

ALGORITMOS E ESTRUTURAS DE DADOS

2018/2019

FILA COM PRIORIDADE

Armanda Rodrigues

22 de Novembro de 2018

Lembram-se das Reservas de Livros ?

- O processo de requisição de uma cópia de um livro implica que um leitor da biblioteca reserve o mesmo.
- As reservas de um livro são organizadas como uma fila de espera ?
- Necessária a implementação do TAD Leitor, que será o tipo a guardar na fila de espera de reservas de um livro.



Reservas com prioridade

- O processo de requisição de uma cópia de um livro implica que um leitor da biblioteca reserve o mesmo.
- As reservas de um livro são organizadas por prioridades de Leitor (Categoria), tendo um leitor com maior prioridade um valor menor associado
- Necessária nova implementação do TAD Leitor, que será o tipo a guardar na fila de espera de reservas de um livro. O Leitor necessita de um atributo com a sua prioridade.



Prioridade

- A categoria de um Leitor determina a sua prioridade
 - Categoria S (Sénior) – prioridade 1
 - Categoria C (Criança) – prioridade 2
 - Categoria E (Estudante) – prioridade 3
 - Categoria O (Outros) – prioridade 5



LibrarianBook

```
package library;
```

```
interface LibrarianBook extends Book {
```

```
//removes reader from reservation queue
```

```
//returns next reader with the highest priority
```

```
Reader nextPriorityReader()
```

```
    throws NoReservationsException();
```

```
//adds reader to reservation queue
```

```
void addReservation(Reader reader);
```

```
}
```

Implementação de
Reader deve conter
(e.g.)

- Nome
- Número Leitor
- Categoria
- Prioridade
- Data de inscrição
- Validade

TAD Fila com prioridade - Acesso por prioridade

TAD Fila com Prioridade Organizada por Mínimos

Chaves do Tipo K e Valores do Tipo V

A chave representa a prioridade – não tem de ser única!

```
// Retorna true sse a fila com prioridade estiver vazia.
```

```
boolean vazia( );
```

```
// Retorna uma entrada com chave mínima da fila com prioridade.
```

```
// Pré-condição: a fila com prioridade não está vazia.
```

```
(K,V) mínimo( );
```

```
// Insere a entrada (chave, valor) na fila com prioridade.
```

```
void insere( K chave, V valor );
```

```
// Remove uma entrada com chave mínima da fila com prioridade
```

```
// e retorna essa entrada.
```

```
// Pré-condição: a fila com prioridade não está vazia.
```

```
(K,V) removeMínimo( );
```

Interface Fila com Prioridade Organizada por Mínimos (K,V) (1)

```
package dataStructures;  
  
public interface MinPriorityQueue<K extends Comparable<K>, V> {  
  
    // Returns true iff the priority queue contains no entries.  
    boolean isEmpty( );  
  
    // Returns the number of entries in the priority queue.  
    int size( );  
}
```

Interface Fila com Prioridade

Organizada por Mínimos (K,V) (2)

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

```
// 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 EmptyPriorityQueueException;
```

```
} // End of MinPriorityQueue.
```

Classes de Exceções da Fila com Prioridade

```
package dataStructures;
```

```
public class EmptyPriorityQueueException extends RuntimeException{  
}
```

```
public class FullPriorityQueueException extends RuntimeException{  
}
```

Diagram illustrating a queue implemented with an array. A separate box labeled **posMín** contains the value 1. The array has 6 cells, indexed 0 to 5. The values are 27, 12, 12, 42, and two empty cells. The label **último** is positioned below the array.

0	1	2	3	4	5
12	12	27	42		

último

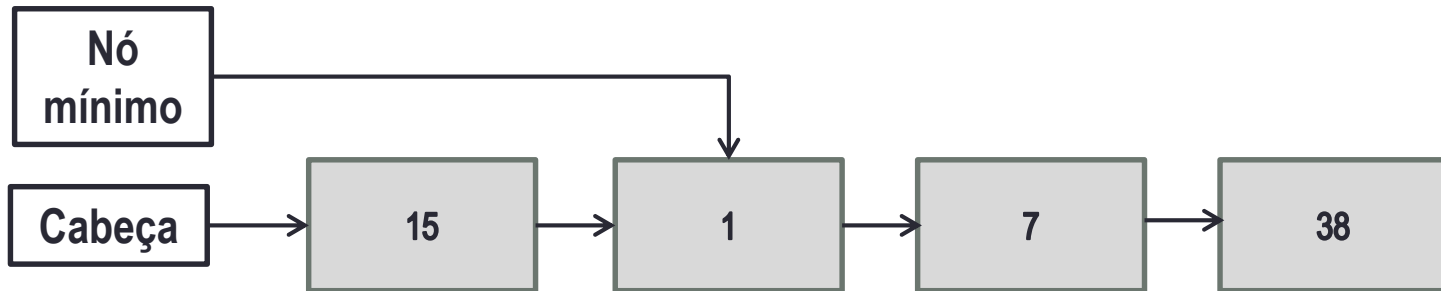
0	1	2	3	4	5
	12	12	27	42	
frente			retaguarda		

Complexidades da Fila com Prioridade em Vetor (com n entradas, no pior caso e no caso esperado)

	Desordenado	Ordenado	Circular Ordenado
new (vazia)	$O(1)$	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$	$O(1)$
size	$O(1)$	$O(1)$	$O(1)$
minEntry	$O(1)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(n)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(n)$

Fila com prioridade em Lista Ligada (n entradas)

Desordenada



Ordenada

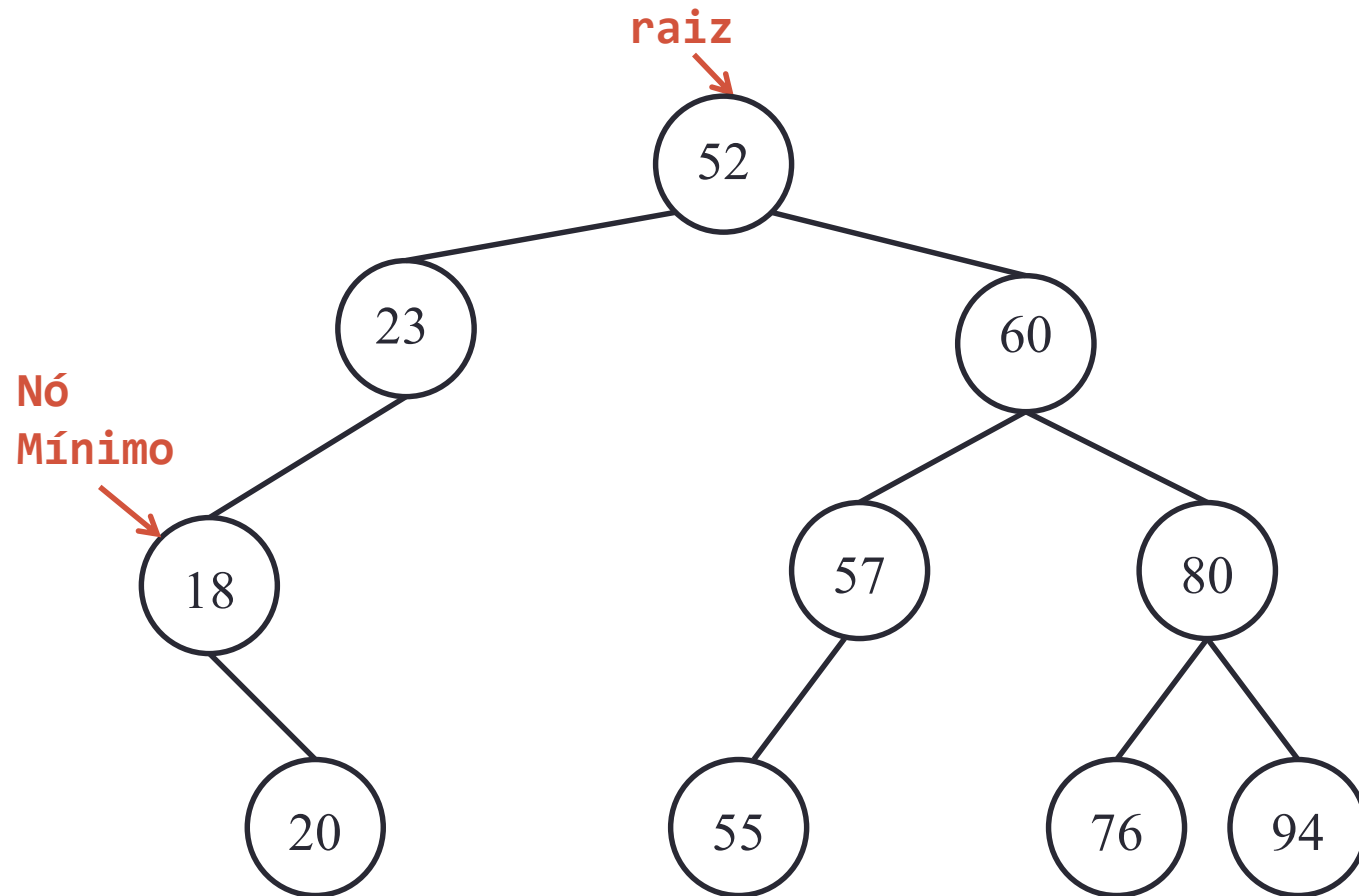


Complexidades da Fila com Prioridade em LL

(com n entradas, no pior caso e no caso esperado)

	Desordenada	Ordenada
new (vazia)	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
size	$O(1)$	$O(1)$
minEntry	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$
insert	$O(1)$	$O(n)$

Fila com Prioridade em Árvore Binária de Pesquisa (n Entradas)



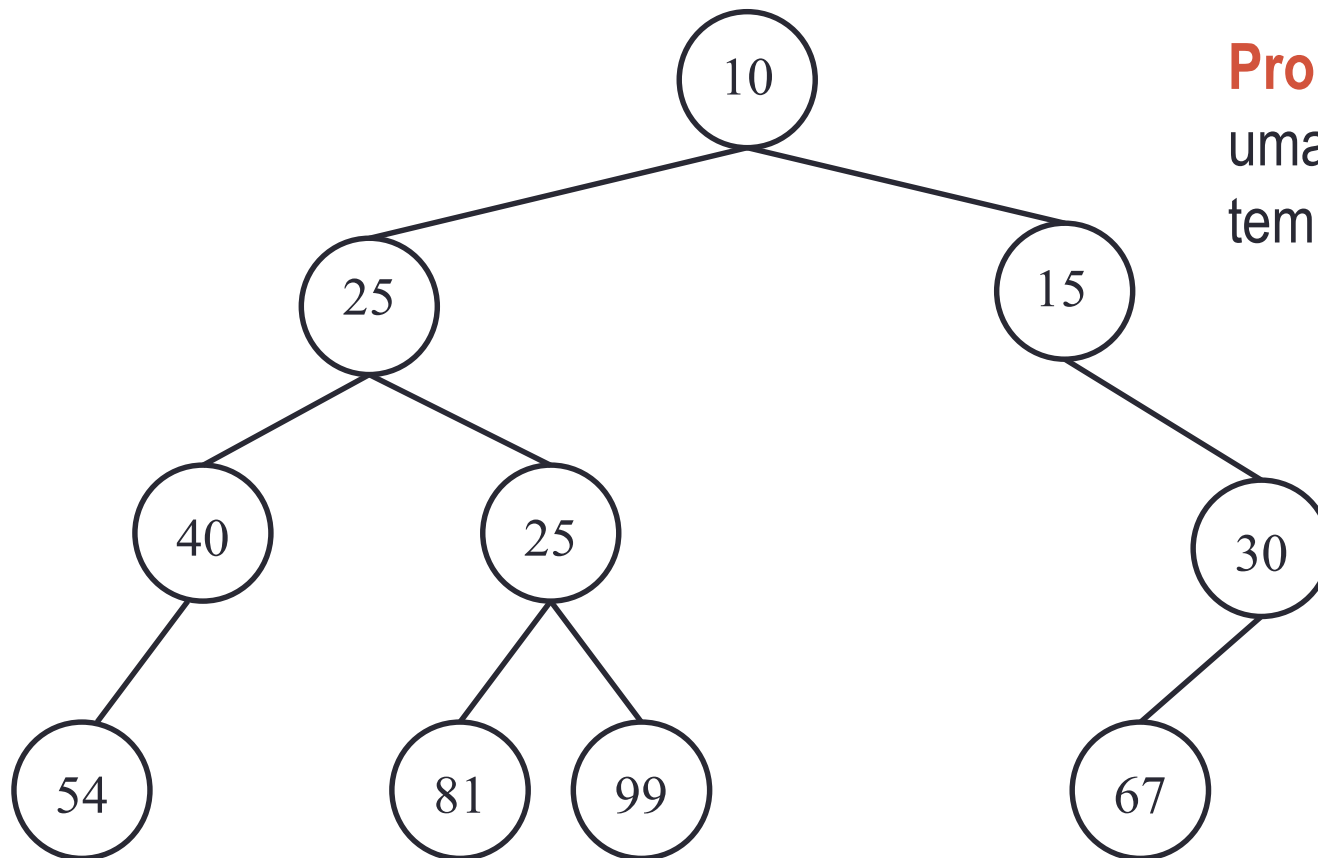
Complexidades da Fila com Prioridade em ABP (com n entradas, no pior caso)

	Sem Restrições	AVL
new (vazia)	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
size	$O(1)$	$O(1)$
minEntry	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(\log n)$
insert	$O(n)$	$O(\log n)$

Árvore com Prioridade por Mínimos

Todo o nó X verifica a propriedade:

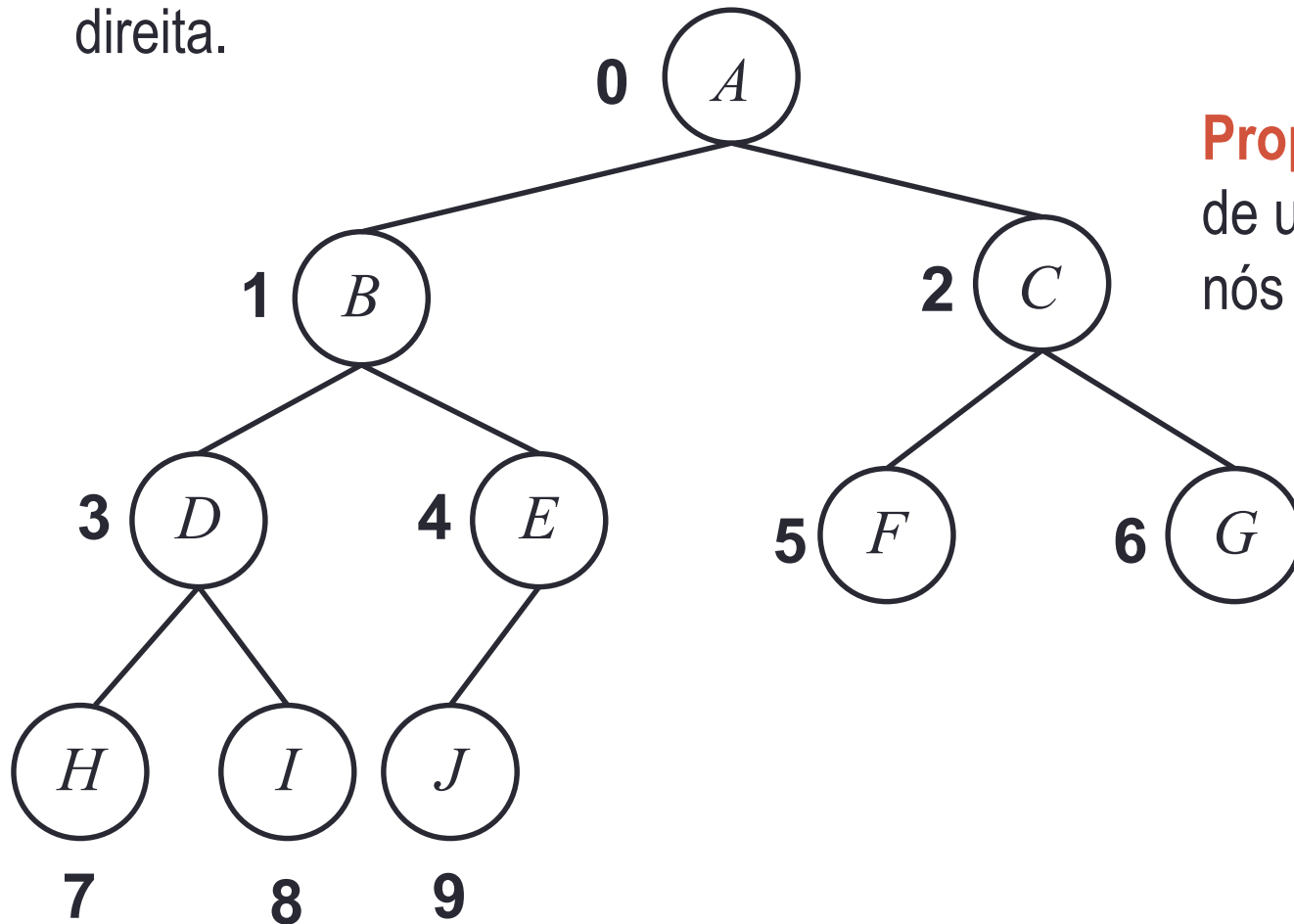
- A prioridade de X é **menor ou igual** à prioridade de qualquer nó que ocorra numa subárvore de X



Propriedade: A raiz (de uma árvore não vazia) tem prioridade **mínima**

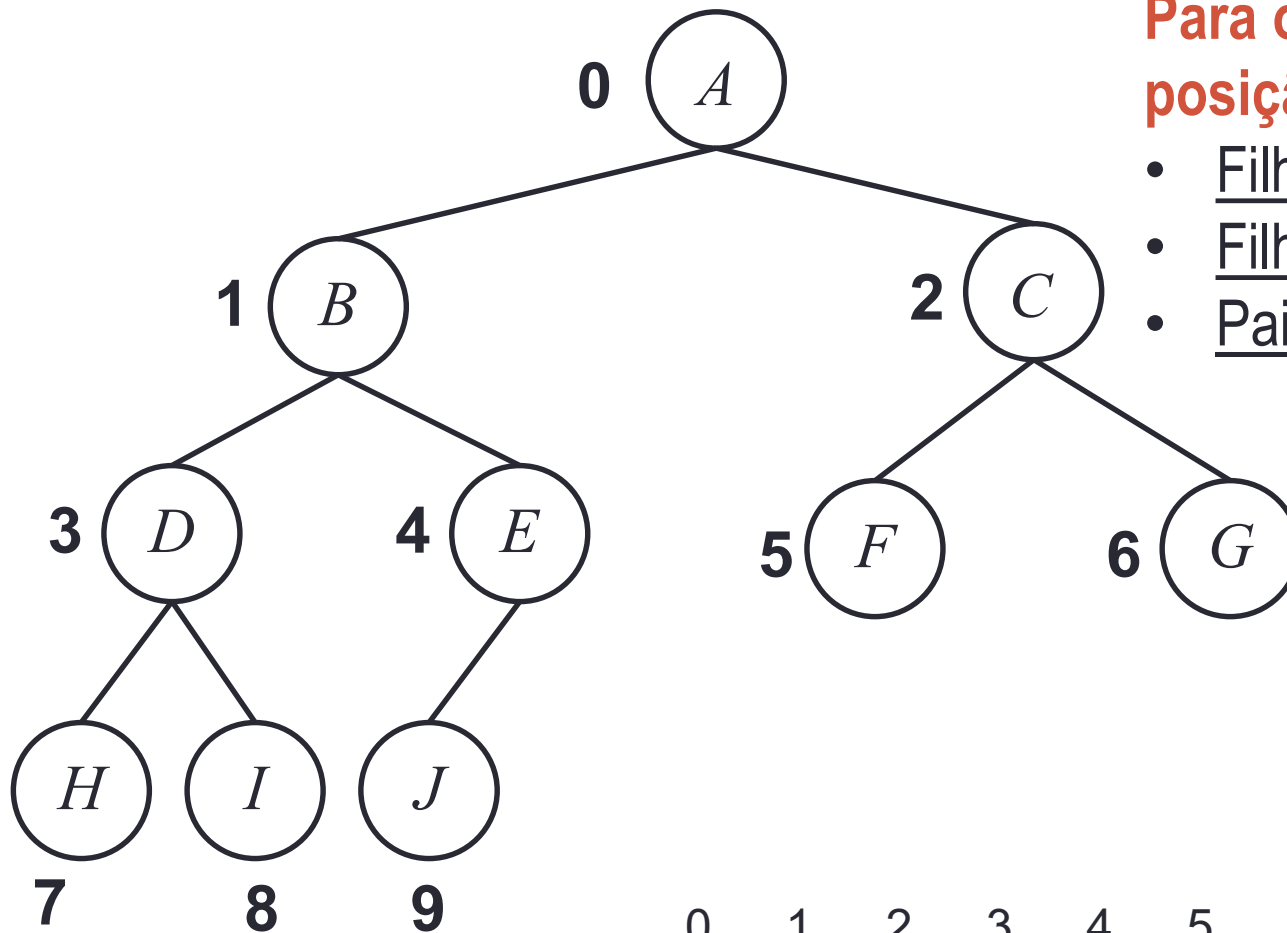
Árvore Binária Completa (Esquerda)

Todos os níveis estão completamente preenchidos, exceto, possivelmente, o último, que está preenchido da esquerda para a direita.



Propriedade: A altura de uma árvore com n nós é $\lceil \log(n+1) \rceil$

Árvore Binária Completa em Vetor



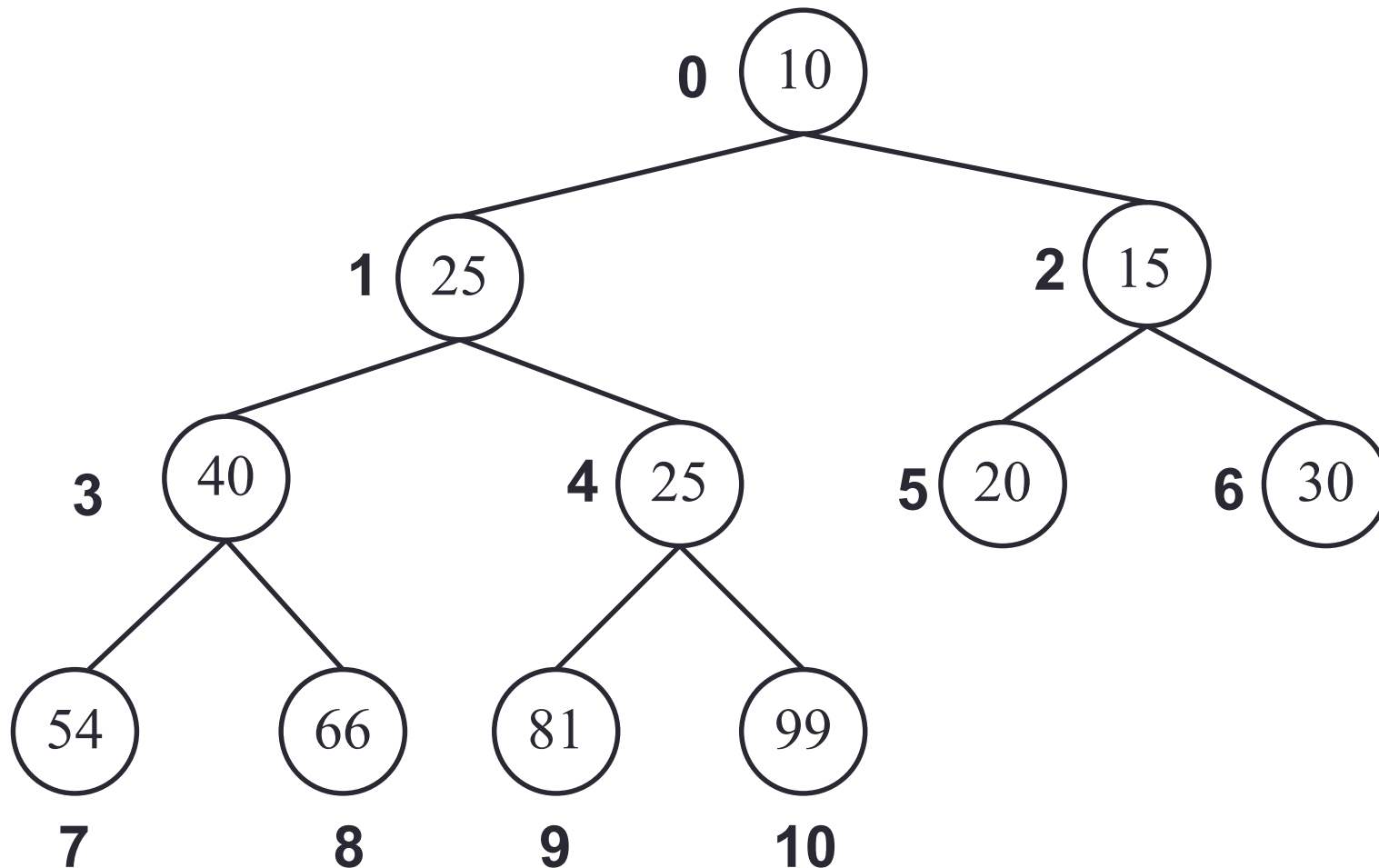
Para o nó guardado na posição i :

- Filho esquerdo em $2i + 1$
- Filho direito em $2i + 2$
- Pai em $(i - 1) \text{ div } 2$

0	1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J	

Heap (J. W. J. Williams, 1964)

- Árvore Binária Completa com prioridade.



Classe MinHeap (1)

```
package dataStructures;

public class MinHeap<K extends Comparable<K>, V>
    implements MinPriorityQueue<K,V> {

    // Default capacity of the priority queue.
    public static final int DEFAULT_CAPACITY = 100;

    // The growth factor of the extendable array.
    public static final int GROWTH_FACTOR = 2;

    // Memory of the priority queue: an extendable array.
    protected Entry<K,V>[] array;

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

Classe MinHeap (2)

```
// Creates a heap with the specified capacity.  
public MinHeap( int capacity ){  
  
    // Compiler gives a warning.  
    array = (Entry<K,V>[]) new Entry[capacity];  
    currentSize = 0;  
}  
  
// Creates a heap with the default capacity.  
public MinHeap( ){  
    this(DEFAULT_CAPACITY);  
}
```

Classe MinHeap (3)

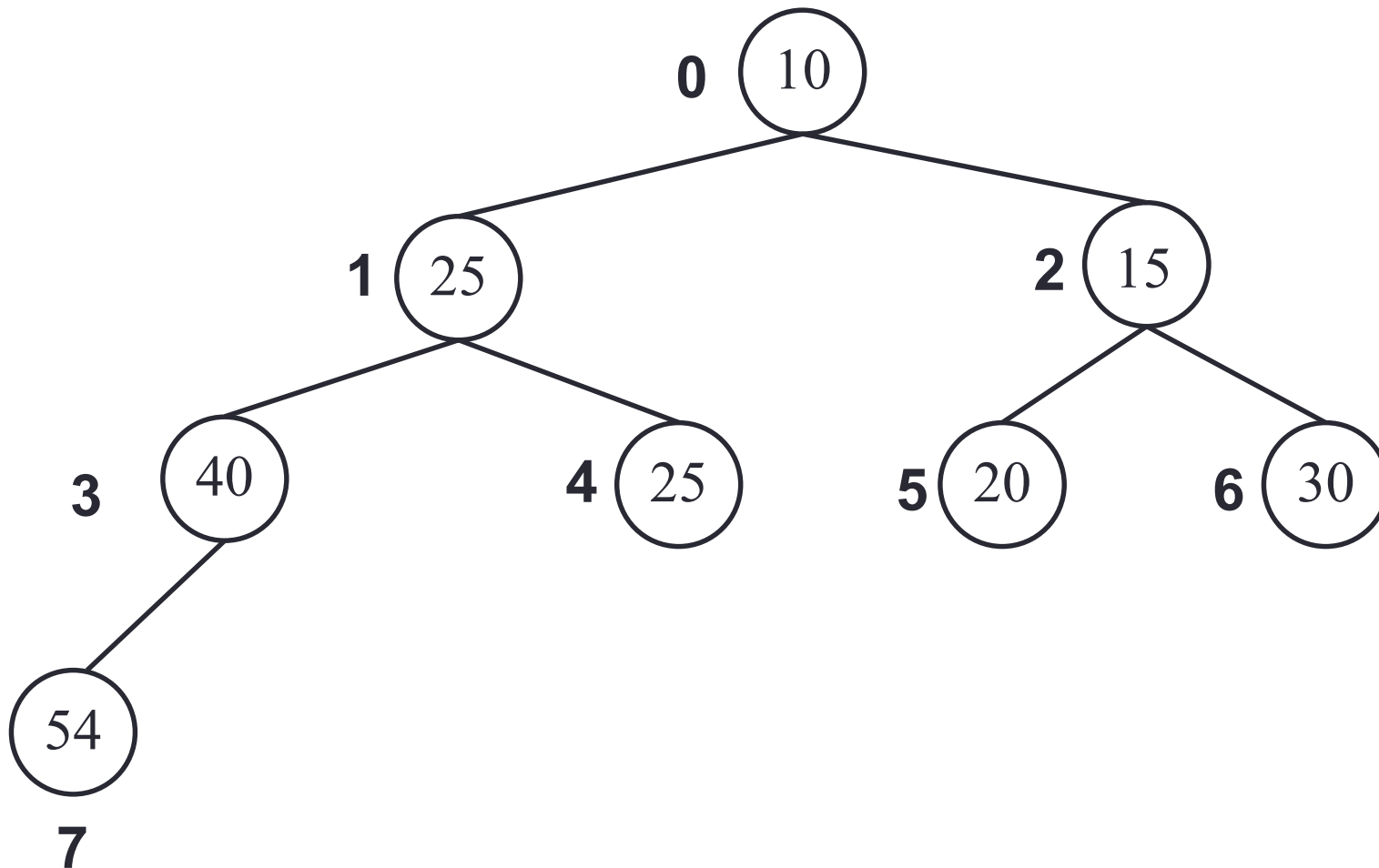
```
// Returns true iff the priority queue contains no entries.  
public boolean isEmpty( ){  
    return currentSize == 0;  
}  
  
// Returns true iff the array cannot contain more entries.  
protected boolean isFull( ){  
    return currentSize == array.length;  
}  
  
// Returns the number of entries in the priority queue.  
public int size( ){  
    return currentSize;  
}
```

Classe MinHeap (4)

```
// Returns an entry with the smallest key in the priority queue.  
public Entry<K,V> minEntry( )  
    throws EmptyPriorityQueueException {  
  
    if ( this.isEmpty() )  
        throw new  
            EmptyPriorityQueueException("Priority queue is empty.");  
    return array[0];  
}  
.....  
} // End of MinHeap.
```

Inserir 12

- Criar buraco

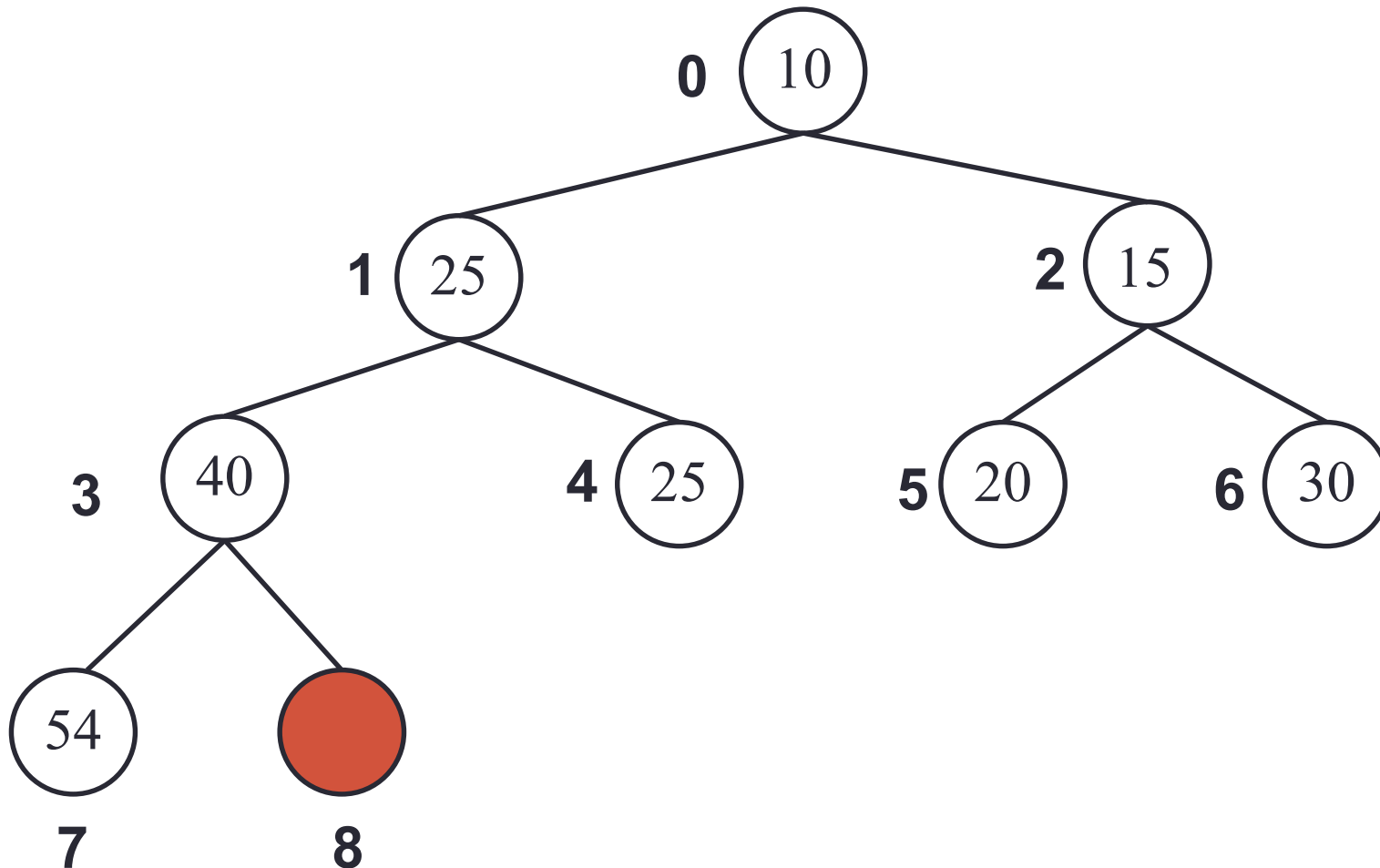


Inserir 12

- Criar buraco

$12 < 40$?

Sim: 40 desce, buraco sobe

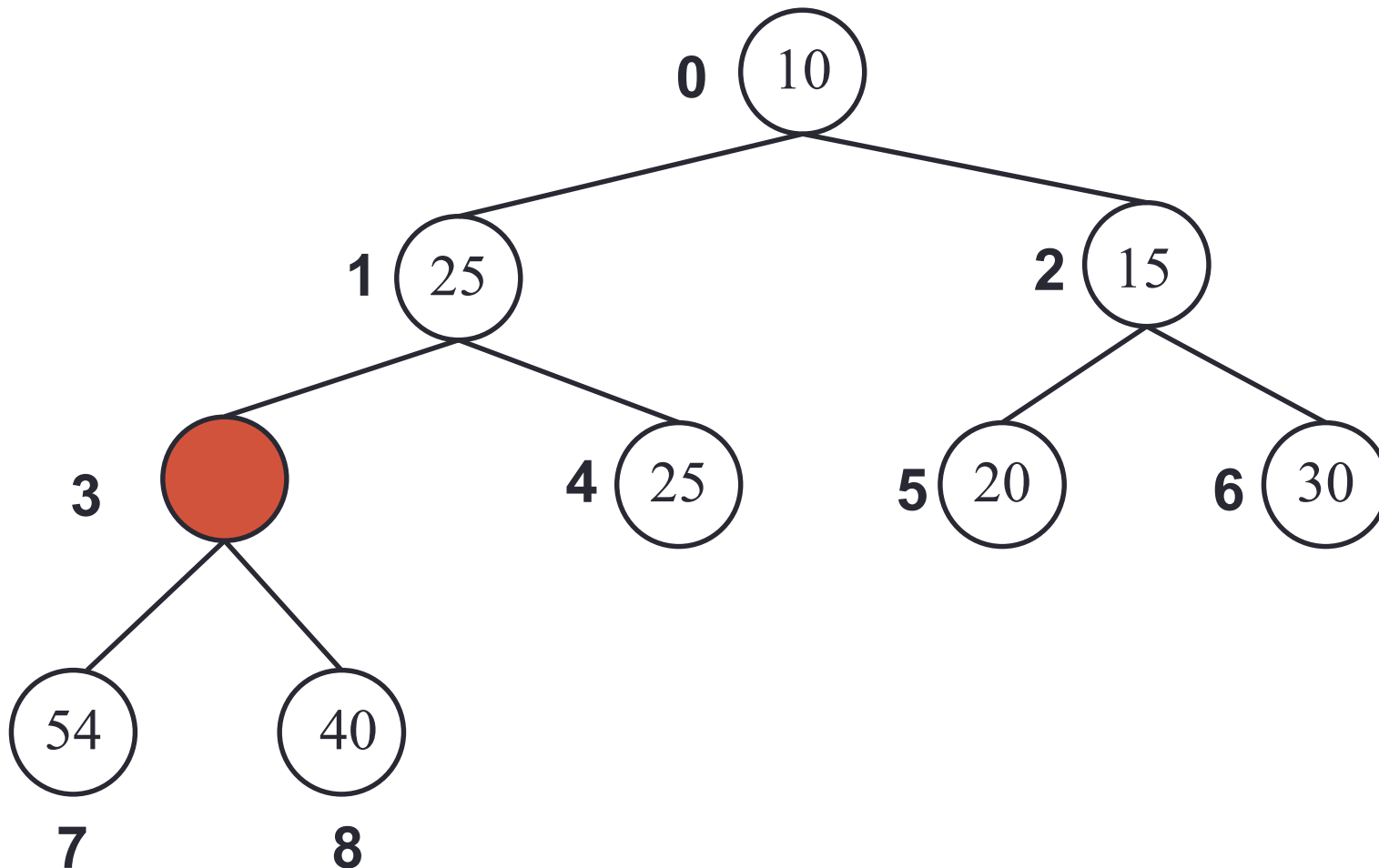


Inserir 12

- Criar buraco

$12 < 40$?

Sim: 40 desce, buraco sobe

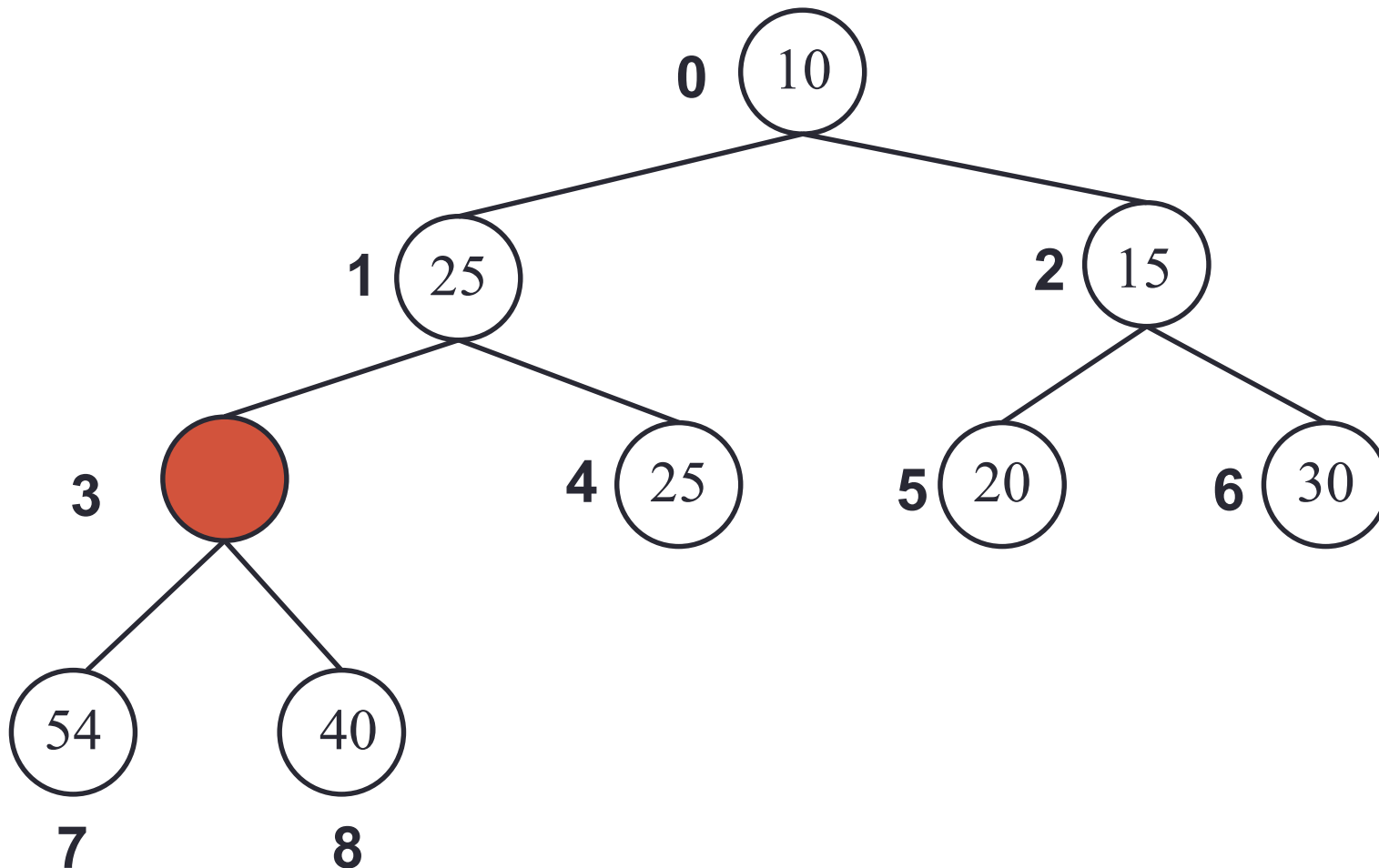


Inserir 12

- Criar buraco

$12 < 25$?

Sim: 25 desce, buraco sobe

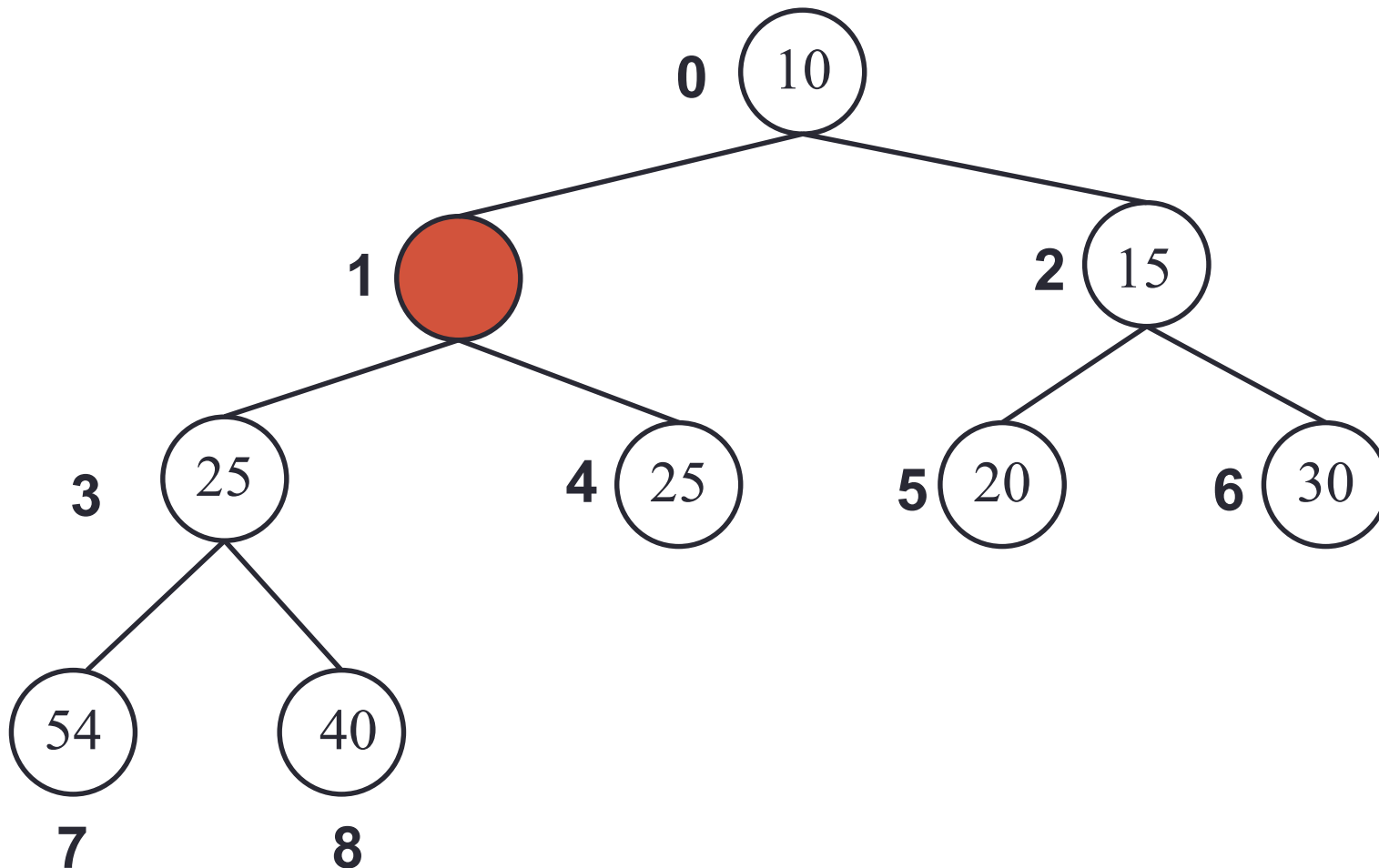


Inserir 12

- Criar buraco

$12 < 25$?

Sim: 25 desce, buraco sobe

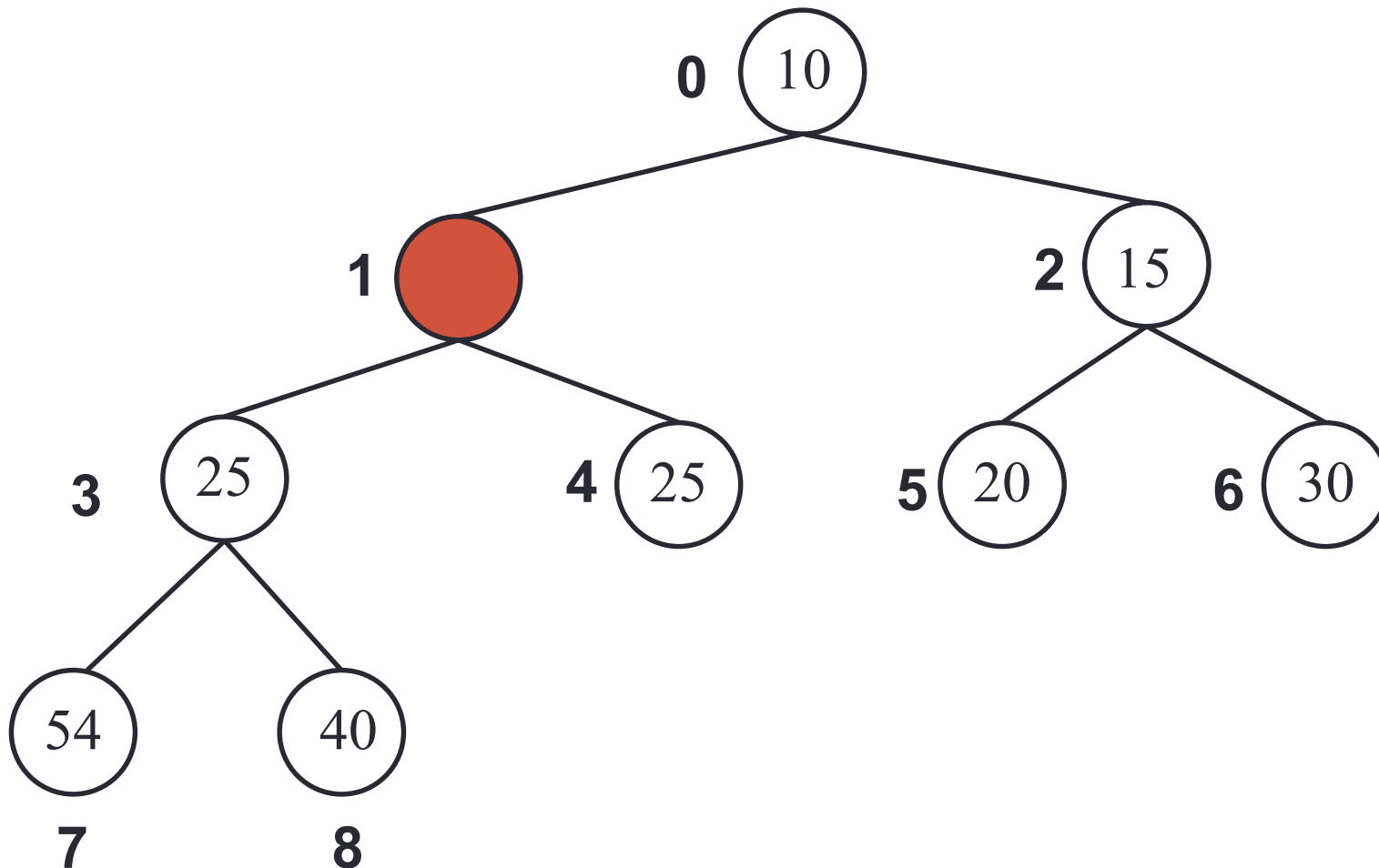


Inserir 12

- Criar buraco

$12 < 10$?

Não: colocar 12 no buraco

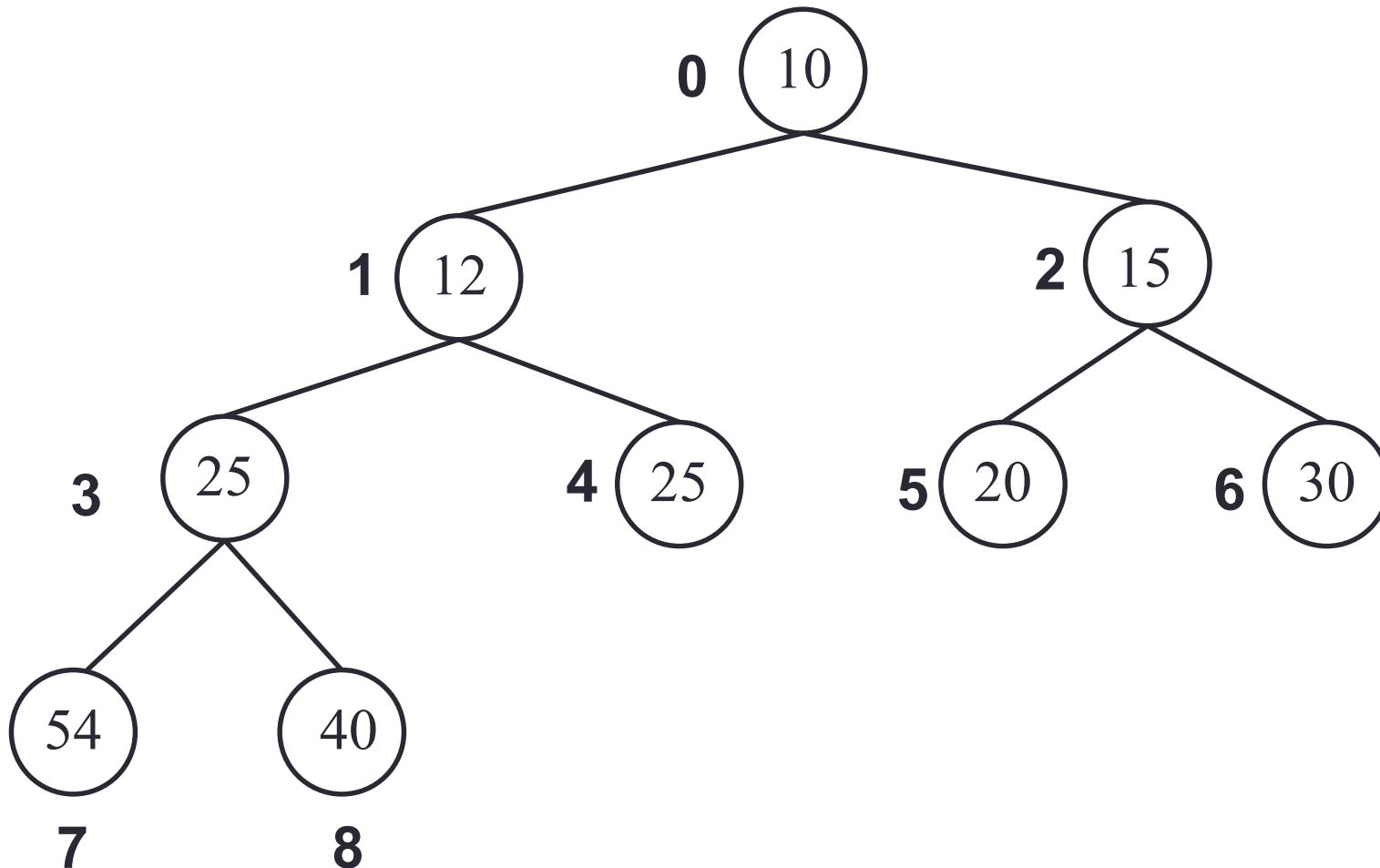


Inserir 12

- Criar buraco

$12 < 10$?

Não: colocar 12 no buraco



Classe MinHeap - insert

```
// Inserts the entry (key, value) in the priority queue.
public void insert( K key, V value ){
    if ( this.isFull() )
        this.buildArray(GROWTH_FACTOR * array.length, array);

    // Percolate up.
    int hole = currentSize;
    int parent = (hole - 1) / 2;
    while ( hole > 0 &&
            key.compareTo( array[parent].getKey() ) < 0 ){
        array[hole] = array[parent];
        hole = parent;
        parent = (hole - 1) / 2;
    }
    array[hole] = new EntryClass<K,V>(key, value);
    currentSize++;
}
```

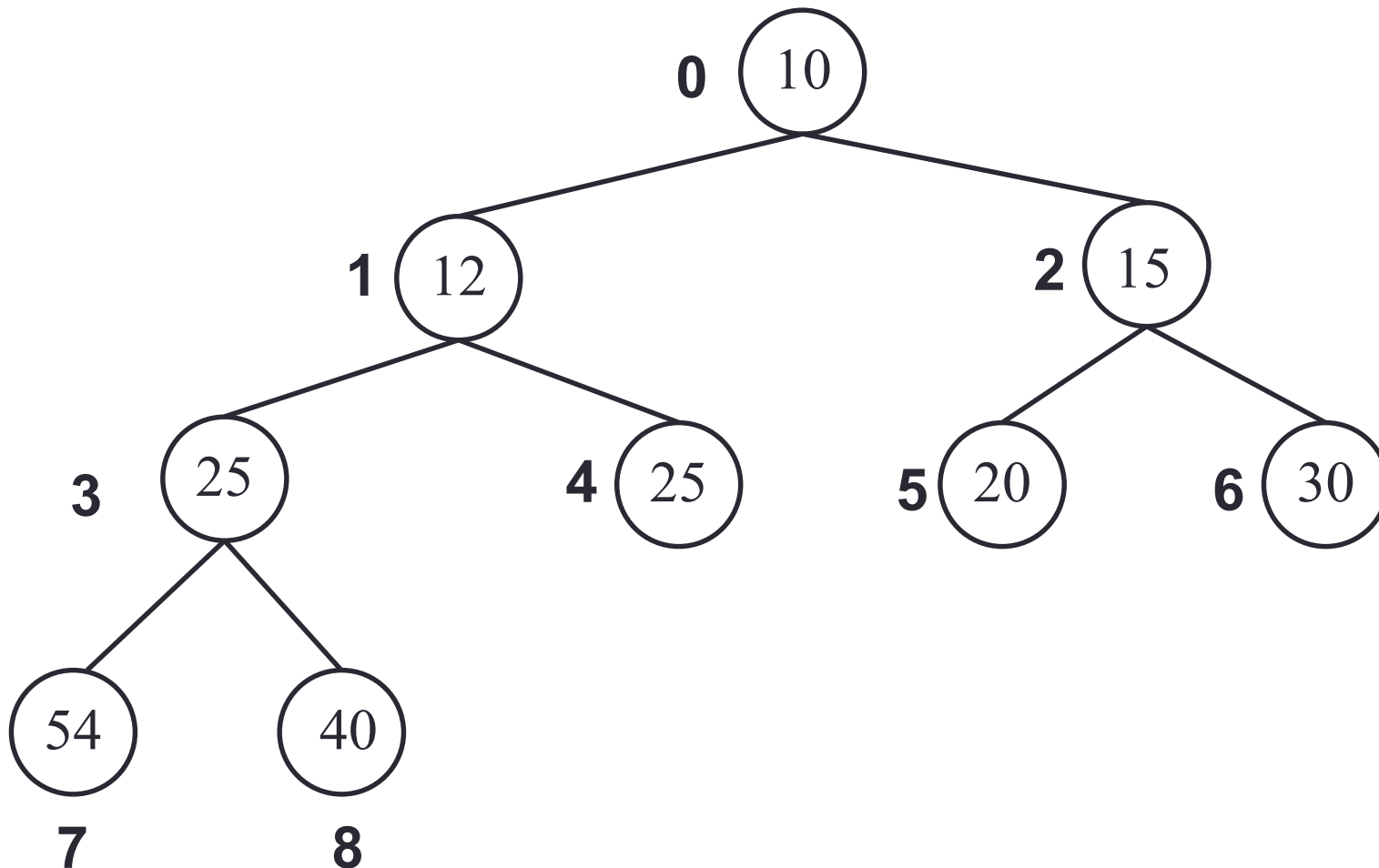
Classe MinHeap - buildArray

```
// Builds the extendable array with the specified capacity
// and with the contents of the specified array.
// Requires: capacity >= contents.length.
protected void buildArray( int capacity, Entry<K,V>[] contents ){

    // Compiler gives a warning.
    Entry<K,V>[] newArray = (Entry<K,V>[]) new Entry[capacity];
    System.arraycopy(contents, 0, newArray, 0, contents.length);
    array = newArray;
}
```

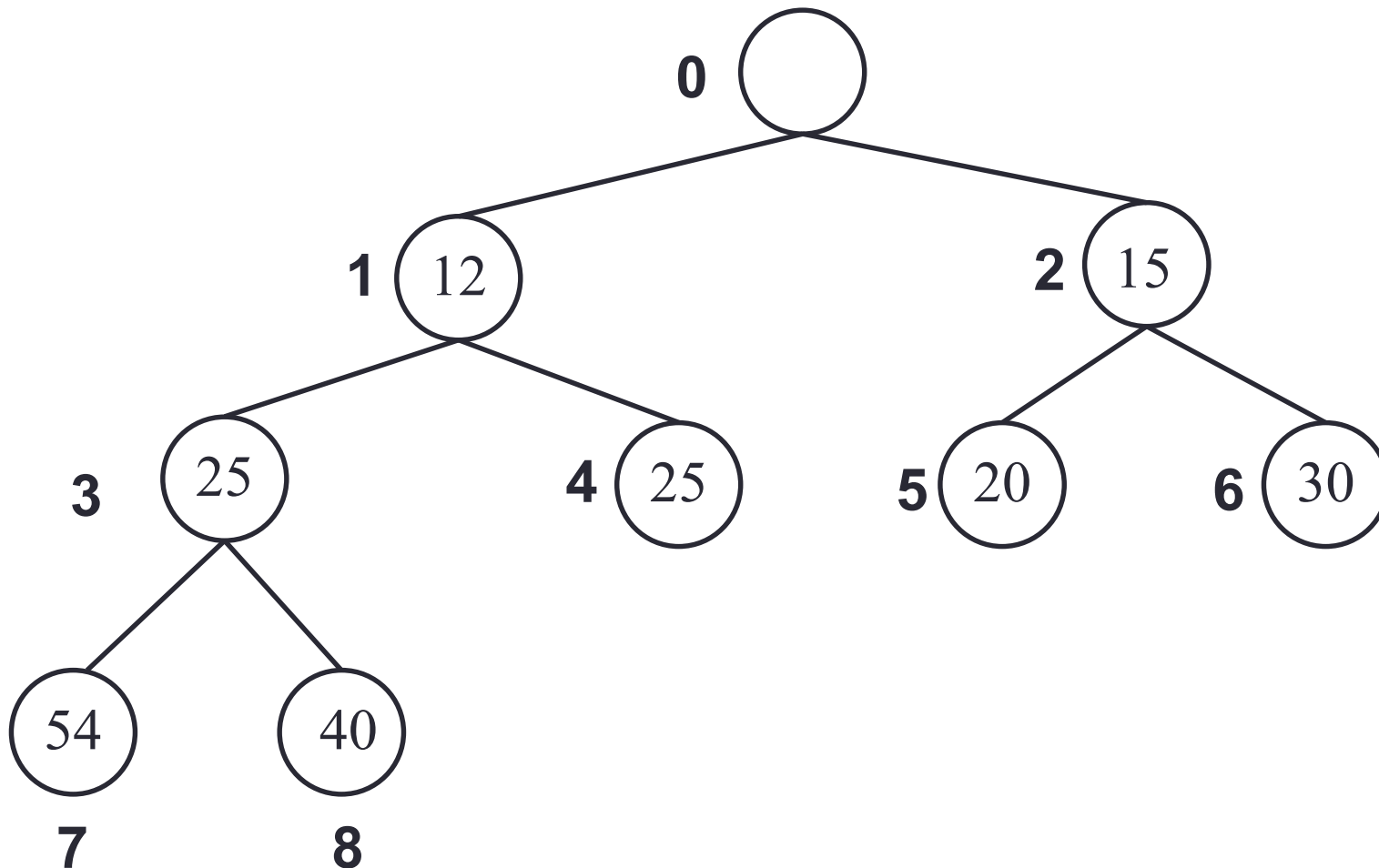
Remover o mínimo

Guardar 10 para devolver no fim



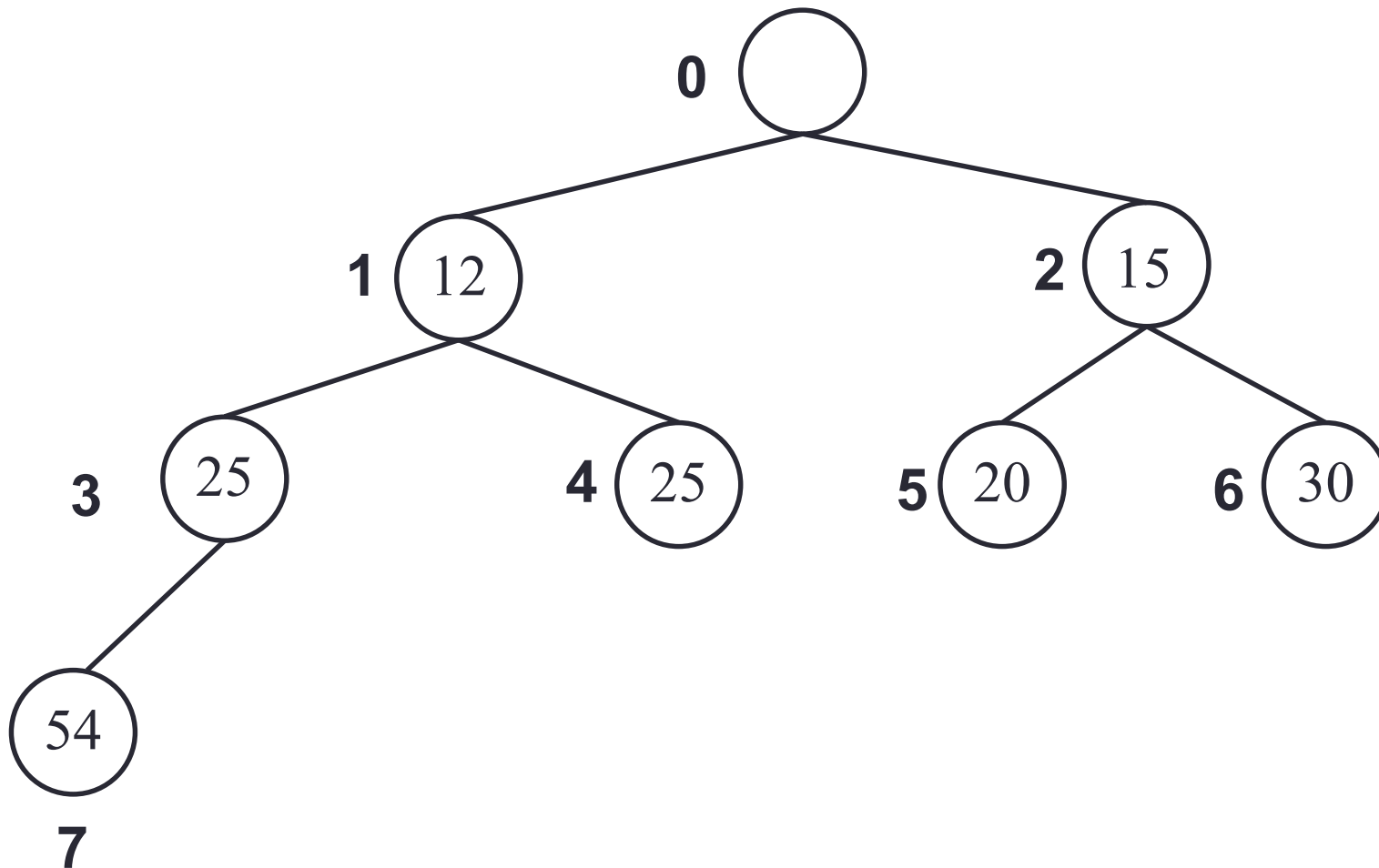
Remover o mínimo

Guardar 10 para devolver no fim
Remover ultima posição



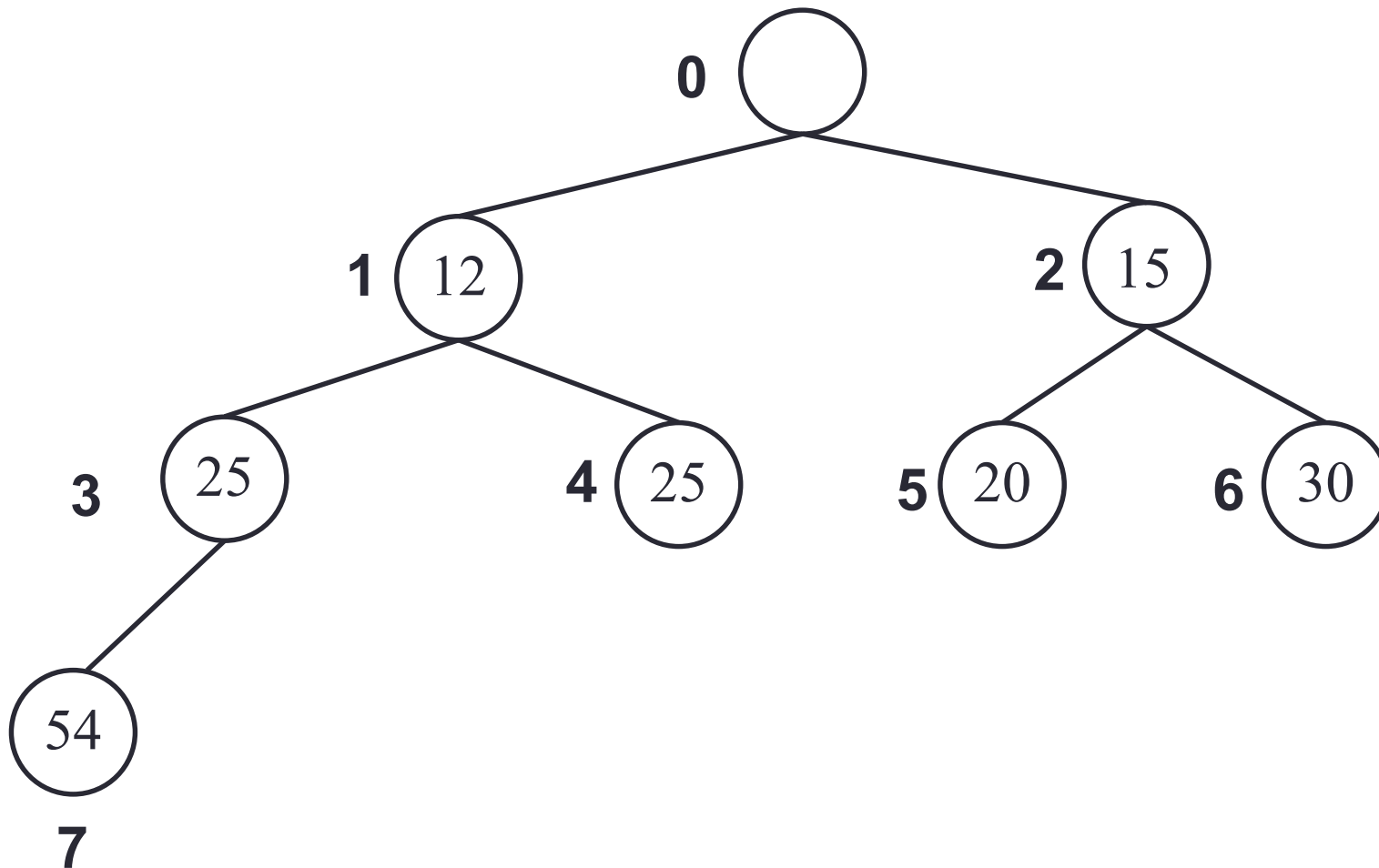
Remover o mínimo

Guardar 10 para devolver no fim
Remover ultima posição



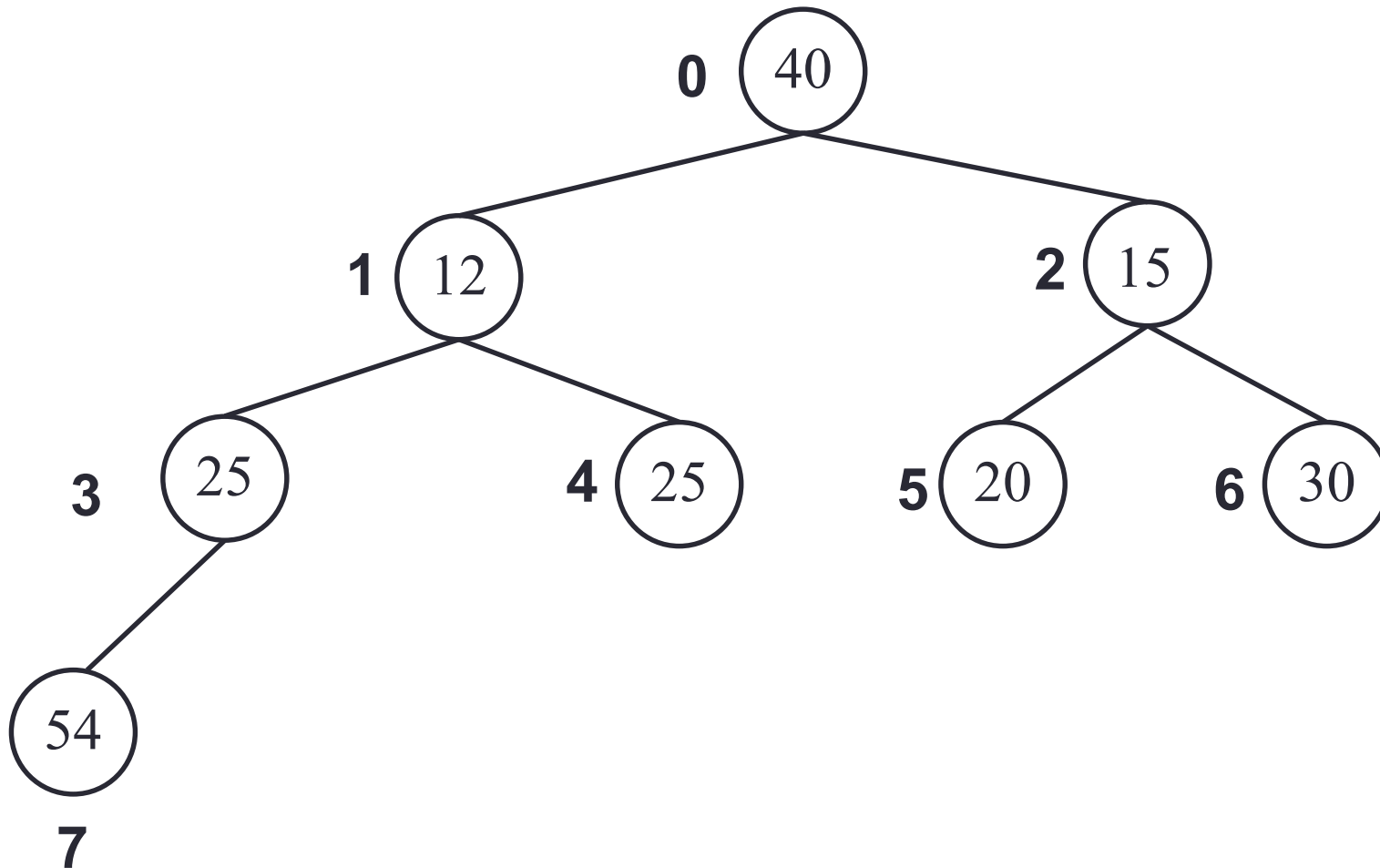
Remover o mínimo

Guardar 10 para devolver no fim
Remover ultima posição
Deslocar 40 para raiz



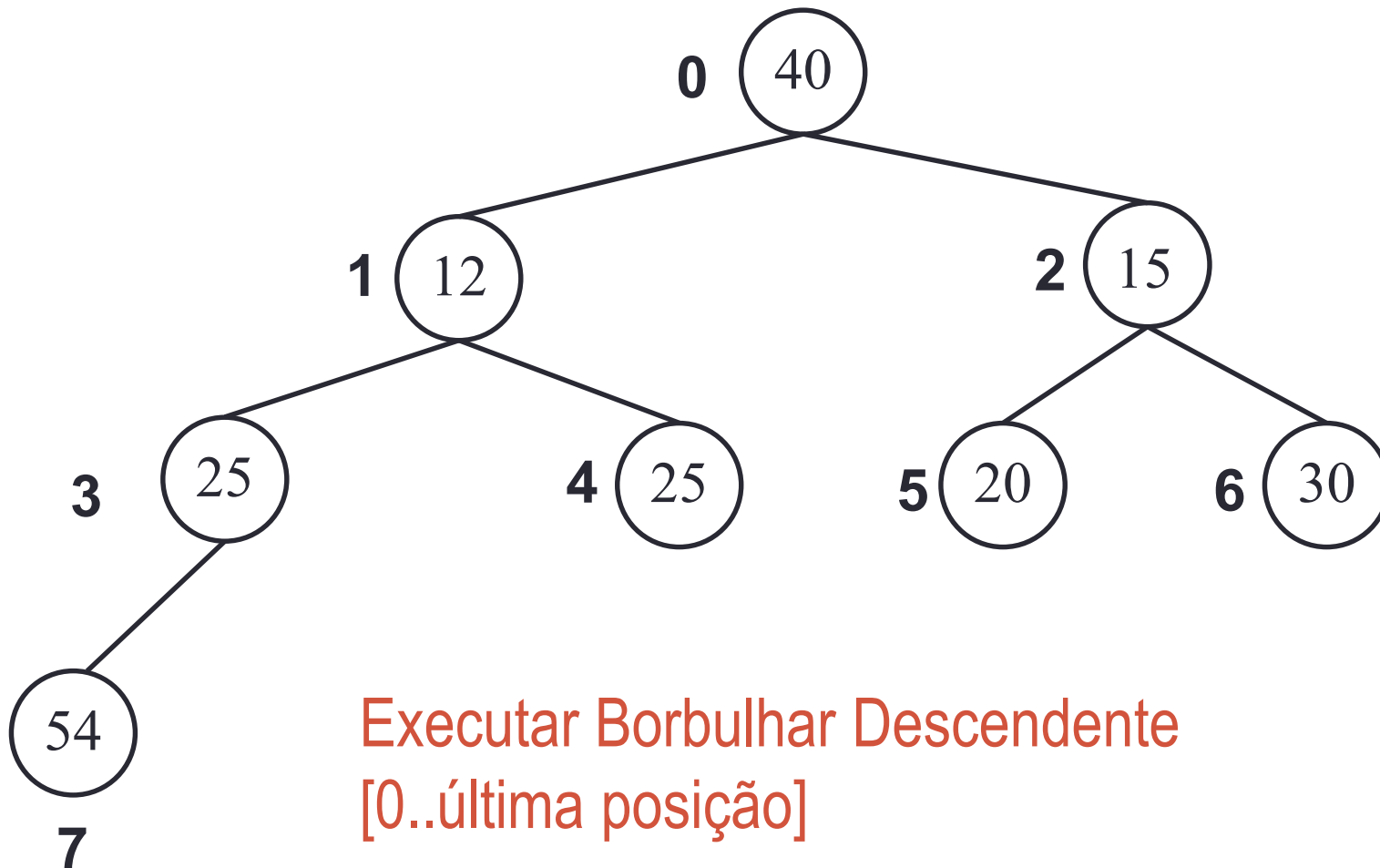
Remover o mínimo

Guardar 10 para devolver no fim
Remover ultima posição
Deslocar 40 para raiz



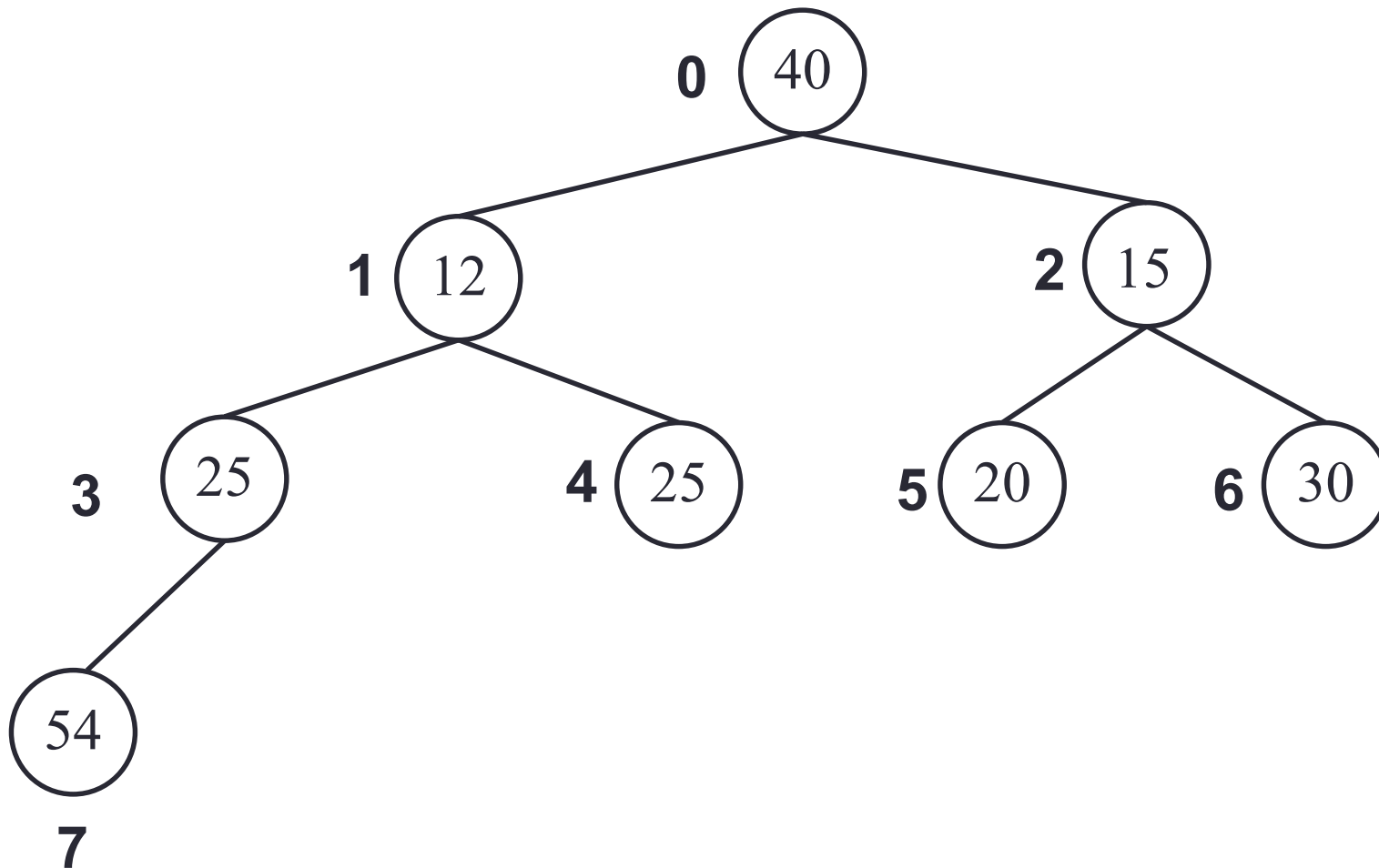
Remover o mínimo

Guardar 10 para devolver no fim
Remover ultima posição
Deslocar 40 para raiz



Borbulhar descendente

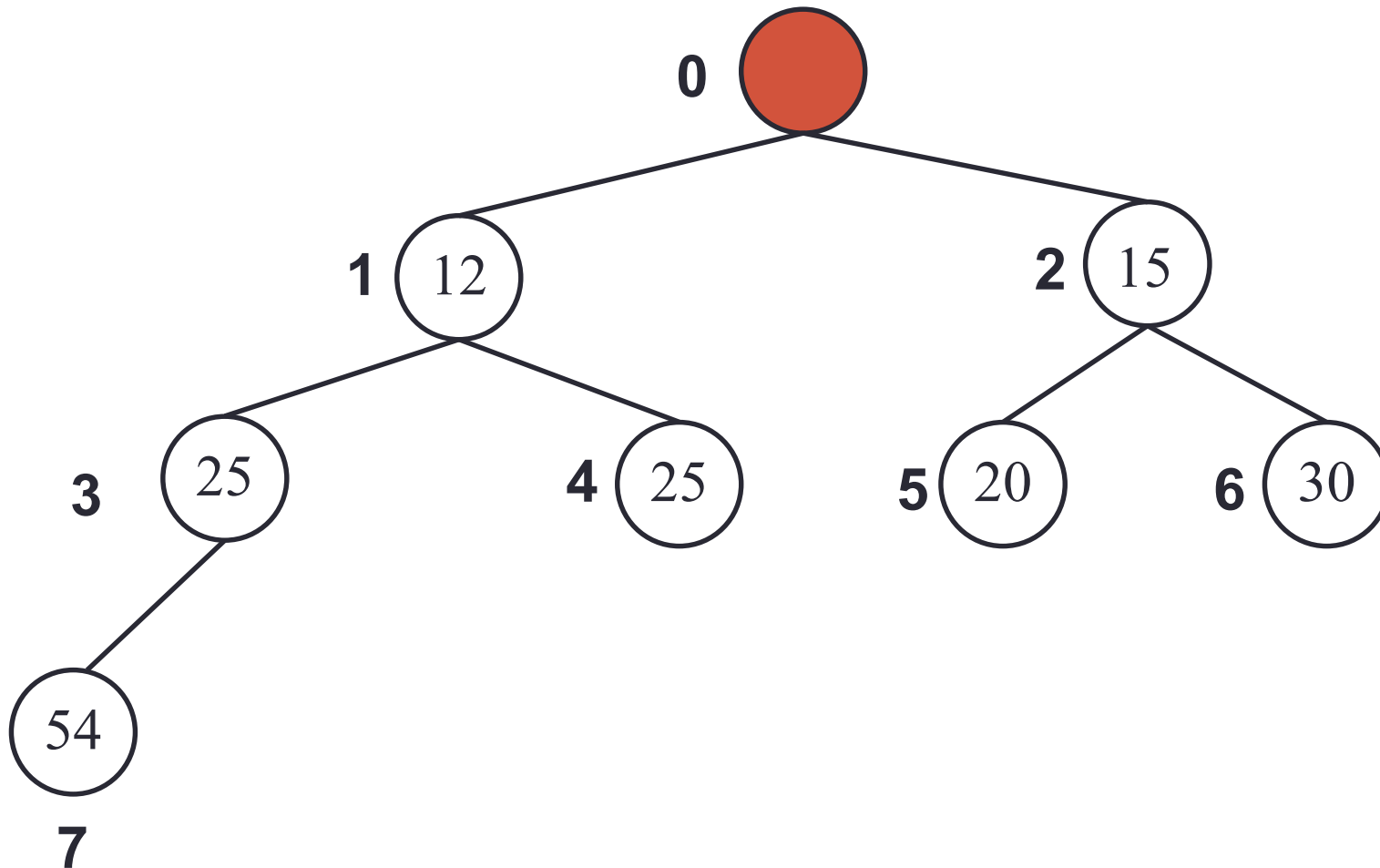
Guardar 40
Criar Buraco



Borbulhar descendente

$40 > \min(12, 15)$?

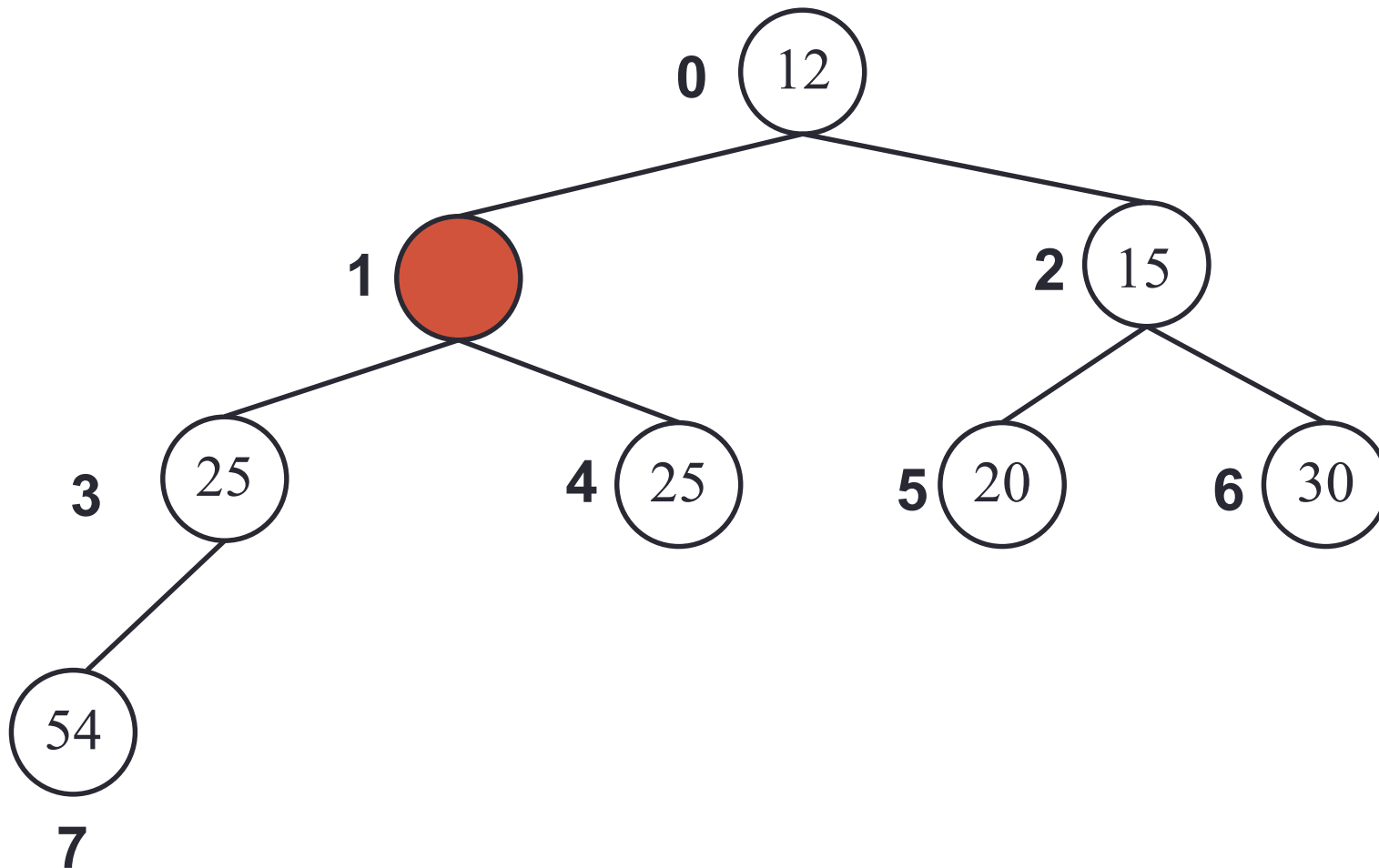
Sim: 12 sobe, buraco desce



Borbulhar descendente

$40 > \min(12, 15)$?

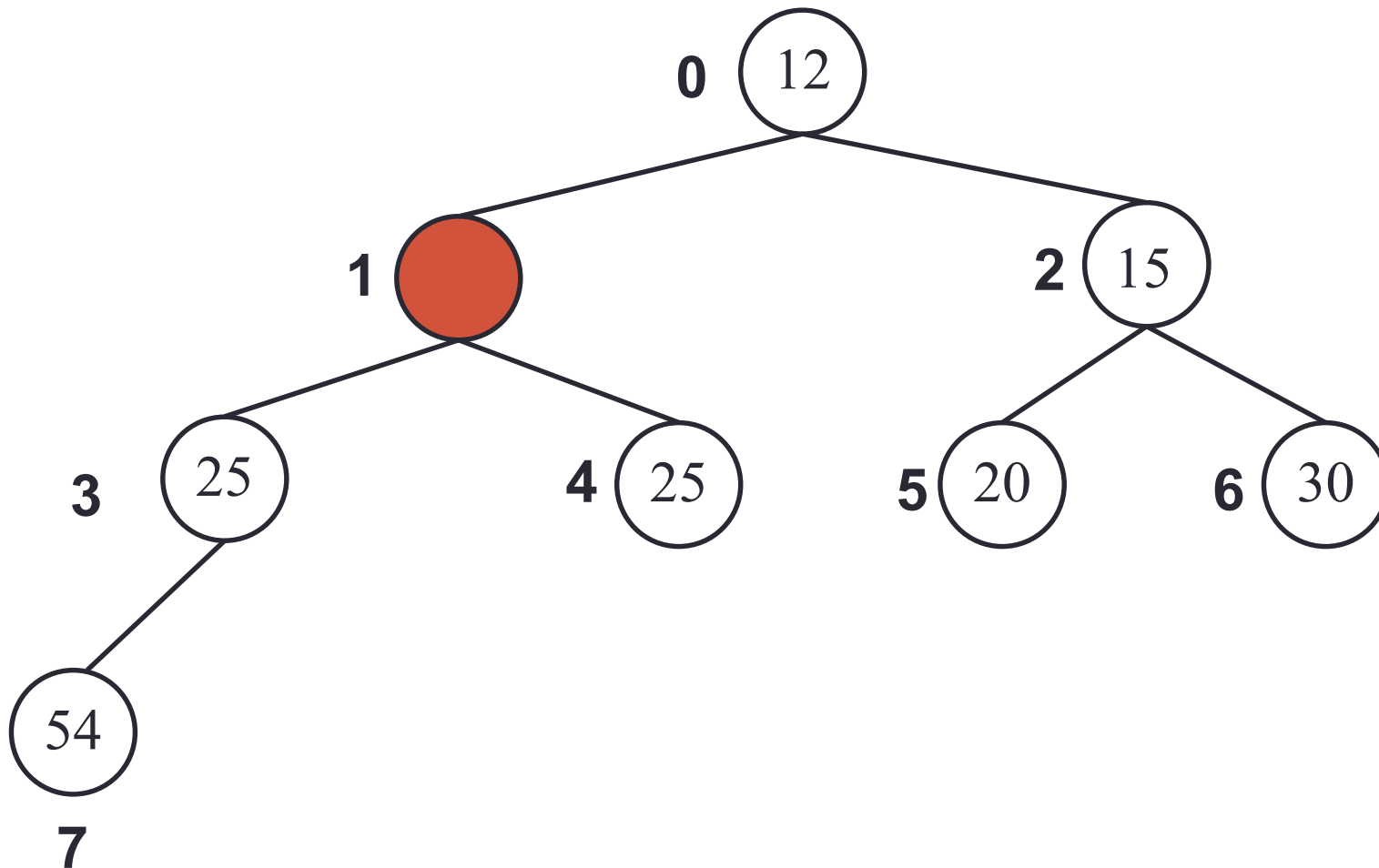
Sim: 12 sobe, buraco desce



Borbulhar descendente

$40 > \min(25, 25)$?

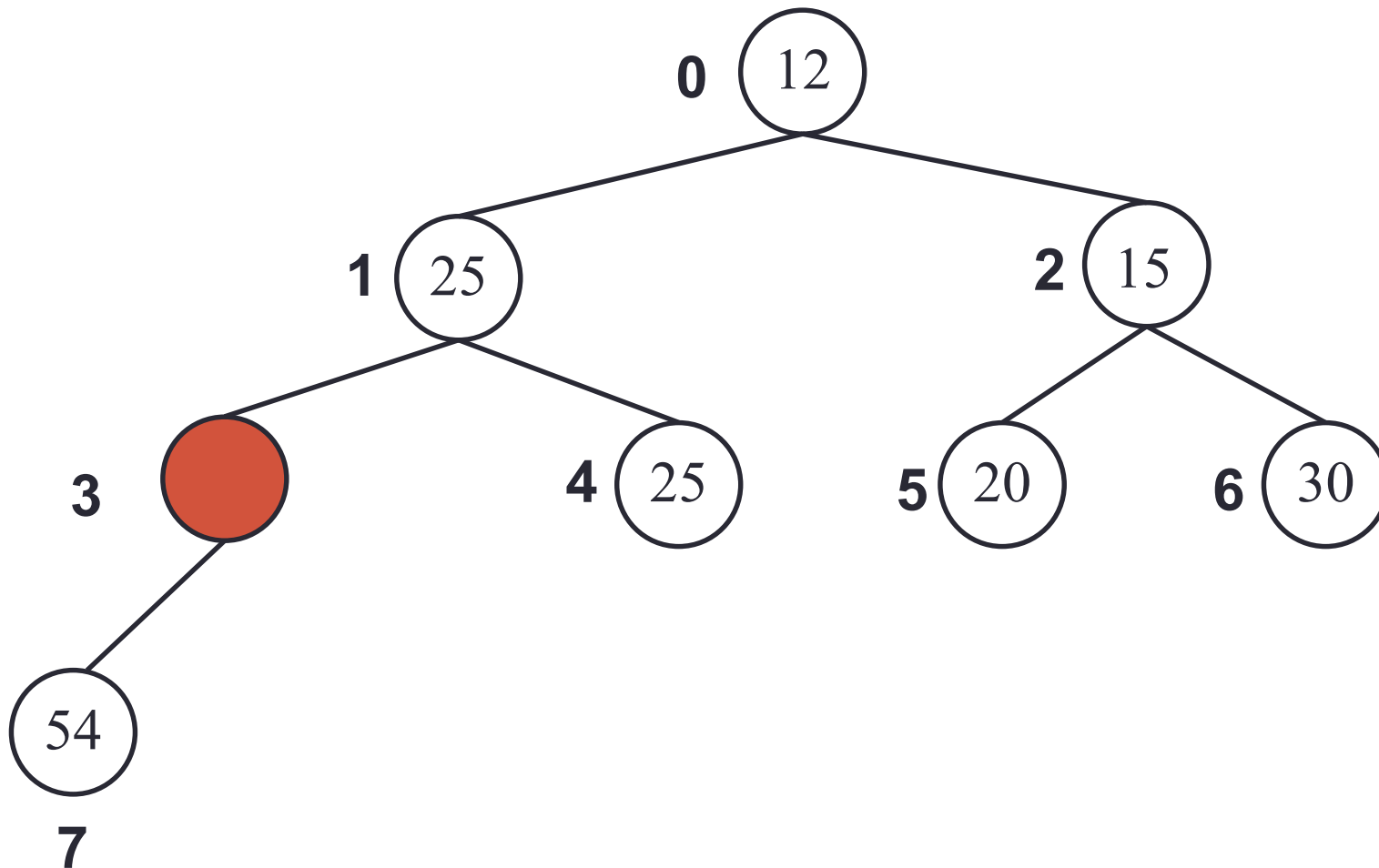
Sim: 25 sobe, buraco desce



Borbulhar descendente

$40 > \min(25, 25)$?

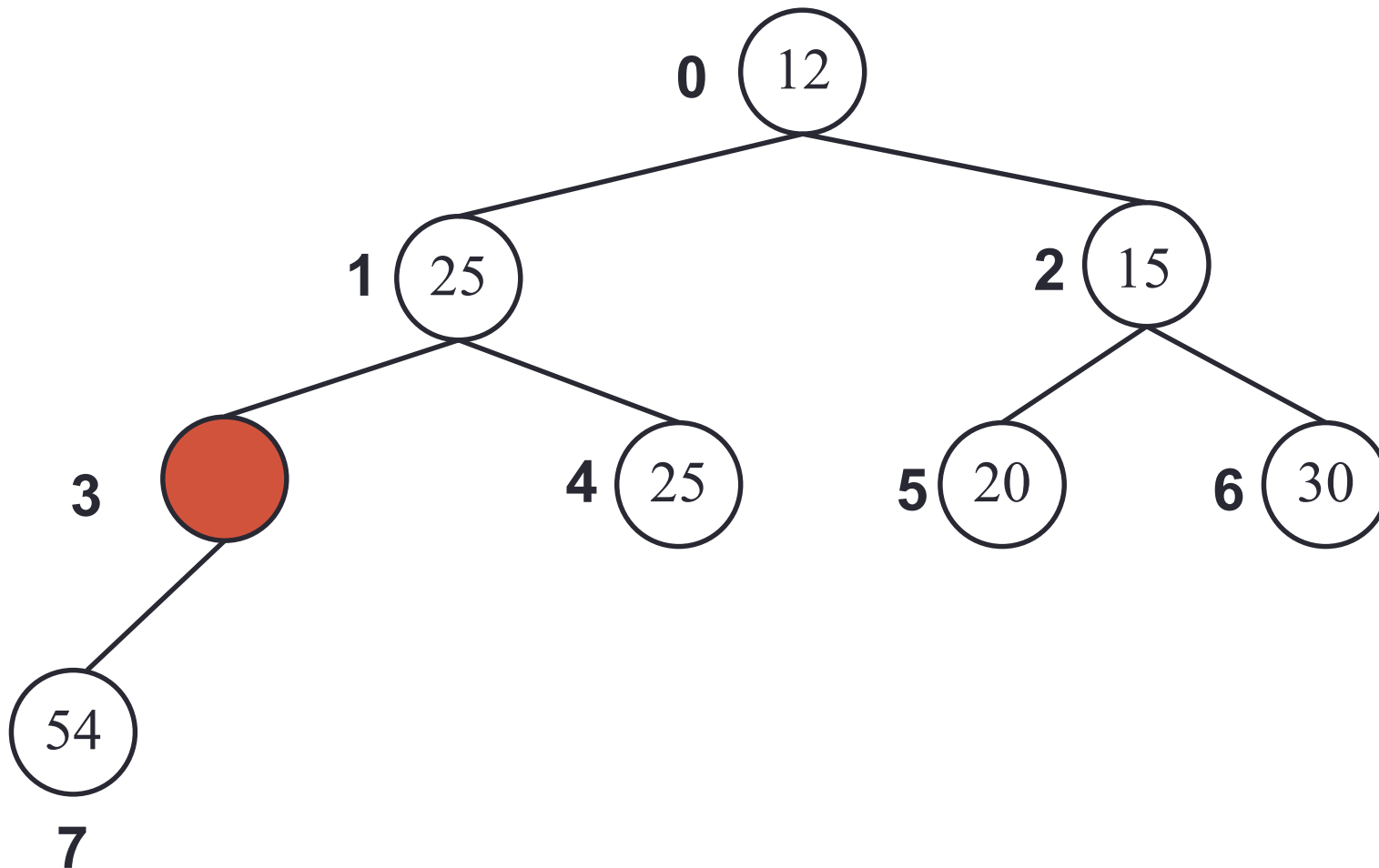
Sim: 25 sobe, buraco desce



Borbulhar descendente

40 > 54 ?

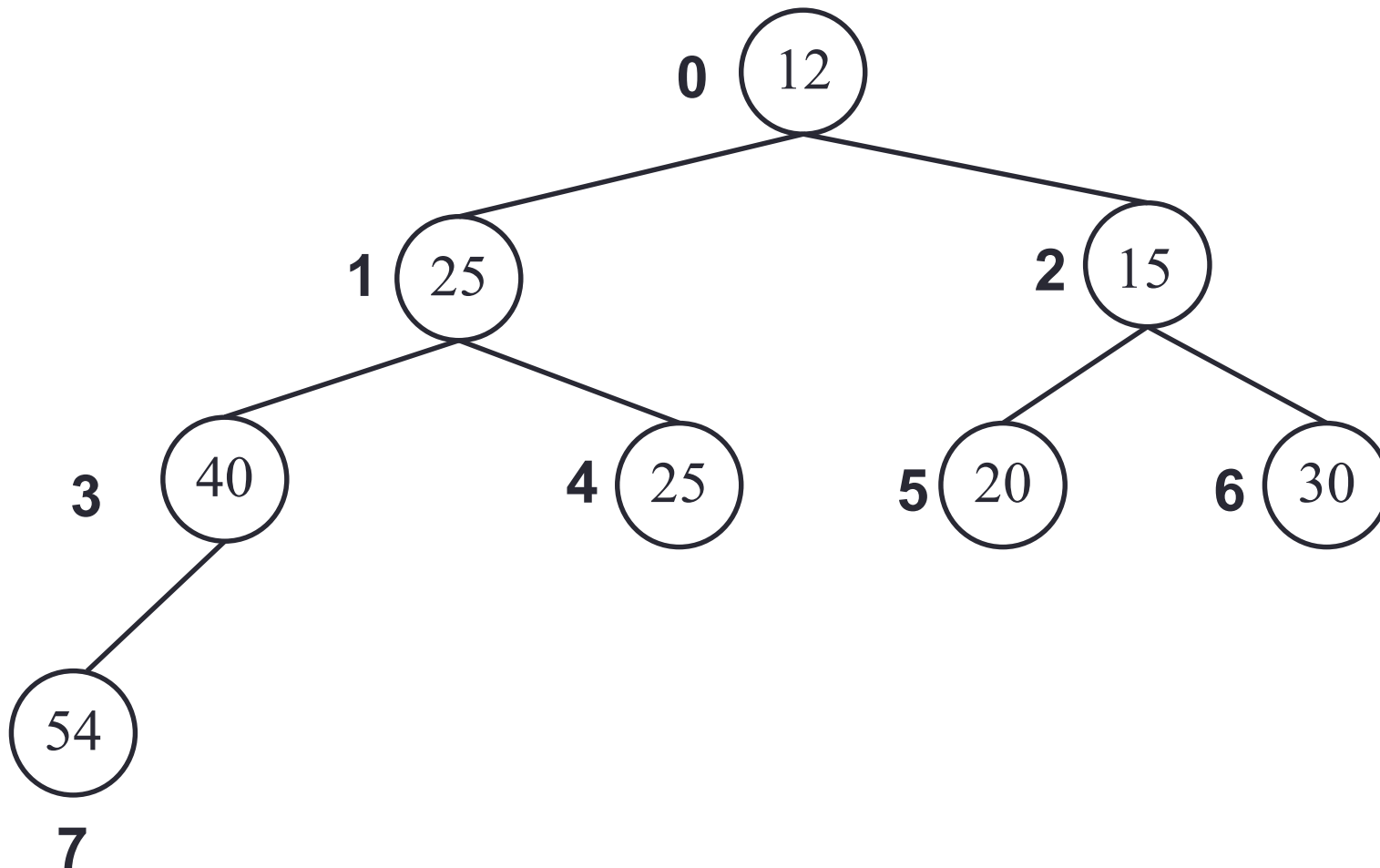
Não: Colocar 40 no buraco



Borbulhar descendente

40 > 54 ?

Não: Colocar 40 no buraco



Classe MinHeap - removeMin

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

    Entry<K,V> minEntry = array[0];
    currentSize--;
    array[0] = array[currentSize];
    array[currentSize] = null; // For garbage collection.
    if ( currentSize > 1 )
        this.percolateDown(0);
    return minEntry;
}
```

Classe MinHeap – percolateDown (1)

```
// Requires: firstPos < currentSize.
```

```
protected void percolateDown( int firstPos ){
```

```
    Entry<K,V> rootEntry = array[firstPos];
```

```
    K rootKey = rootEntry.getKey();
```

```
    int hole = firstPos;
```

```
    int child = 2 * hole + 1; // Left child.
```

```
    while ( child < currentSize ) {
```

```
        // Find the smallest child.
```

```
        .....
```

```
        // Compare the smallest child with rootKey.
```

```
        .....
```

```
    }
```

```
    array[hole] = rootEntry;
```

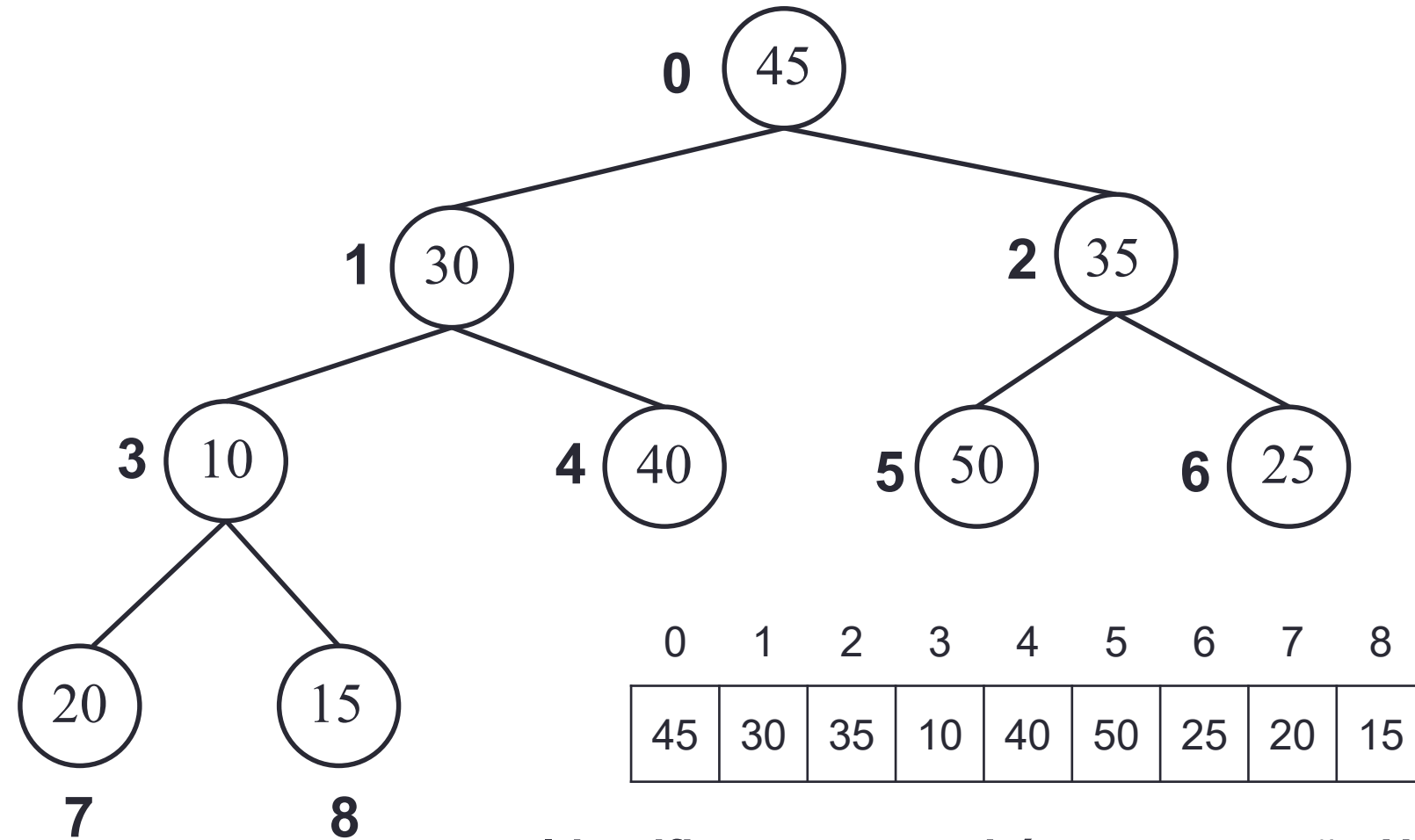
```
}
```

Ver próximo slide

Classe MinHeap – percolateDown (2)

```
while ( child < currentSize ) {  
    // Find the smallest child.  
    if ( child < currentSize - 1 &&  
        array[child+1].getKey().compareTo(  
            array[child].getKey() ) < 0 )  
        child++;  
    // Compare the smallest child with rootKey.  
    if ( array[child].getKey().compareTo( rootKey ) < 0 ) {  
        array[hole] = array[child];  
        hole = child;  
        child = 2 * hole + 1; // Left child.  
    }  
    else  
        break;  
}  
array[hole] = rootEntry;
```

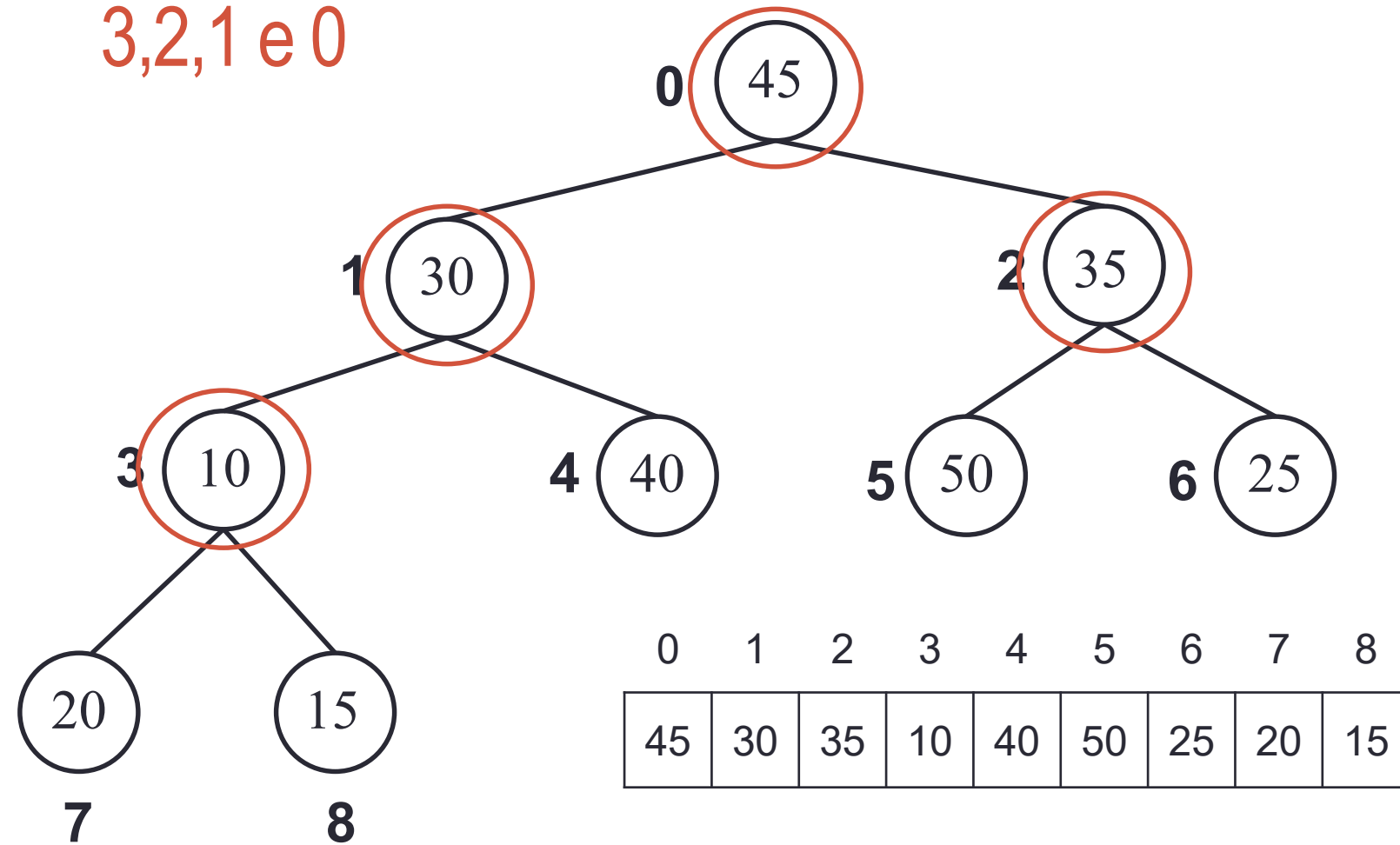
Criação de um Heap a partir de um Vetor



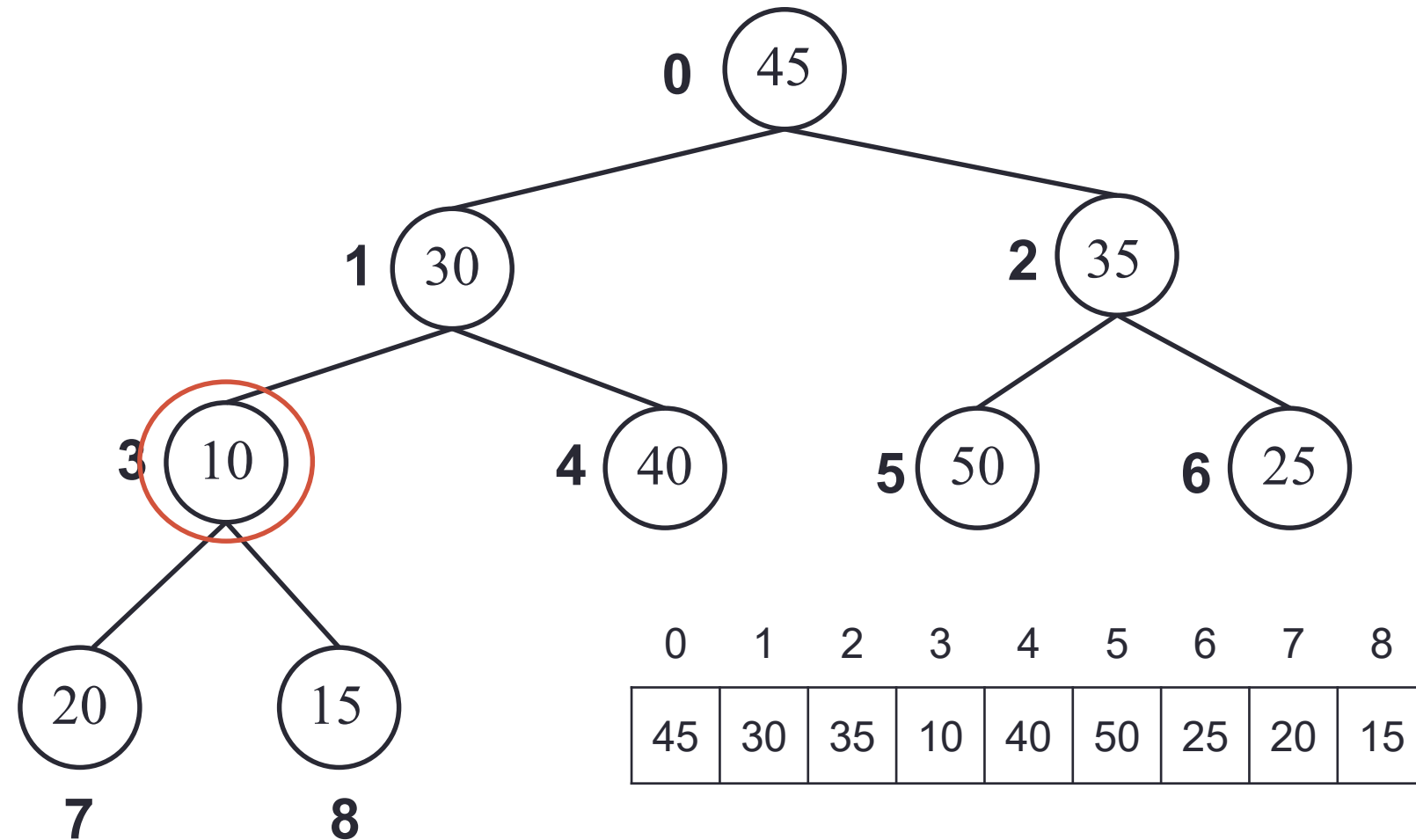
Identificam-se as subárvores que são Heaps

8, 7, 6, 5 e 4 são Heaps

Executa-se Borbulhar descendente para índices 3,2,1 e 0



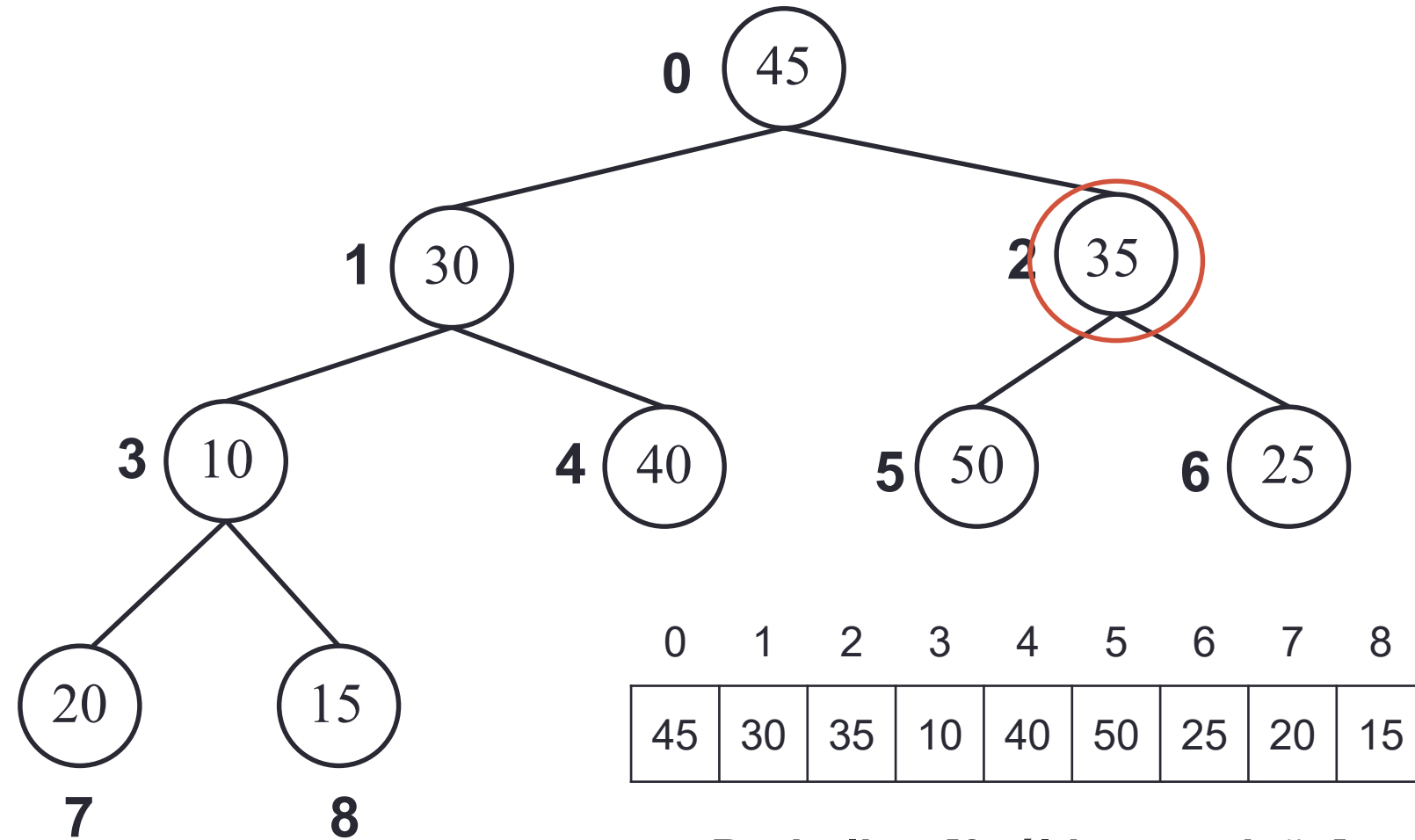
Borbulhar descendente para índice 3



Borbulhar [3..última posição]

8, 7, 6, 5, 4 e 3 são Heaps

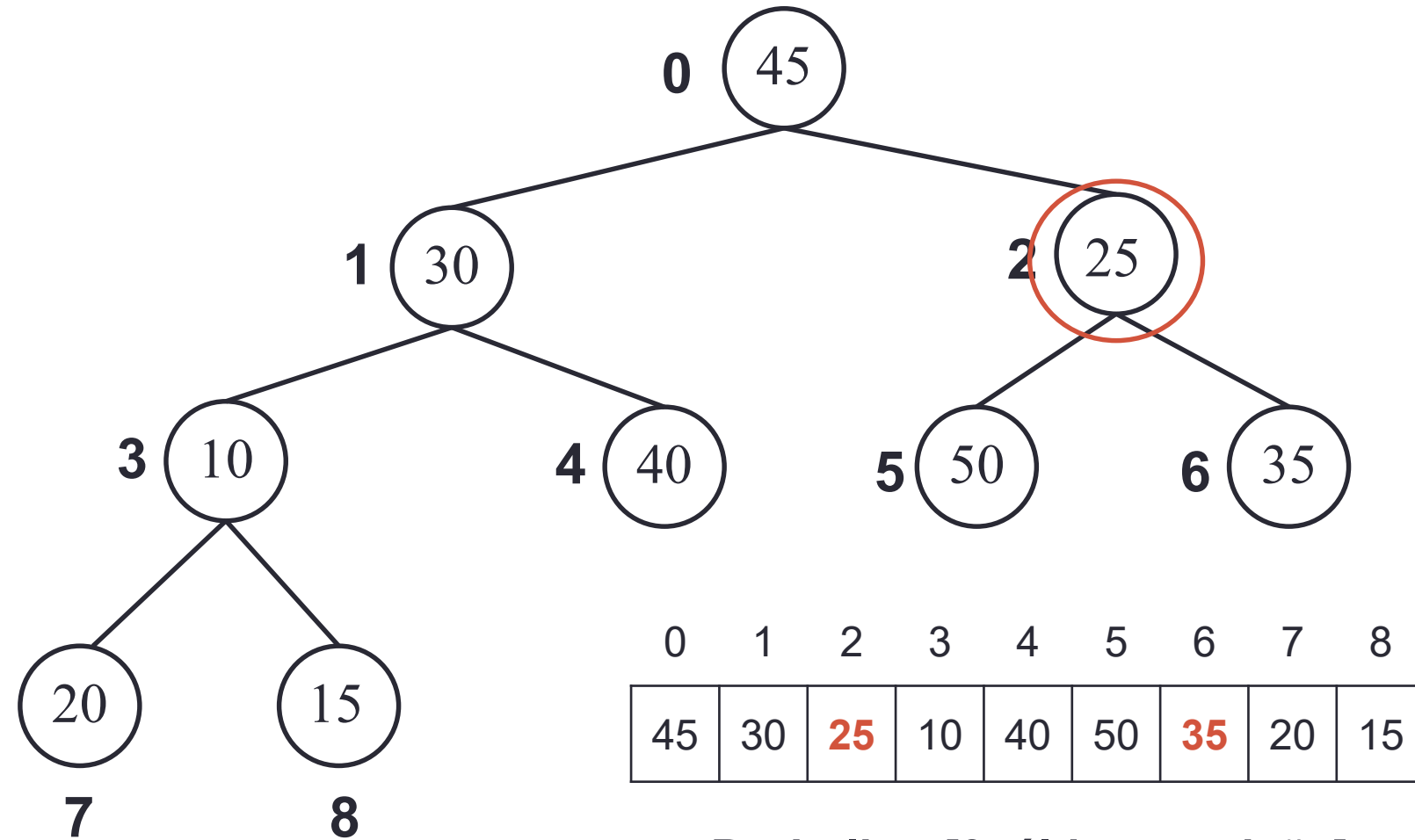
Borbulhar descendente para índice 2



Borbulhar [2..última posição]

8, 7, 6, 5, 4 e 3 são Heaps

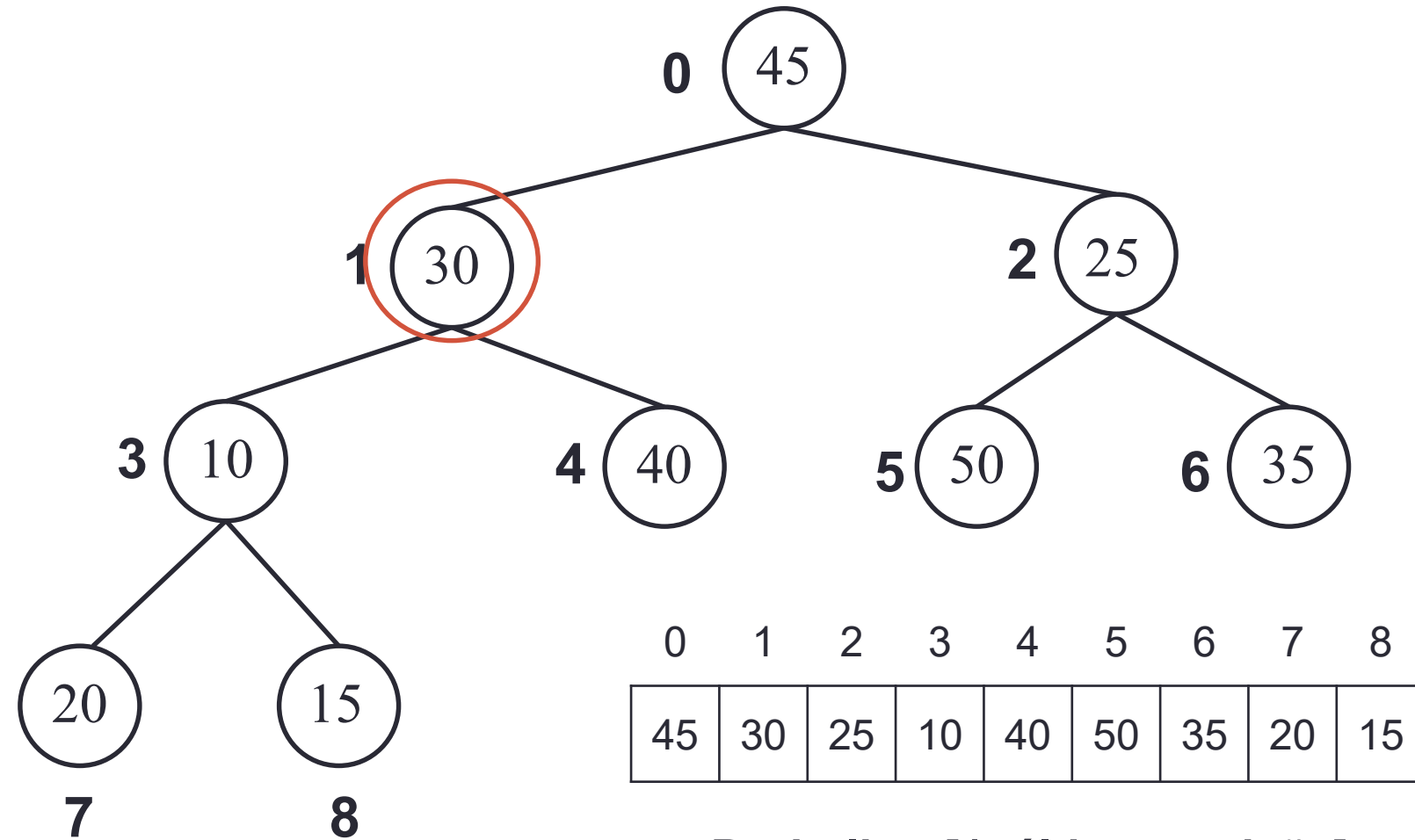
Borbulhar descendente para índice 2



Borbulhar [2..última posição]

8, 7, 6, 5, 4, 3 e 2 são Heaps

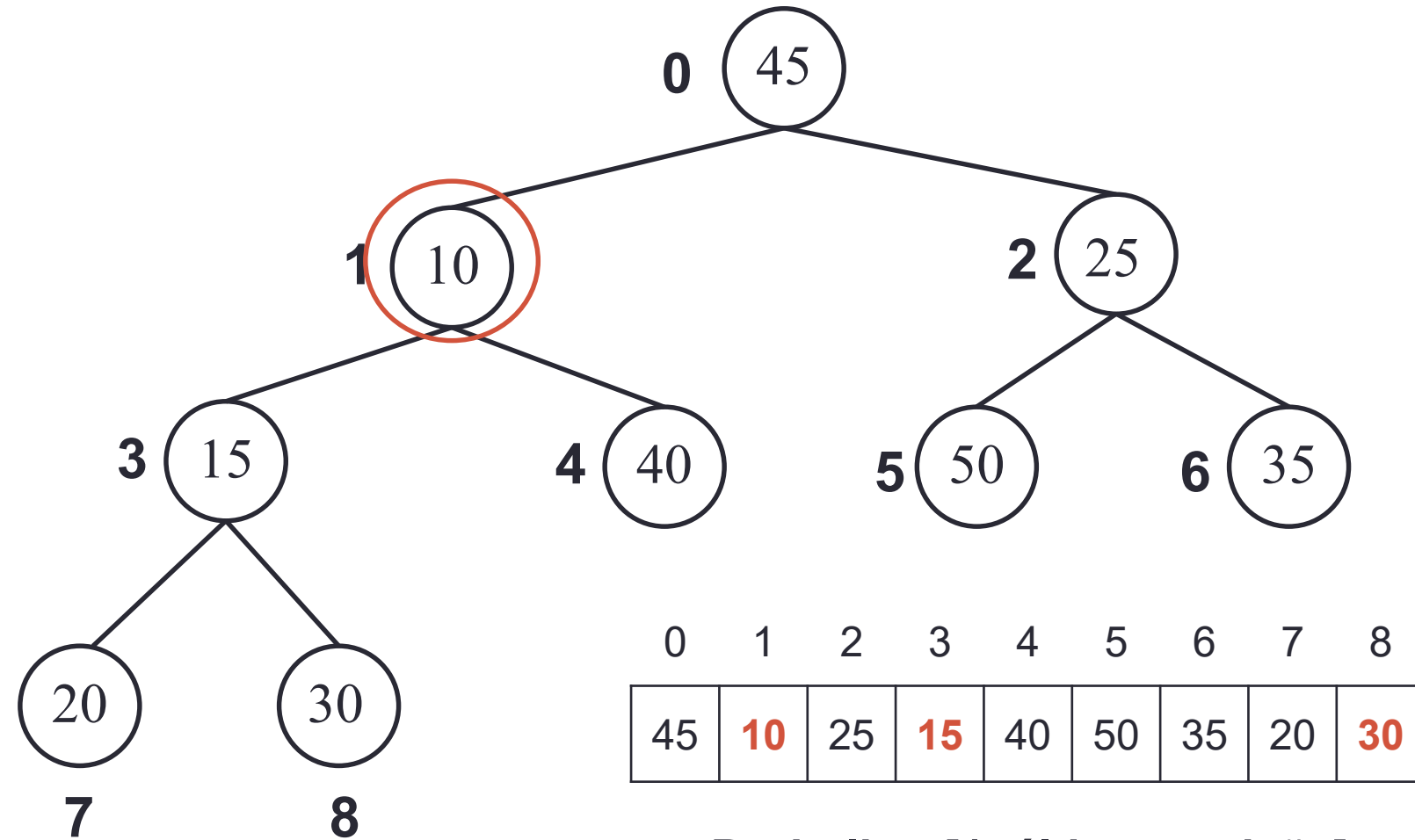
Borbulhar descendente para índice 1



Borbulhar [1..última posição]

8, 7, 6, 5, 4, 3 e 2 são Heaps

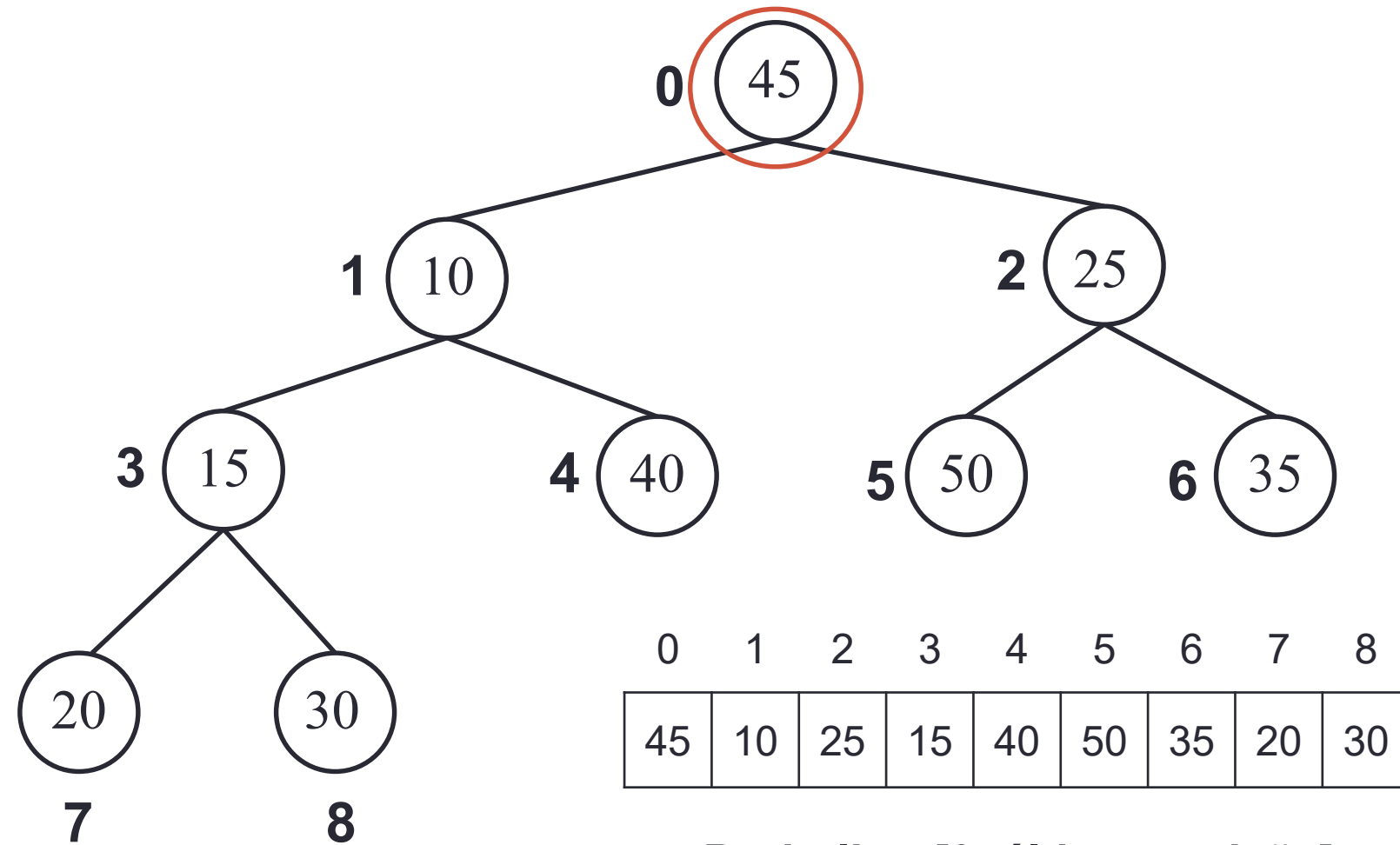
Borbulhar descendente para índice 1



Borbulhar [1..última posição]

8, 7, 6, 5, 4, 3, 2 e 1 são Heaps

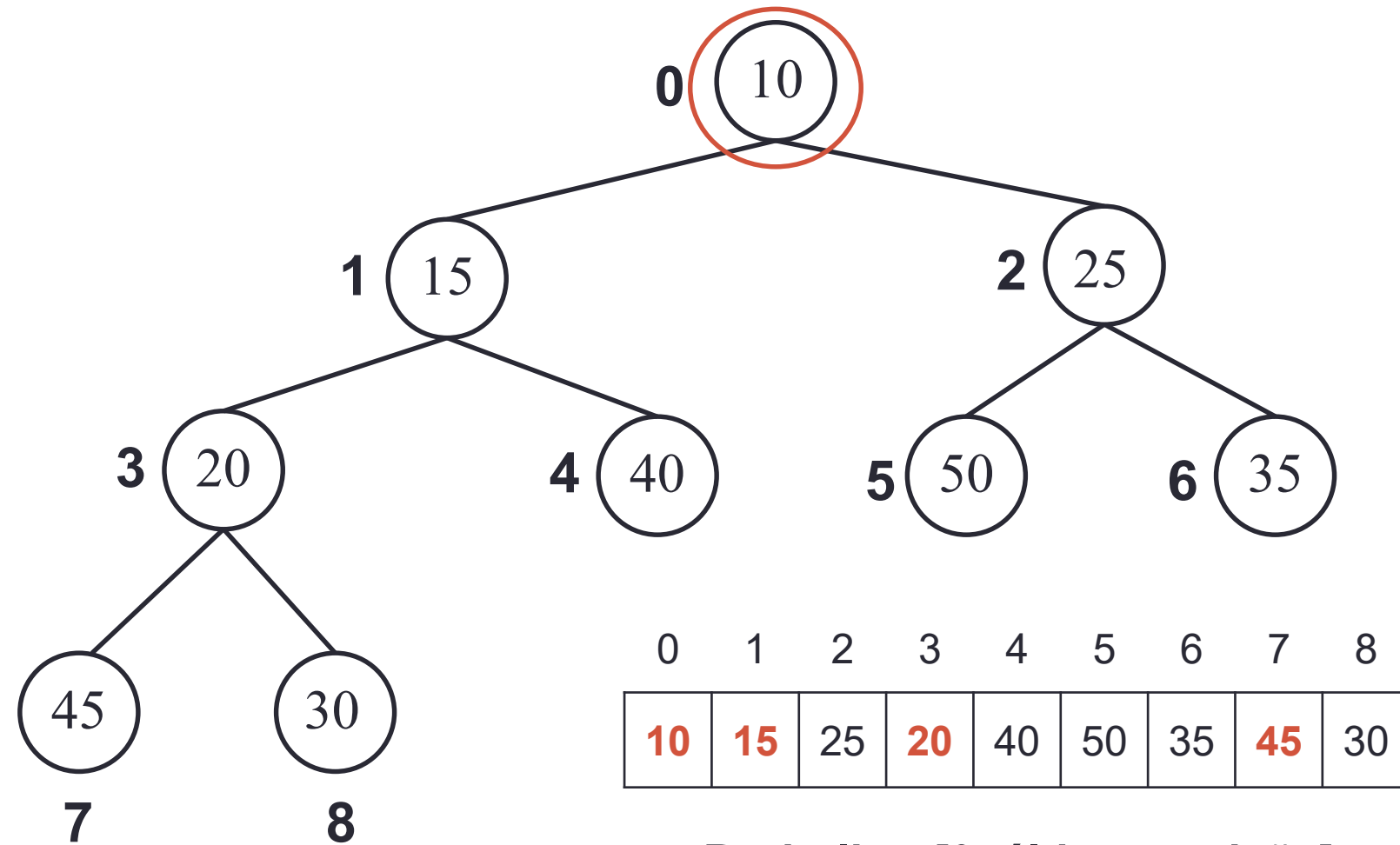
Borbulhar descendente para índice 0



Borbulhar [0..última posição]

8, 7, 6, 5, 4, 3, 2 e 1 são Heaps

Borbulhar descendente para índice 0

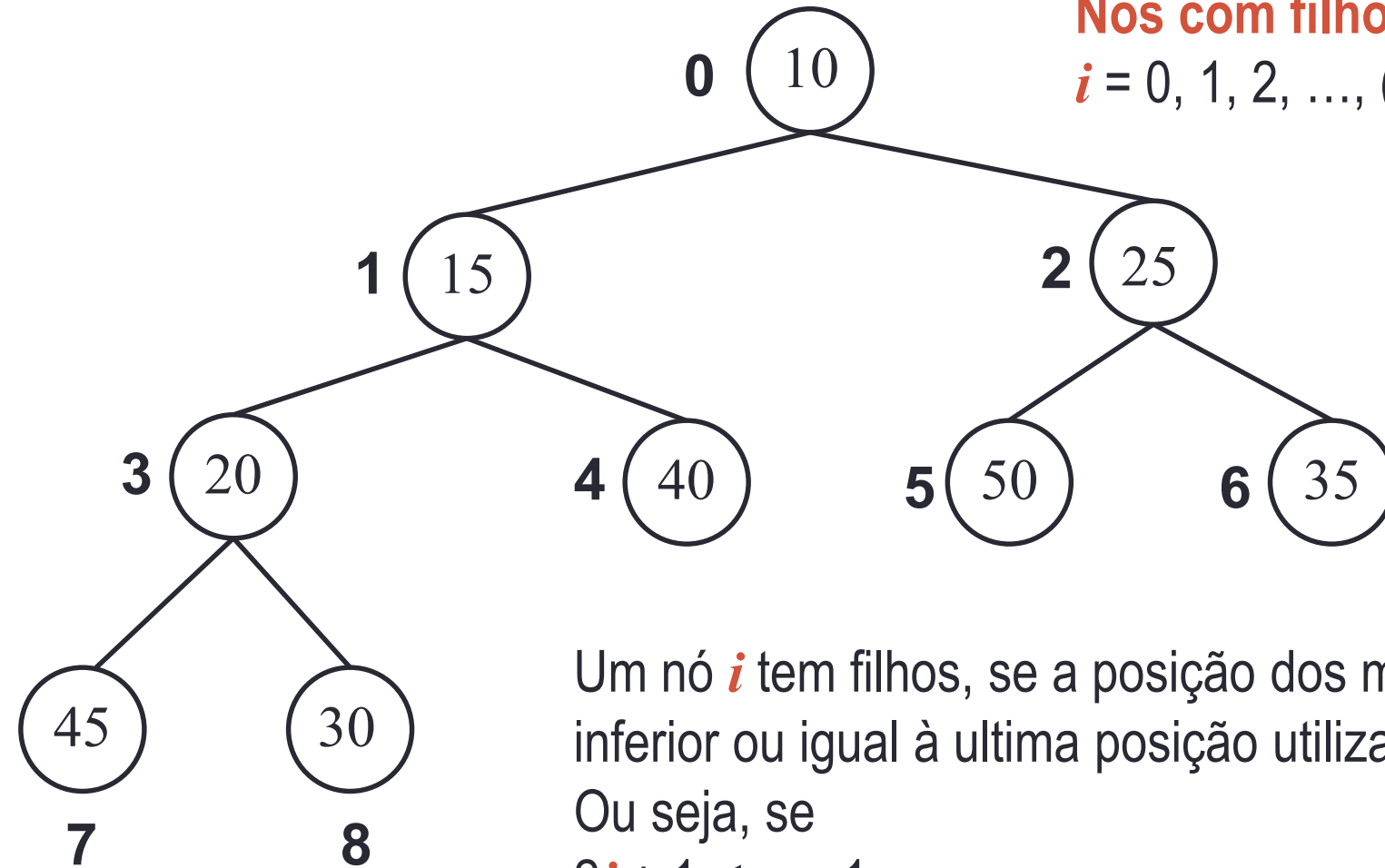


Borbulhar [0..última posição]
8, 7, 6, 5, 4, 3, 2, 1 e 0 são Heaps

Localizar Nós com filhos

Nós com filhos:

$i = 0, 1, 2, \dots, (n - 2) \text{ div } 2$



Um nó i tem filhos, se a posição dos mesmos for inferior ou igual à ultima posição utilizada, no vetor.

Ou seja, se

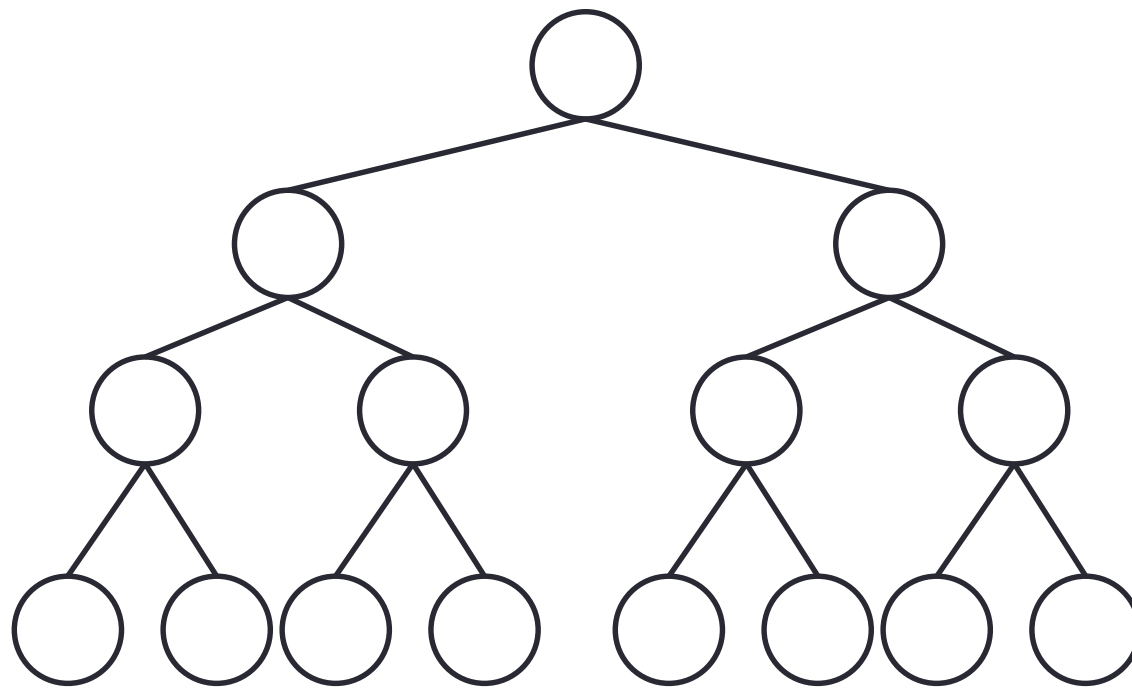
$$2i + 1 \leq n - 1$$

$$i \leq (n - 2) \text{ div } 2$$

Criação de Heap a partir de Vetor

```
public MinHeap( Entry<K,V>[] theArray ) {  
    // Build a complete tree.  
    this.buildArray(theArray.length, theArray);  
    currentSize = theArray.length;  
    // Build a priority tree.  
    this.buildPriorityTree();  
}  
  
protected void buildPriorityTree( ) {  
    for ( int i = (currentSize - 2) / 2; i >= 0; i-- )  
        this.percolateDown(i);  
}
```

Número máximo de Trocas (árvore de altura h)



Nível	Nós	Trocas
1	$1 = 2^0$	3
2	$2 = 2^1$	2
3	$4 = 2^2$	1
4	$8 = 2^3$	0
i	2^{i-1}	$h - i$

$$\text{maxTrocas} = \sum_{i=1}^h 2^{i-1} (h - i) = 2^h - h - 1$$

Criação de Heap a partir de Vetor - Complexidade

Para uma árvore com altura h , o número máximo de trocas será:

$$\text{maxTrocas} = 2^h - h - 1$$

Quando $n \geq 2$ (e $h \geq 2$), e sabendo que os $h - 1$ primeiros níveis estão completamente preenchidos, temos que:

- O número de nós no nível $h-1$ é menor ou igual ao número de nós na árvore

$$2^{h-1} \leq n,$$

logo

$$2^h \leq 2n.$$

Portanto,
e

$$\text{maxTrocas} \leq 2n = O(n)$$

$$\text{Criar Heap a partir de Vetor } (n) = O(n)$$

Complexidades da Fila com Prioridade em Heap (com n entradas)

	Melhor Caso	Pior Caso	Caso Esperado
new (vazia)	$O(1)$	$O(1)$	$O(1)$
new (de vetor[n])	$O(n)$	$O(n)$	$O(n)$
isEmpty	$O(1)$	$O(1)$	$O(1)$
size	$O(1)$	$O(1)$	$O(1)$
minEntry	$O(1)$	$O(1)$	$O(1)$
removeMin	$O(1)$	$O(\log n)$	$O(\log n)$
insert	$O(1)$	$O(n) *$	$O(\log n)$

* Seria $O(\log n)$ se o vetor não fosse extensível

LibrarianBook

```
package library;
```

```
interface LibrarianBook extends Book {
```

```
    //removes reader from reservation queue
```

```
    //returns next reader with the highest priority
```

```
    Reader nextPriorityReader()
```

```
        throws NoReservationsException();
```

```
    //adds reader to reservation queue
```

```
    void addReservation(Reader reader);
```

```
}
```

Implementação de
Reader deve conter
(e.g.)

- Nome
- Número Leitor
- Categoria
- Prioridade
- Data de inscrição
- Validade

LibrarianBookClass (1) – Actualizada

```
package library;
import dataStructures.*;

class LibrarianBookClass implements LibrarianBook {
    public static final int INITIAL_MAX_RESERVATIONS=200;
    protected String author;
    ...
    protected MinPriorityQueue<Integer, Reader> reservations;

    public LibrarianBookClass(String author, long ISBN,
        String title, String subject, String code,
        String publisher) {
        this.author=author;
        ...
        reservations=new MinHeap<Integer,
            Reader>(INITIAL_MAX_RESERVATIONS);
    }
}
```

LibrarianBookClass (3) – não completa

```
//removes reader from reservation queue
//returns next reader with the highest priority
//Requires: !noReservations()
Reader nextPriorityReader(){
    throws NoReservationsException{
        if (this.noReservations())
            throw new NoReservationsException();
        Reader reader = reservations.removeMin();
        return reader;
    }

    //adds reader to reservation queue
    public void addReservation(Reader reader) {
        reservations.insert(reader.getPriority(), reader);
    }
} // End of LibrarianBookClass.
```

Exercício 1(1)

- Assuma que todas as entradas de uma fila com prioridade têm chaves distintas. Neste contexto, pode-se afirmar que o método `minEntry` da interface `MinPriorityQueue` retorna a entrada com a menor chave.
- Suponha que se acrescenta um novo método àquela interface (chamado `secondMinEntry`), que devolve a entrada com a segunda menor chave.
- Por exemplo, se a fila com prioridade tivesse as entradas (onde a chave é o primeiro elemento do par) `(3, 'z')`, `(4, 'a')`, `(7, 'b')` e `(10, 'j')`, `minEntry` e `secondMinEntry` retornariam `(3, 'z')` e `(4, 'a')`, respectivamente.

Exercício 1 (2)

- Implemente o método `secondMinEntry` na classe `MinHeap` e programe todos os métodos auxiliares de que necessitar.
- Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando. Assuma que a complexidade temporal do método `compareTo`, quando aplicado a instâncias do tipo `K`, é sempre constante.

Exercício 2 – Fila com Prioridade Inteligente

```
public interface SmartMinPriorityQueue<K extends Comparable<K>, V> {  
  
    // 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 EmptyPriorityQueueException;  
  
    // If the priority queue contains an entry of the form  
    // (otherKey, value), with otherKey <= key, this method does nothing  
    // except returning false; otherwise, inserts the entry (key, value)  
    // in the priority queue and returns true.  
    boolean 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 EmptyPriorityQueueException;  
}
```

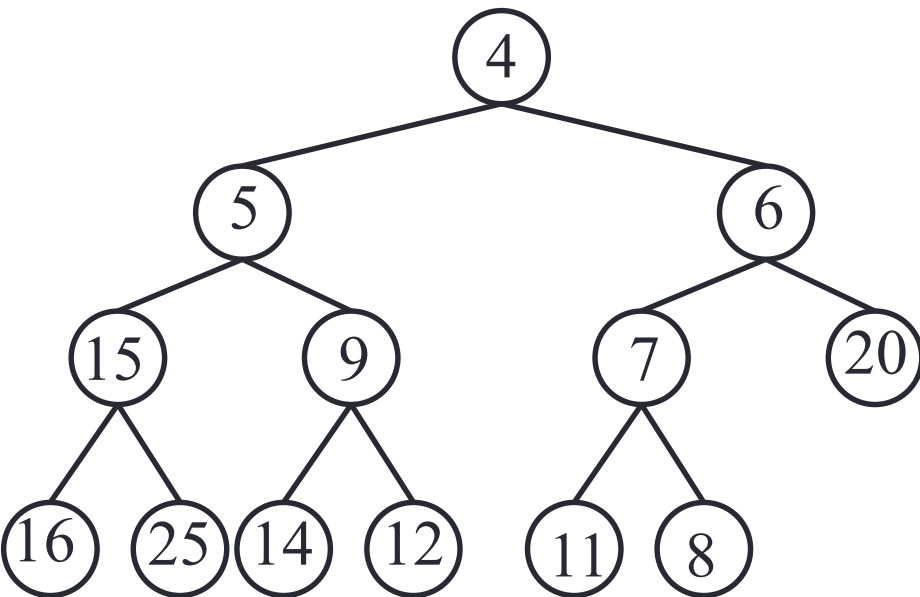
Exercício 2 (2)

- Explícite detalhadamente as estruturas de dados mais adequadas para implementar a interface `SmartMinPriorityQueue`, descreva brevemente como implementaria as quatro operações e calcule as suas complexidades temporais, no caso esperado, justificando.
- Assuma que a complexidade temporal do método `compareTo`, quando aplicado a instâncias do tipo `K`, é sempre constante. Pode assumir que os valores também são comparáveis entre si (`V` extends `Comparable<V>`), com complexidade constante.

Exercício 3

- Qual a sequência mínima de inserts e/ou removeMin necessárias para transformar o Heap A no Heap B ? Desenhe o heap depois de cada operação.

Heap A



Heap B

