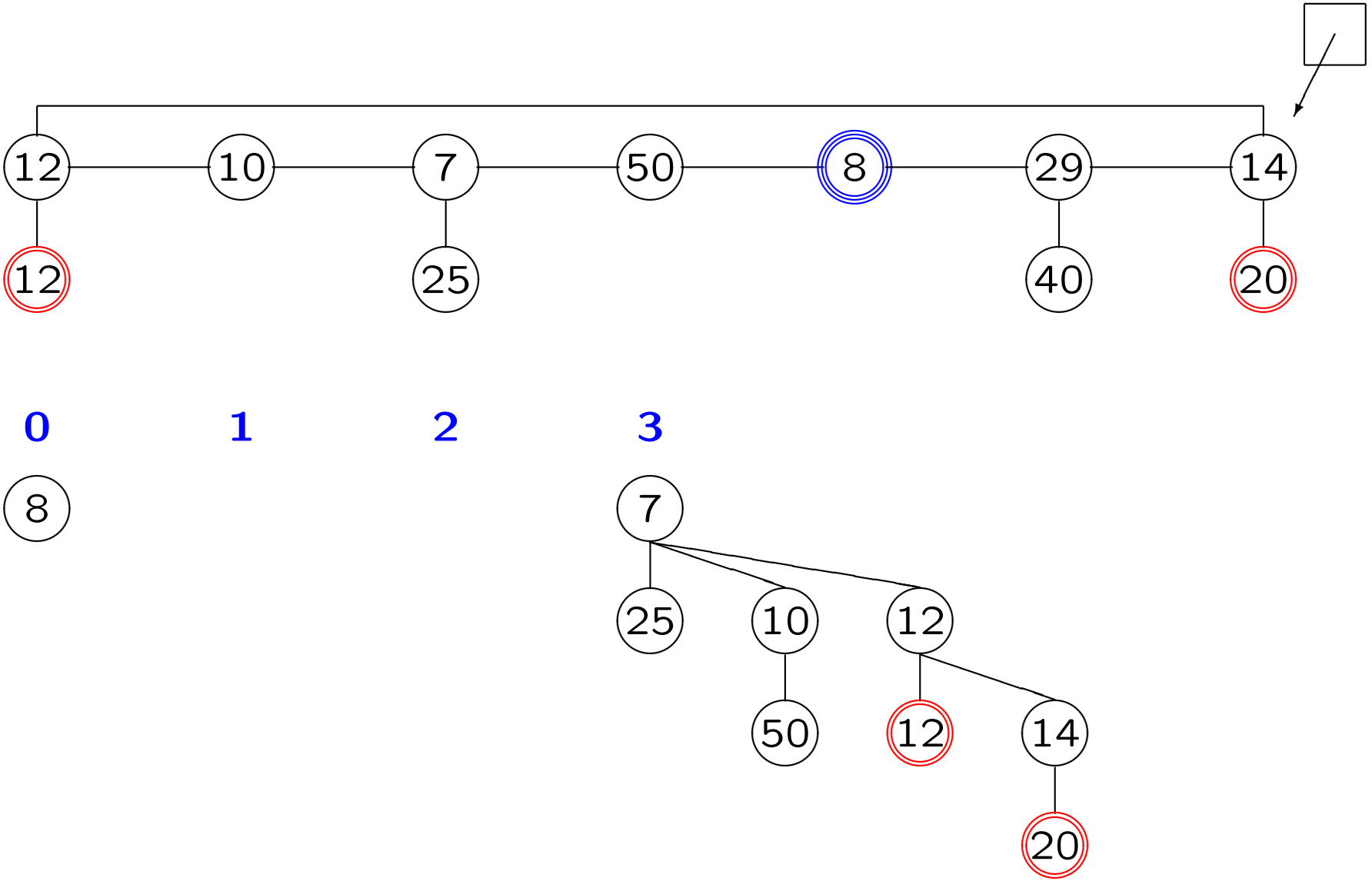
Remover Entrada com Chave Mínima (6)



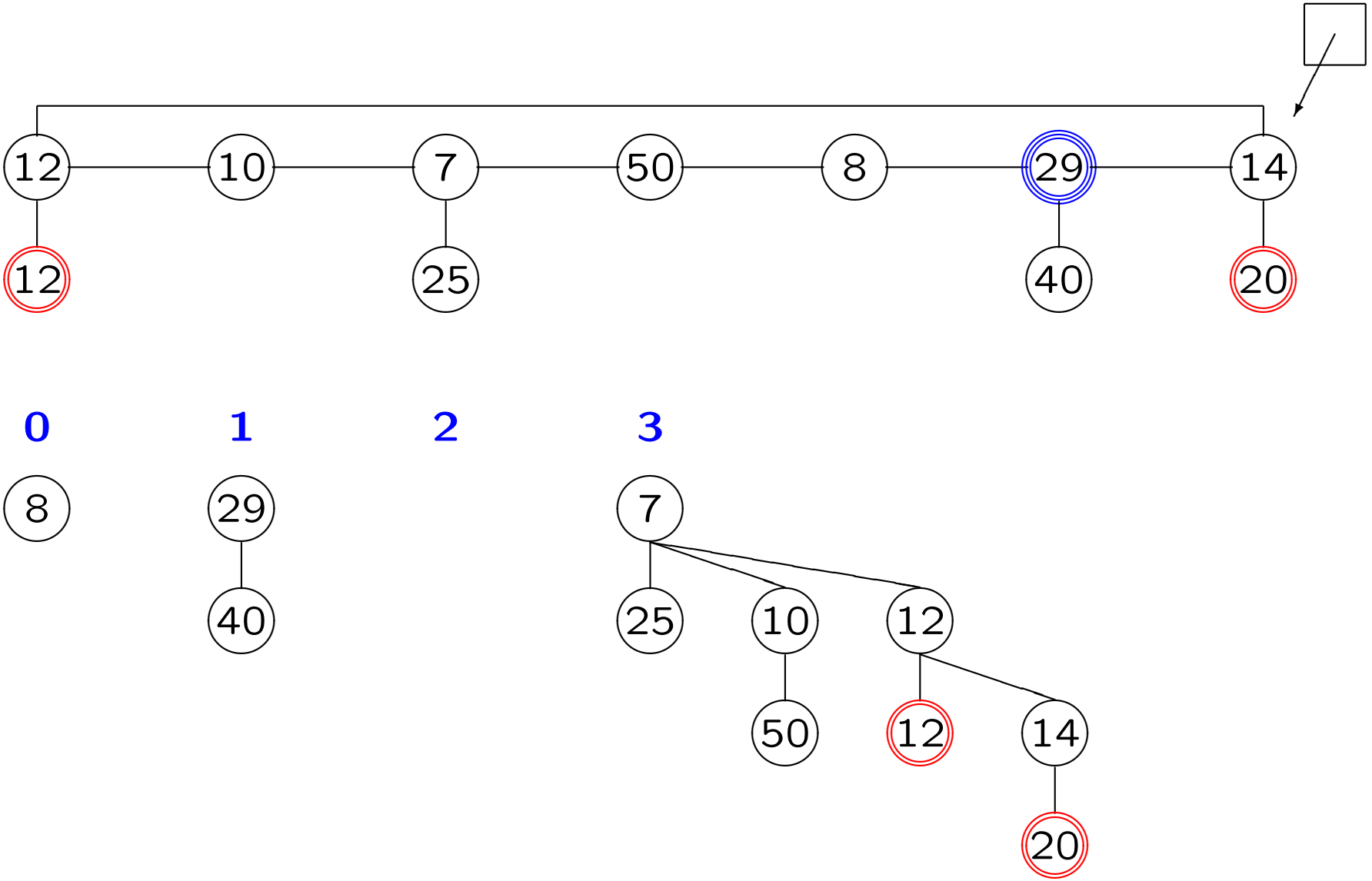
0 **1** **2** **3**



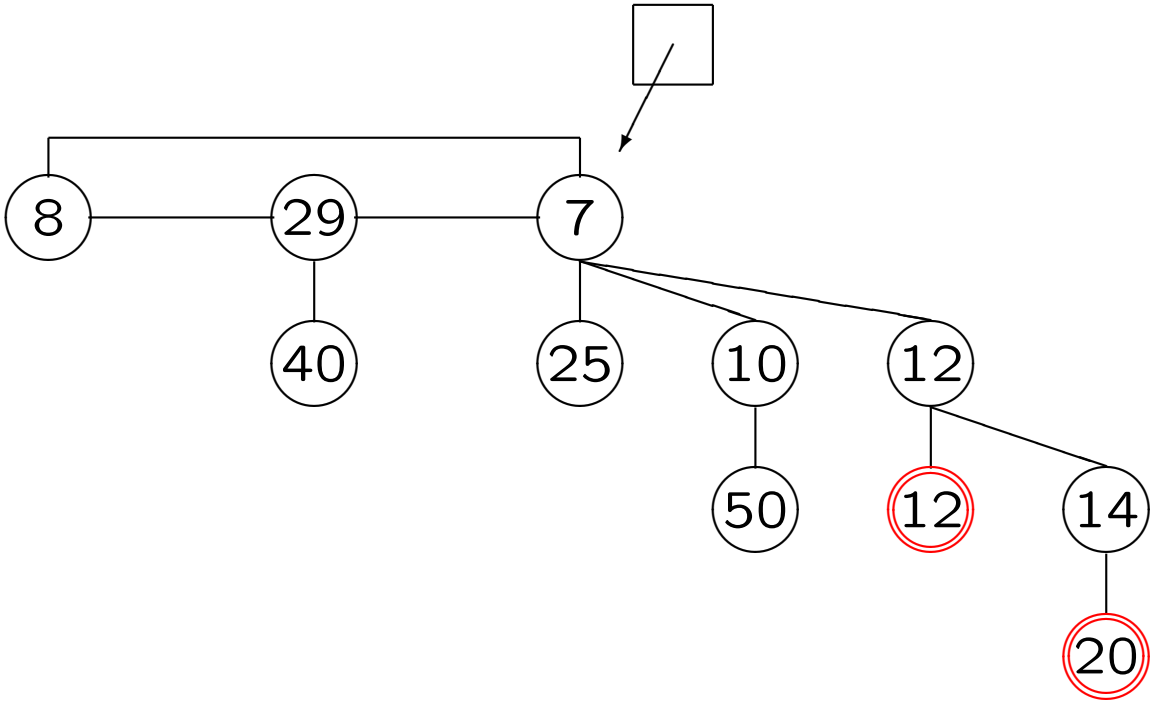
Remover Entrada com Chave Mínima (7)



Remover Entrada com Chave Mínima (8)



Remover Entrada com Chave Mínima (9)



Classe **Interna** Fila de Fibonacci

```
// Removes an entry with the smallest key from the queue
// and returns that entry.
// Pre-condition: the queue is not empty.
public Entry<K,V> removeMin( )
{
    Entry<K,V> entry = min.getEntry();
    this.insertChildren( min.getChild() );
    this.removeMinTree();
    if ( min != null )
        this.consolidate();
    currentSize--;
    return entry;
}
```

```
protected void insertChildren( FibNode<K,V> firstChild )
{
    if ( firstChild != null )
    {
        FibNode<K,V> tree = firstChild;
        do {
                FibNode<K,V> nextTree = tree.getRightSibling();
                tree.setParent(null);
                tree.unmark();
                this.insertTree(min, tree);
                tree = nextTree;
            }
        while ( tree != firstChild );
    }
}
```

Classe **Interna** Fila de Fibonacci

```
// Rebuilds the queue in such a way that
// every resulting tree has a distinct degree.
// Pre-condition: the queue is not empty.
protected void consolidate( )
{
    FibNode<K,V>[] trees = this.buildTrees();

    this.rebuildQueue(trees);
}
```

```
// Returns an array with the new queue trees (indexed by degree).
// Pre-condition: the queue is not empty.
@SuppressWarnings("unchecked")
protected FibNode<K,V>[] buildTrees( )
{
    int capacity = this.maxDegree() + 1;
    // Compiler would give a warning.
    FibNode<K,V>[] trees = (FibNode<K,V>[]) new FibNode[capacity];
    FibNode<K,V> currTree = min;
    do {
            ..........................................
       }
    while ( currTree != min );
    return trees;
}
```

```
FibNode<K,V> currTree = min;
do {
        int currDegree = currTree.getDegree();
        FibNode<K,V> nextTree = currTree.getRightSibling();
        while ( trees[currDegree] != null )
        {
            currTree = this.linkTrees(currTree, trees[currDegree]);
            trees[currDegree] = null;
            currDegree++;
        }
        trees[currDegree] = currTree;
        currTree = nextTree;
    }
while ( currTree != min );
```

Complexidade da Consolidação

Sejam:

- t o número de árvores na fila;
- d o grau da árvore **min**; e
- $d_{\max}$ o valor retornado pelo método maxDegree (que é o maior grau que uma árvore da fila pode ter).

Então:

- Ciclo exterior (do-while) de buildTrees: cada passo é constante e o número de passos é $t + d - 1$. $\Theta(t + d)$
- Ciclo interior (while) de buildTrees: cada passo é constante e o número de passos não excede $t + d - 2$. $O(t + d)$
- Complexidade de rebuildQueue: cada passo é constante e o número de passos é $d_{\max} + 1$. $\Theta(d_{\max})$

Complexidade Total: $\Theta(t + d_{\max})$

Complexidade de removeMin

(fila com n entradas)

Sejam:

- t o número de árvores na fila;
- d o grau da árvore **min**; e
- $d_{\max}$ o valor retornado pelo método maxDegree (que é o maior grau que uma árvore da fila pode ter).

Então:

- Complexidade de insertChildren: $\Theta(d)$
- Complexidade de removeMinTree: $\Theta(1)$
- Complexidade de consolidate: $\Theta(t + d_{\max})$

Complexidade Total: $\Theta(t + d_{\max})$

Como t pode ser n e $d_{\max} < n$, a complexidade de removeMin é $O(n)$.

Observações

Se só se efetuarem operações de isEmpty, size, minEntry, insert e removeMin, todas as árvores da fila são árvores binomiais.

Nesse caso, o maior grau que uma árvore da fila pode ter é:

$$d_{\max} = \lfloor \log n \rfloor \, .$$

A capacidade do vetor auxiliar de buildTrees deve ser:

$$\lfloor \log n \rfloor + 1 \, .$$

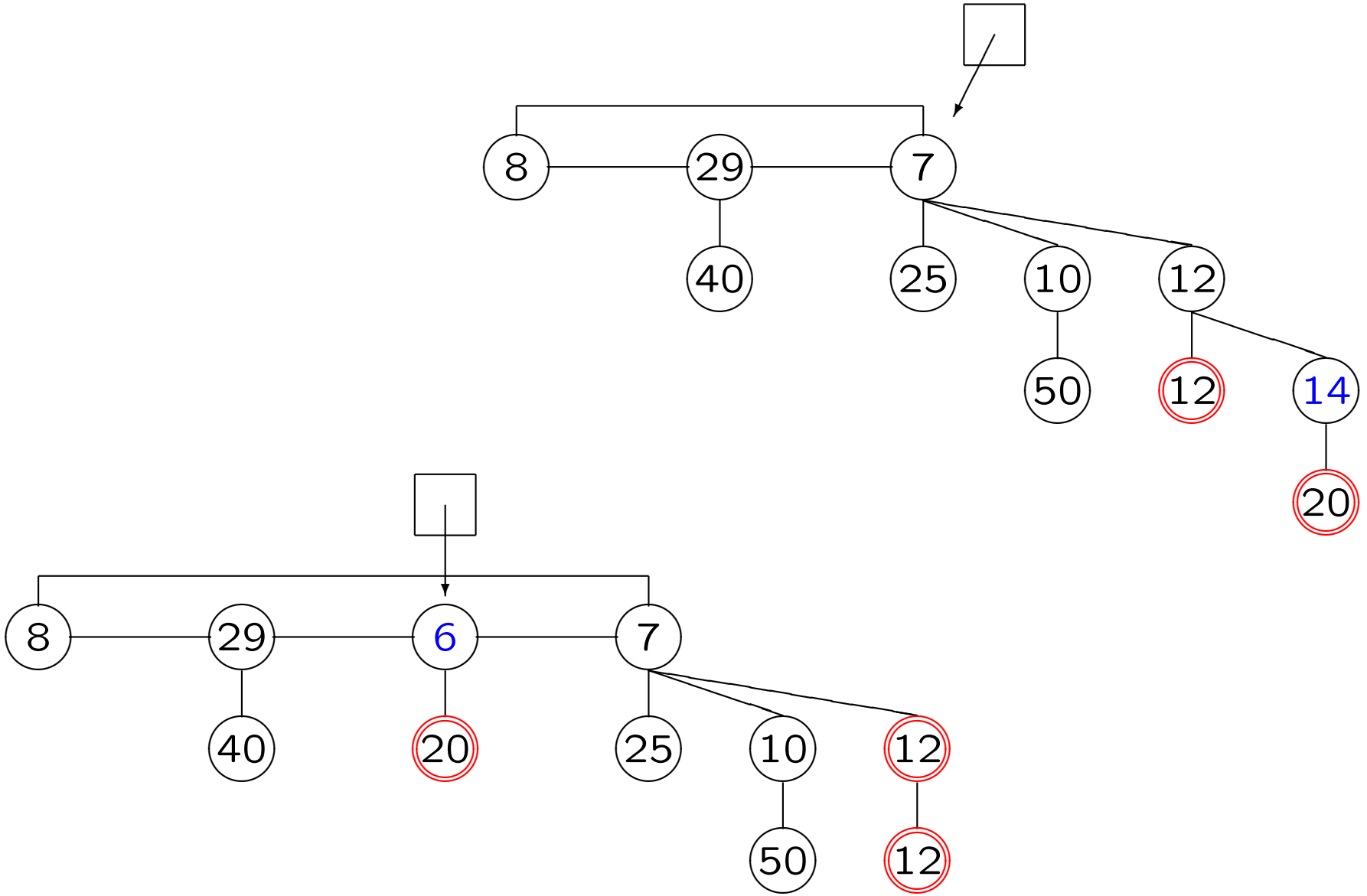
A consolidação constrói uma "fila binomial" (em que a cabeça da lista aponta para uma árvore cuja raiz tem a chave mínima).

Decremento da Chave de um Nó

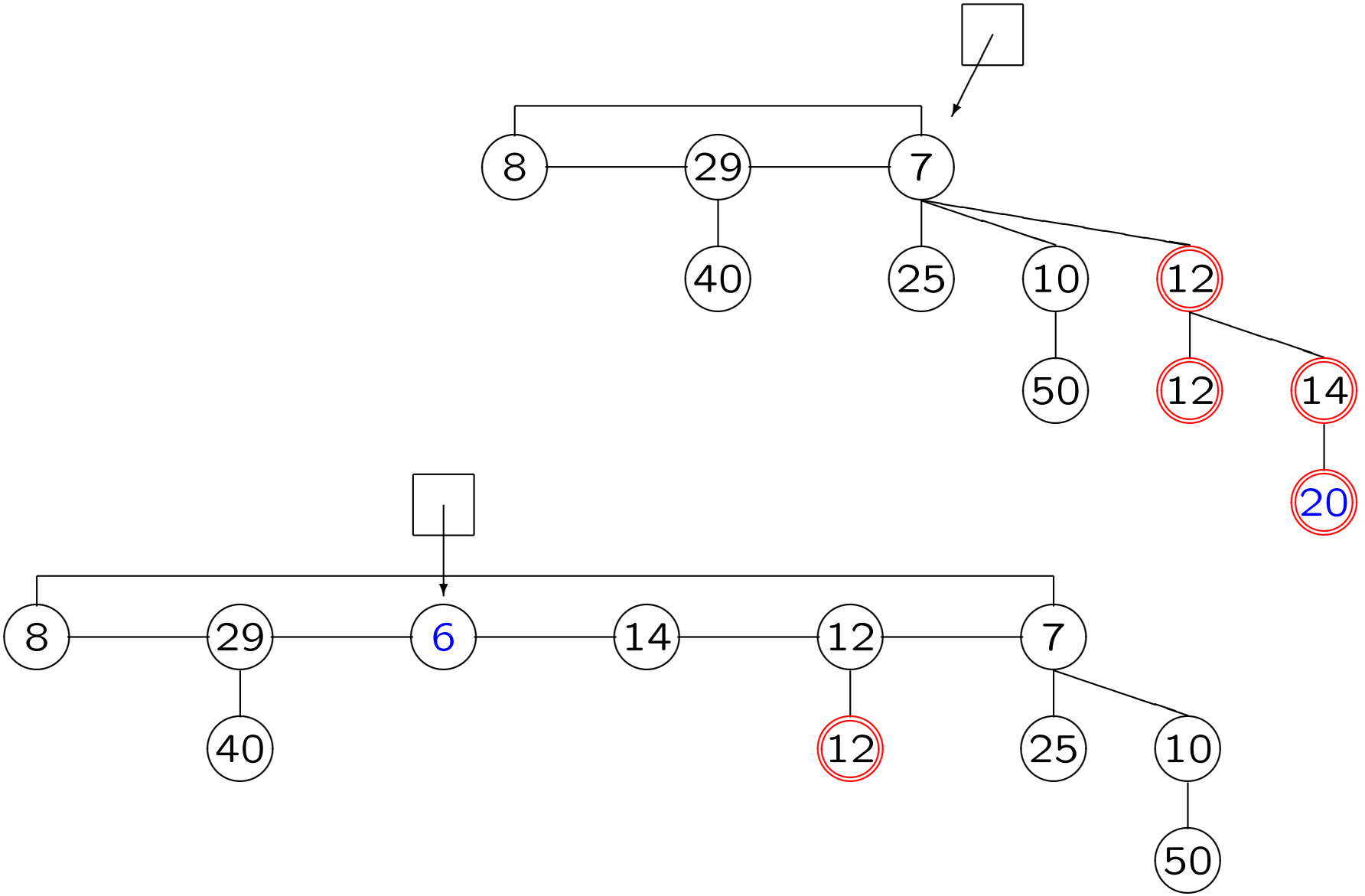
- **void decreaseKey**(FibNode<K,V> node, K newKey)
  1. Se (node é raiz **ou** newKey $\geq$ chave do pai), passar a (3.).
  2. No caso contrário, corta-se node: remove-se node da lista dos filhos do pai; coloca-se o pai de node a **null**; coloca-se a marca de node a **false**; e insere-se node na fila.
     - Se o pai é raiz, passar a (3.)
     - Se o pai não é raiz e não está marcado, coloca-se a marca do pai a **true**. Passar a (3.)
     - Se o pai está marcado, (o pai não é raiz e) corta-se pai.
  3. Atualiza-se **min**, se newKey for menor que a chave em **min**.

  Complexidade: altura da árvore, no pior caso.

Decrementar 14 para 6



Decrementar 20 para 6



Classe **Interna** Fila de Fibonacci

```
// Replaces the key in the specified node by the specified key.
// Pre-condition: the specified key is less than the one in the node.
public void decreaseKey( FibNode<K,V> node, K newKey )
{
    node.setKey(newKey);

    FibNode<K,V> parent = node.getParent();

    if ( parent != null && parent.getKey().compareTo(newKey) > 0 )
    {
        this.cut(node, parent);

        this.cascadingCut(parent);
    }

    // Update min.

    if ( newKey.compareTo( min.getKey() ) < 0 )

        min = node;
}
```

Classe **Interna** Fila de Fibonacci

```
// Cuts the link between the specified node and its parent
// and makes the specified node a queue root.
// The parent's degree is decreased but its mark is not changed.
protected void cut( FibNode<K,V> node, FibNode<K,V> parent )
{
    this.removeChild(parent, node);

    node.setParent(null);

    node.unmark();

    this.insertTree(min, node);
}
```

Classe **Interna** Fila de Fibonacci

```
// Recursively cuts all marked ancestors of the specified node
// (starting with the specified node) until an unmarked node is found.
// Marks that unmarked node, unless it is the root.
protected void cascadingCut( FibNode<K,V> node )
{
    FibNode<K,V> parent = node.getParent();

    if ( parent != null )

        if ( node.isMarked() )
        {
            this.cut(node, parent);

            this.cascadingCut(parent);
        }

        else

            node.mark();
}
```

Altura Máxima de uma Árvore de uma Fila de Fibonacci com n Entradas

$$h \leq n$$

Justificação

- $(n = h = 2)$
  inserir $-1, -2, -3$; remMin.
- $(n = h = 3)$
  inserir $-3, -4, -5$; remMin; decrementar -3 para $-\infty$; remMin.
- $(n = h = 4)$
  inserir $-5, -6, -7$; remMin; decrementar -5 para $-\infty$; remMin.
- $(n = h = 5)$
  inserir $-7, -8, -9$; remMin; decrementar -7 para $-\infty$; remMin.
- .....

Número Mínimo de Nós de uma Árvore com Grau d

Propriedade Fundamental (PF)

Seja x um nó qualquer com grau d (para $d > 0$). Se $y_1, y_2, \ldots, y_d$ forem os filhos de x, pela ordem com que foram ligados a x, então:

- grau$(y_1) \geq 0$; e
- grau$(y_i) \geq i - 2$, para $i = 2, 3, \ldots, d$.

Corolário (Cor)

Se $N(d)$ for o número de nós de uma árvore com grau d (para $d \geq 0$):

$$N(d) \geq \text{Fibonacci}(d+2)$$

$$(\forall d \geq 0) \quad N(d) \geq \text{Fibonacci}(d+2)$$

Demonstração

$$
\begin{array}{lllll}
(d=0) & N(0) & = & 1 & = \text{Fibonacci}(2) \\
(d=1) & N(1) & \geq & 1+1 & = \text{Fibonacci}(3) \\
(d \geq 2) & N(d) & \overset{\text{(PF)}}{\geq} & \underbrace{1}_{\text{raiz}} + \underbrace{1}_{y_1} + \underbrace{\sum_{i=2}^{d} N(i-2)}_{y_2\ y_3 \cdots y_d} & \\
 & & \overset{\text{H.I.}}{\geq} & 2 + \sum_{i=2}^{d} \text{Fibonacci}(i) & \\
 & & = & 1 + \sum_{i=0}^{d} \text{Fibonacci}(i) & \\
 & & \overset{\text{(L1)}}{=} & \text{Fibonacci}(d+2) &
\end{array}
$$

Grau Máximo de uma Árvore de uma Fila de Fibonacci com n Entradas

Seja d o grau de uma árvore qualquer da fila. Então:

$$n \geq N(d) \overset{\text{(Cor)}}{\geq} \text{Fibonacci}(d+2) \overset{\text{(L2)}}{\geq} \phi^d$$

onde

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618\,.$$

Portanto,

$d \leq \log_\phi n$ e, porque d é um número inteiro, $d \leq \lfloor \log_\phi n \rfloor$.

Conclusão:

$$\boxed{d_{\max} = \lfloor \log_\phi n \rfloor}$$

Classe **Interna** Fila de Fibonacci

$$d_{\max} = \lfloor \log_{\phi} n \rfloor = \left\lfloor \frac{\ln n}{\ln \phi} \right\rfloor$$

```
public static final double GOLDEN_RATIO = 1.618;


// Returns the maximum possible degree of a tree in the queue.
protected int maxDegree( )
{
  return (int) ( Math.log(currentSize) / Math.log(GOLDEN_RATIO) );
}
```

Lema 1 (L1)

$$(\forall n \geq 0) \quad \text{Fibonacci}(n+2) = 1 + \sum_{i=0}^{n} \text{Fibonacci}(i)$$

Demonstração

$$(n = 0) \qquad \text{Fib}(2) \quad = \quad 1 \quad = \quad 1 + \sum_{i=0}^{0} \text{Fib}(i)$$

$$(n \geq 1) \qquad \text{Fib}(n+2) \overset{\text{Def}}{=} \text{Fib}(n) + \text{Fib}(n+1)$$

$$\overset{\text{H.I.}}{=} \text{Fib}(n) + 1 + \sum_{i=0}^{n-1} \text{Fib}(i)$$

$$= \quad 1 + \sum_{i=0}^{n} \text{Fib}(i)$$

Lema 2 (L2)

$$(\forall n \geq 0) \quad \text{Fibonacci}(n+2) \geq \phi^n$$

Demonstração

$$(n = 0) \qquad \text{Fib}(2) \quad = \quad 1 \quad = \quad \phi^0$$

$$(n = 1) \qquad \text{Fib}(3) \quad = \quad 2 \quad \geq \quad \phi^1 \ (\approx 1.618)$$

$$(n \geq 2) \qquad \text{Fib}(n+2) \overset{\text{Def}}{=} \text{Fib}(n+1) + \text{Fib}(n)$$

$$\overset{\text{H.I.}}{\geq} \quad \phi^{n-1} + \phi^{n-2} \quad = \quad \phi^{n-2}\,(\phi + 1)$$

$$\overset{\text{(L3)}}{=} \quad \phi^{n-2}\,\phi^2 \quad = \quad \phi^n$$

Lema 3 (L3)

$$\phi^2 = \phi + 1$$

Demonstração

$$x^2 = x + 1$$

$$x^2 - x - 1 = 0$$

$$x = \frac{1 \pm \sqrt{1 - 4 \times 1 \times (-1)}}{2}$$

$$x = \frac{1 \pm \sqrt{5}}{2}$$

$$x = \phi \quad \vee \quad x = \widehat{\phi}$$

Complexidades da Fila com Prioridade Adaptável (n entradas)

	Heap	Fila Binomial	Fila de Fibonacci
isEmpty	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
size	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
minEntry	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
insert	$O(\log n)$	$O(\log n)$	$\Theta(1)$
removeMin	$O(\log n)$	$O(\log n)$	$O(n)$
decreaseKey	$O(\log n)$	$O(\log n)$	$O(n)$

Complexidades Amortizadas

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q.

$$\Phi(Q) \;=\; t_Q + 2\, m_Q\,.$$

A função Φ é **válida**:

(P1) $\Phi(Q_0) = 0$, onde Q_0 é a fila de Fibonacci (inicial) vazia.

(P2) $\Phi(Q) \geq 0$.

Portanto, o custo total amortizado nunca será inferior ao custo total real.

O custo amortizado da operação i é:

$$\widehat{c}_i \;=\; c_i + \Phi(Q_i) - \Phi(Q_{i-1}).$$

Operações Simples

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q.

$$\Phi(Q) = t_Q + 2\, m_Q\,.$$

Operação	**Custo Real** c_i	**Dif. de Potencial** $\Phi(Q_i) - \Phi(Q_{i-1})$	**Custo Amortizado** $\widehat{c}_i = c_i + \Delta\Phi$
isEmpty	1	0	$O(1)$
size	1	0	$O(1)$
minEntry	1	0	$O(1)$
insert	1	1	$O(1)$

Complexidade Amortizada de removeMin

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q.

$$\Phi(Q) \;=\; t_Q + 2\,m_Q\,.$$

Operação	**Custo Real** c_i	**Dif. de Potencial** $\Phi(Q_i) - \Phi(Q_{i-1})$	**Custo Amortizado** $\hat{c}_i = c_i + \Delta\Phi$
removeMin	$t + d_{\max}$	$\leq d_{\max} + 1 - t$	$O(d_{\max}) = O(\log_\phi(n))$

- $t' \leq d_{\max} + 1$, porque as árvores têm graus distintos após a consolidação.
- $m' \leq m$, porque os filhos marcados de **min** são desmarcados.

Complexidade Amortizada de decreaseKey

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q.

$$\Phi(Q) \;=\; t_Q + 2\,m_Q\,.$$

Operação	**Custo Real** c_i	**Dif. de Potencial** $\Phi(Q_i) - \Phi(Q_{i-1})$	**Custo Amortizado** $\widehat{c}_i = c_i + \Delta\Phi$
decreaseKey	$1 + k$	$\leq 4 - k$	$O(1)$

(k é o número de cortes efectuados.)

- $t' = t + k$.
- $m' \leq m - k + 2$, porque o nó inicial poderia não estar marcado e pode-se marcar um novo nó.

Complexidades da Fila com Prioridade Adaptável no Pior Caso e Amortizadas (n entradas)

	Heap	Fila Binomial	Fila de Fibonacci	
isEmpty	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(1)$
size	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(1)$
minEntry	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$O(1)$
insert	$O(\log n)$	$O(\log n)$	$\Theta(1)$	$O(1)$
removeMin	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(\log n)$	$O(n)$	$O(1)$

Complexidades dos Algoritmos Prim e Dijkstra

Grafo em vetor de listas de adjacências com Fila de Fibonacci (e Vetor)

criar fila	$\Theta(\|V\|)$
inicializar 2 vetores	$\Theta(\|V\|)$
inserir origem na fila	$\Theta(1)$
($\|V\|$ ou $\leq \|V\|$) remover mínimo da fila	$O(\|V\| \times \log \|V\|)$
($\|V\|$ ou $\leq \|V\|$) percorrer sucessores diretos	$O(\|A\|)$
$\leq \|A\|$ inserir na fila ou decrementar chave	$O(\|A\| \times \cancel{\log \|V\|})$
TOTAL Prim ou Dijkstra	$O(\|A\| + \|V\| \times \log \|V\|)$