Colas de Prioridad y Heaps

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025
Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET

TAD Cola de prioridad

- Una cola de prioridad es una cola en la que los elementos tienen una prioridad, y los elementos que se eliminan de la cola son los de mayor prioridad (max queue)
  - También se pueden implementar colas en las que se eliminan los elementos de menor prioridad (min queue)
- Requiere que los elementos de la cola tengan un orden (respecto de sus prioridades)
  - Ej: Que extiendan la interfaz Comparable de Java
- Matemáticamente, vamos a representar las colas de prioridad con secuencias ordenadas $s = [e_1, e_2, \ldots, e_n]$, donde $e_1 \leq e_2 \leq \ldots \leq e_n$
- Utiles para en diferentes aplicaciones: planificación de tareas, scheduling de procesos, sistemas de simulación, optimización del uso de la red, etc.

TAD Cola de prioridad: Operaciones

```java
/**
 * We represent PriorityQueues with unbounded sequences of
 * objects of type T, where the elements are sorted in
 * ascending order with respect to their priorities.
 * A typical PriorityQueue is [o1, o2, . . . , on],
 * where o1 <= o2 <= ... <= on.
 *
 * PriorityQueue requires that the key type T implements the
 * Comparable interface.
 *
 * The methods use compareTo to determine equality of elements.
 */
public interface PriorityQueue<T extends Comparable<? super T>> {
```

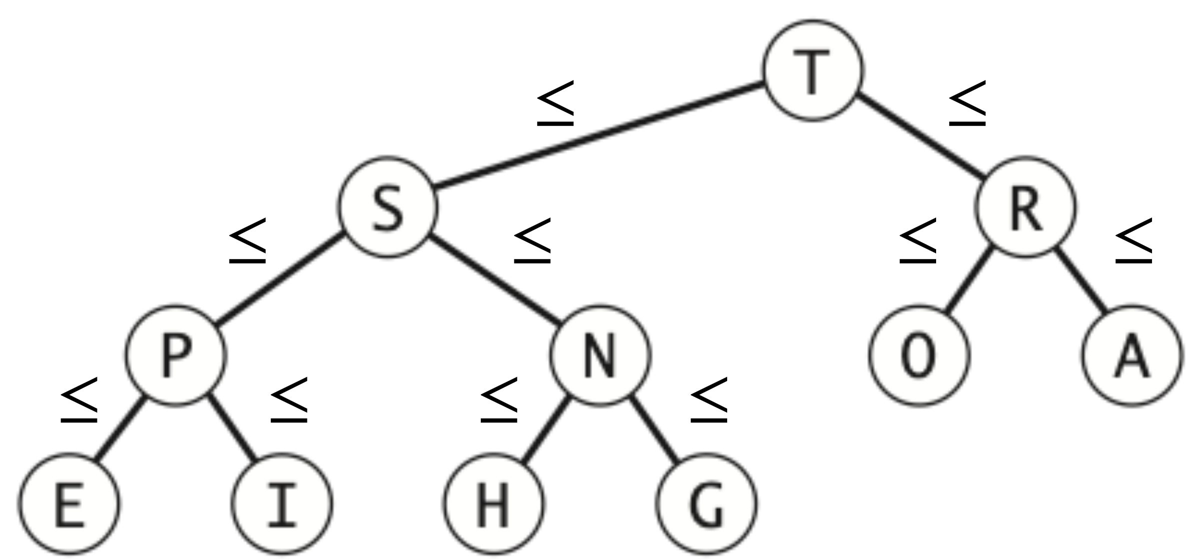
TAD Cola de prioridad: Operaciones

```java
/**
 * We represent PriorityQueues with u
 * objects of type T, where the eleme
 * ascending order with respect to th
 * A typical PriorityQueue is [o1, o2
 * where o1 <= o2 <= ... <= on.
 *
 * PriorityQueue requires that the ke
 * Comparable interface.
 *
 * The methods use compareTo to deter
 */
public interface PriorityQueue<T exte
```

```java
/**
 * @post Inserts x to the queue.
 */
public void insert(T x);
/**
 * @post Returns true iff the queue contains no elements.
 *   More formally, it satisfies: result = #this = 0.
 */
public boolean isEmpty();
/**
 * @post Returns the number of elements in the queue.
 *   More formally, it satisfies: result = #this.
 */
public int size();
/**
 * @pre !isEmpty()
 * @post Returns the largest element of the queue.
 */
public T max();
/**
 * @post Deletes and returns the largest element of the queue.
 */
public T removeMax();
```

Heaps

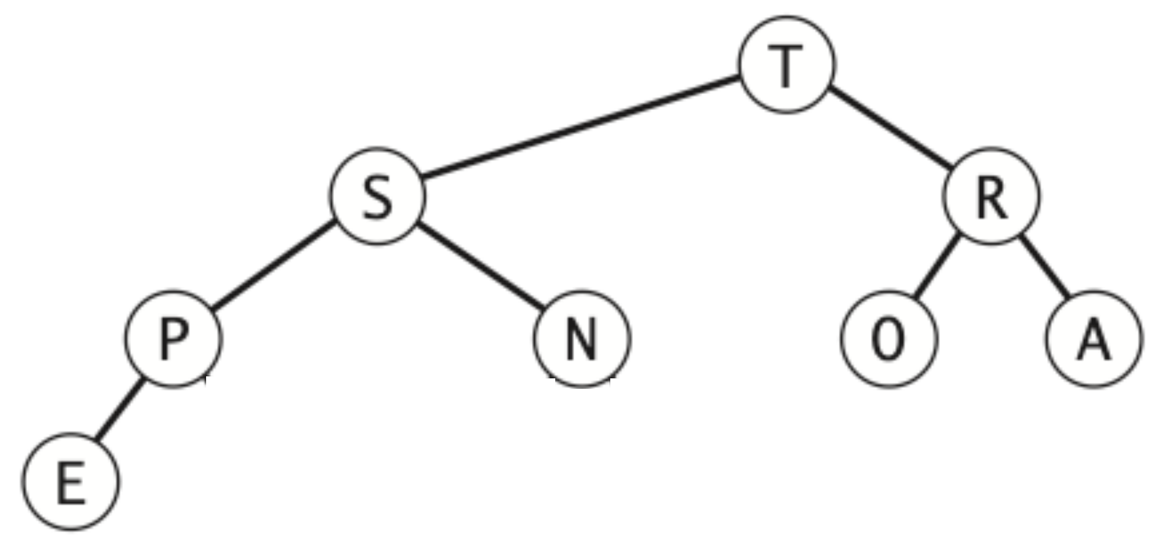
- Los heaps son una implementación eficiente y elegante de colas de prioridad
- Son árboles binarios, usualmente implementados con arreglos
- La raíz de cada subárbol es mayor o igual que los hijos, y esta propiedad se cumple recursivamente (max heaps)
  - También existen los min heaps en los que la raiz es menor o igual que los hijos
- Son completos: cada nivel tiene todos los nodos, excepto el último en donde pueden faltar algunos nodos a la derecha
- La altura de un heap es O(log n), donde n es la cantidad de elementos en el heap



Altura de un heap

- Recordemos que: $full(t) \implies n = 2^h - 1$
- La menor cantidad de nodos n que tiene un heap con altura h es la altura del nivel anterior más uno (ver Figura):

$n \geq 2^{h-1} - 1 + 1$
{ tomando logaritmos }
$log_2 n \geq log_2 2^{h-1}$
{ simplificando }
$log_2 n + 1 \geq h$



- Es decir: $h \in O(log_2 n)$

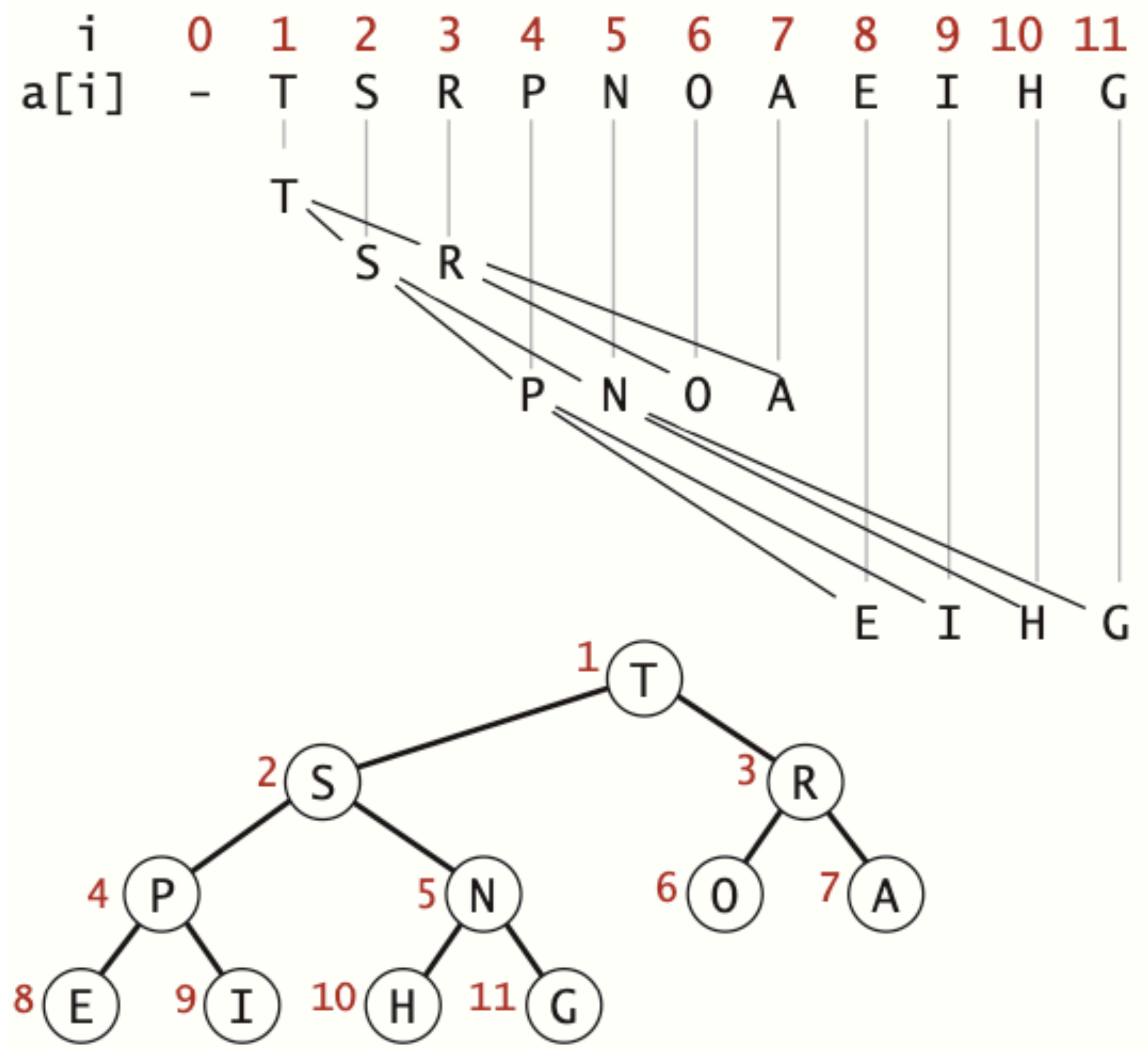
Posibles implementaciones de colas de prioridad

- Tasa de crecimiento en el peor caso de las operaciones de cola de prioridad con diferentes implementaciones:

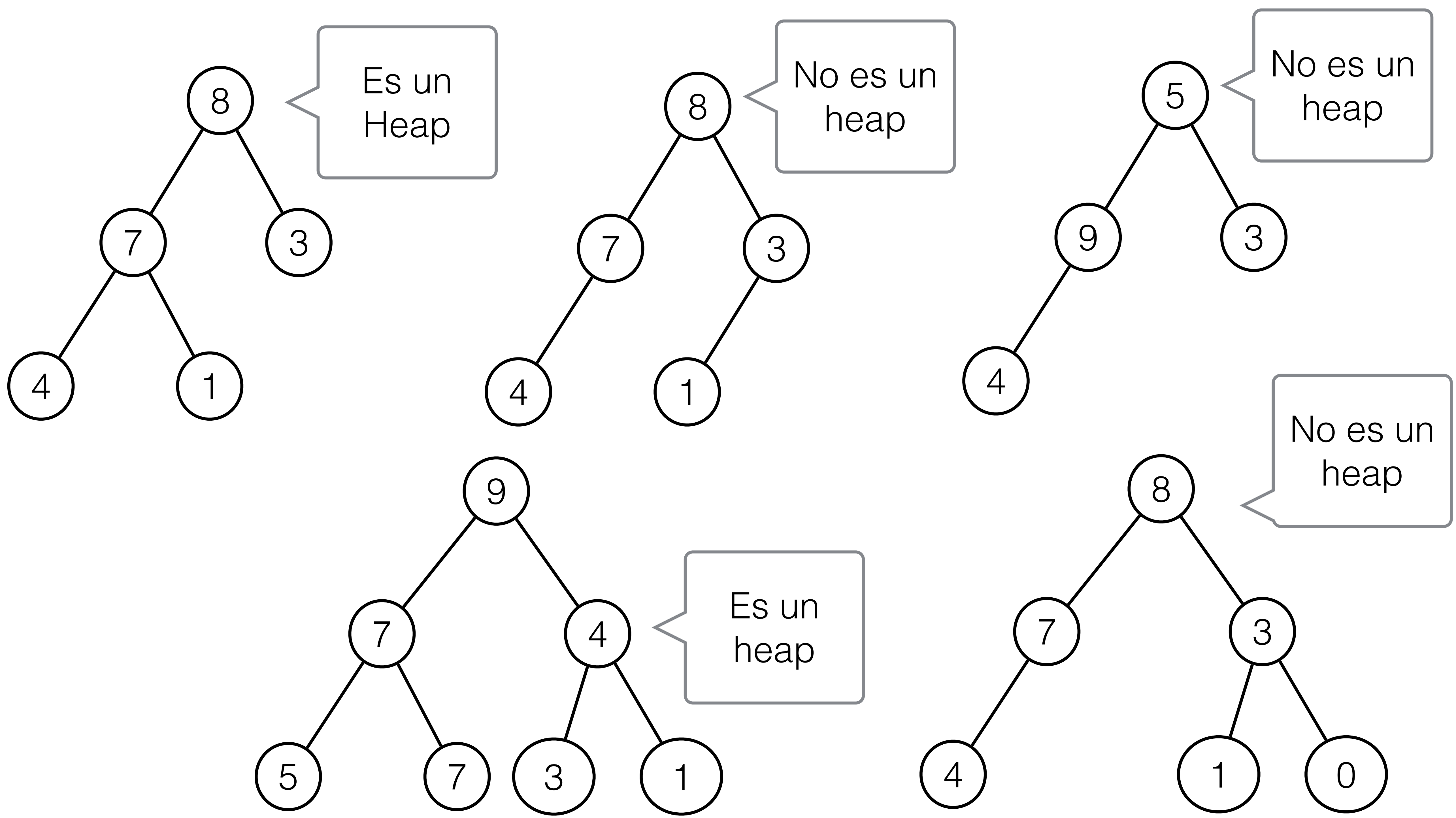
data structure	insert	remove maximum
ordered array	N	1
unordered array	1	N
heap	$\log N$	$\log N$

Implementación de heaps con arreglos

- Como el heap es un árbol completo, podemos almacenar los valores en posiciones contiguas de un arreglo
- Por simplicidad almacenaremos la raiz en la posición 1
  - La posición 0 la dejaremos libre
- Los hijos izquierdo y derecho del nodo en la posición k están en:
  - left(k) = 2*k
  - right(k) = 2*k+1
- Podemos obtener el padre del nodo en la posición k con:
  - parent(k) = (int) k/2 (parte entera de la division)



Heaps: Invariante de representación



```java
/**
 * HeapPriorityQueue is a priority queue implementation using
 * heaps.
 *
 * We represent PriorityQueues with unbounded sequences of
 * objects of type T, where the elements are sorted in
 * ascending order with respect to their priorities.
 * A typical PriorityQueue is [o1, o2, . . . , on],
 * where o1 <= o2 <= ... <= on.
 *
 * HeapPriorityQueue requires that the key type T implements the
 * Comparable interface.
 *
 * The methods use compareTo to determine equality of elements.
 */
public class HeapPriorityQueue<T extends Comparable<? super T>>
    implements PriorityQueue<T>
{
    // array where items are stored at indices 1 to n
    private T[] queue;
    // number of items on priority queue
    private int size;
    // initial capacity of underlying resizing array
    private static final int INIT_CAPACITY = 8;

    /**
     * @post Creates an empty priority queue.
     *   More formally, it satisfies: this = [].
     */
    public HeapPriorityQueue() {
        queue = (T[]) new Comparable[INIT_CAPACITY+1];
        size = 0;
    }
```

```java
/**
 * HeapPriorityQueue is a priority queue implementation using
 * heaps.
 *
 * We represent PriorityQueues with unbounded sequences of
 * objects of type T, where the elements are sorted in
 * ascending order with respect to their priorities.
 * A typical PriorityQueue is [o1, o2, . . . , on],
 * where o1 <= o2 <= ... <= on.
 *
 * HeapPriorityQueue requires that the key type T implements the
 * Comparable interface.
 *
 * The methods use compareTo to determine equality of elements.
 */
public class HeapPriorityQueue<T extends Comparable<? super T>>
    implements PriorityQueue<T>
{
    // array where items are stored at indices 1 to n
    private T[] queue;
    // number of items on priority queue
    private int size;
    // initial capacity of underlying resizing array
    private static final int INIT_CAPACITY = 8;

    /**
     * @post Creates an empty priority queue.
     *   More formally, it satisfies: this = [].
     */
    public HeapPriorityQueue() {
        queue = (T[]) new Comparable[INIT_CAPACITY+1];
        size = 0;
    }
```

Un elemento más porque no usamos la posición 0

Heaps: Invariante de representación

```java
/**
 * @post Returns true if and only if the structure is a
 *   valid max heap.
 */
public boolean repOK() {
    for (int i = 1; i <= size; i++) {
        if (queue[i] == null) return false;
    }
    for (int i = size+1; i < queue.length; i++) {
        if (queue[i] != null) return false;
    }
    if (queue[0] != null) return false;
    return isMaxHeapOrdered(1);
}
```

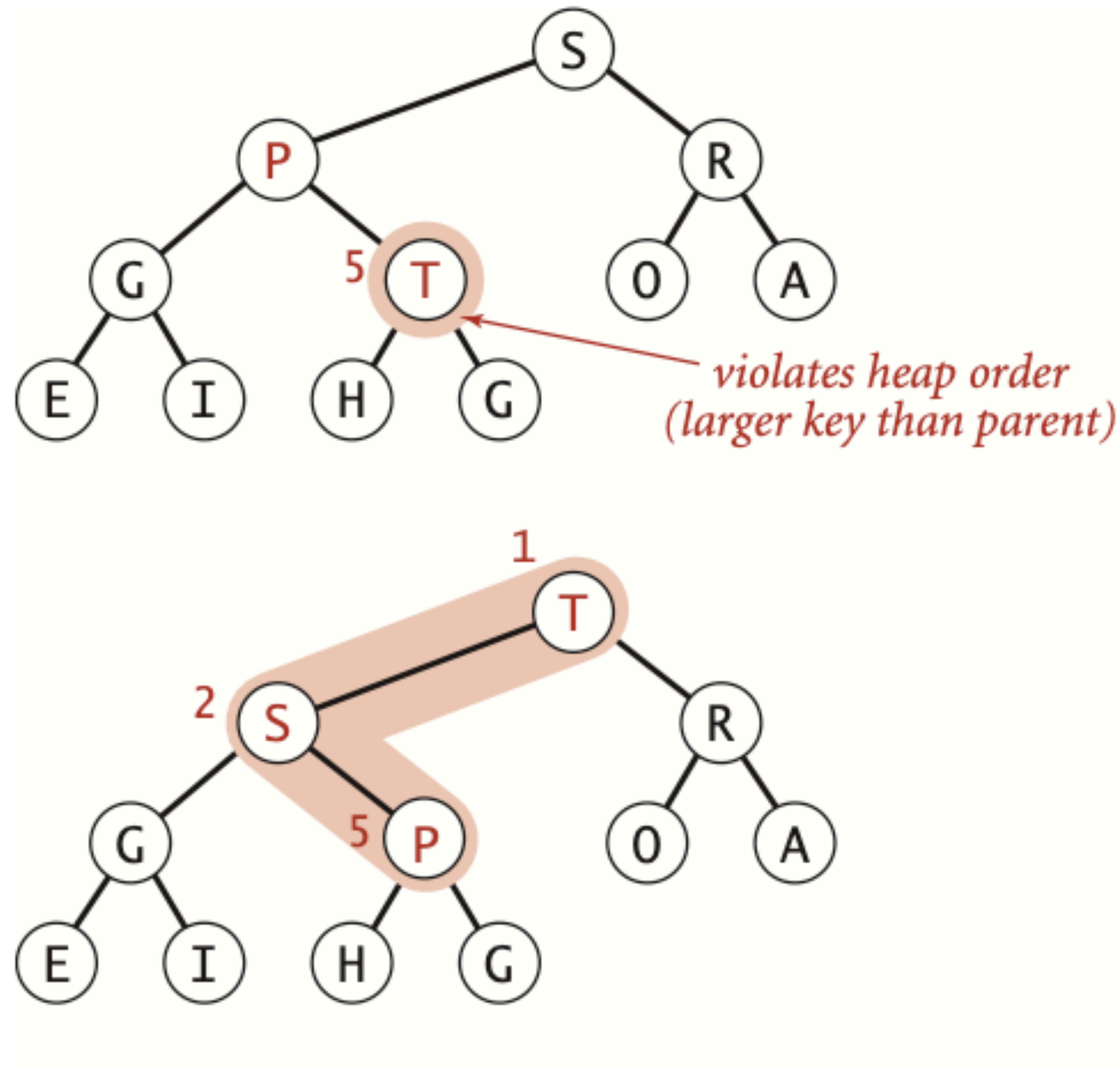
Heaps: Invariante de representación

```java
/**
 * @post Returns true if and only if the structure is a
 *   valid max heap.
 */
public boolean repOK() {
    for (int i = 1; i <= size; i++) {
        if (queue[i] == null) return false;
    }
    for (int i = size+1; i < queue.length; i++) {
        if (queue[i] != null) return false;
    }
    if (queue[0] != null) return false;
    return isMaxHeapOrdered(1);
}
```

```java
/**
 * @post Returns true if and only if the subtree with
 *   root k is a valid max heap.
 */
private boolean isMaxHeapOrdered(int k) {
    if (k > size)
        return true;
    int left = 2*k;
    int right = 2*k + 1;
    if (left <= size && less(queue[k], queue[left]))
        return false;
    if (right <= size && less(queue[k], queue[right]))
        return false;
    return isMaxHeapOrdered(left) && isMaxHeapOrdered(right);
}
```

Restaurar invariante de heaps: swim

- Las operaciones de heaps pueden violar temporalmente el invariante de orden de los valores del heap
- Para restaurarlo existen dos operaciones: swim y sink
- swim(k) toma la posición de un nodo que posiblemente viola la propiedad, y va intercambiando el valor del nodo con el de sus padres hasta llevar el valor a la posición correcta
  - Opera de abajo hacia arriba (bottom-up)
- De esta manera restaura el invariante de orden del heap



Bottom-up reheapify (swim)

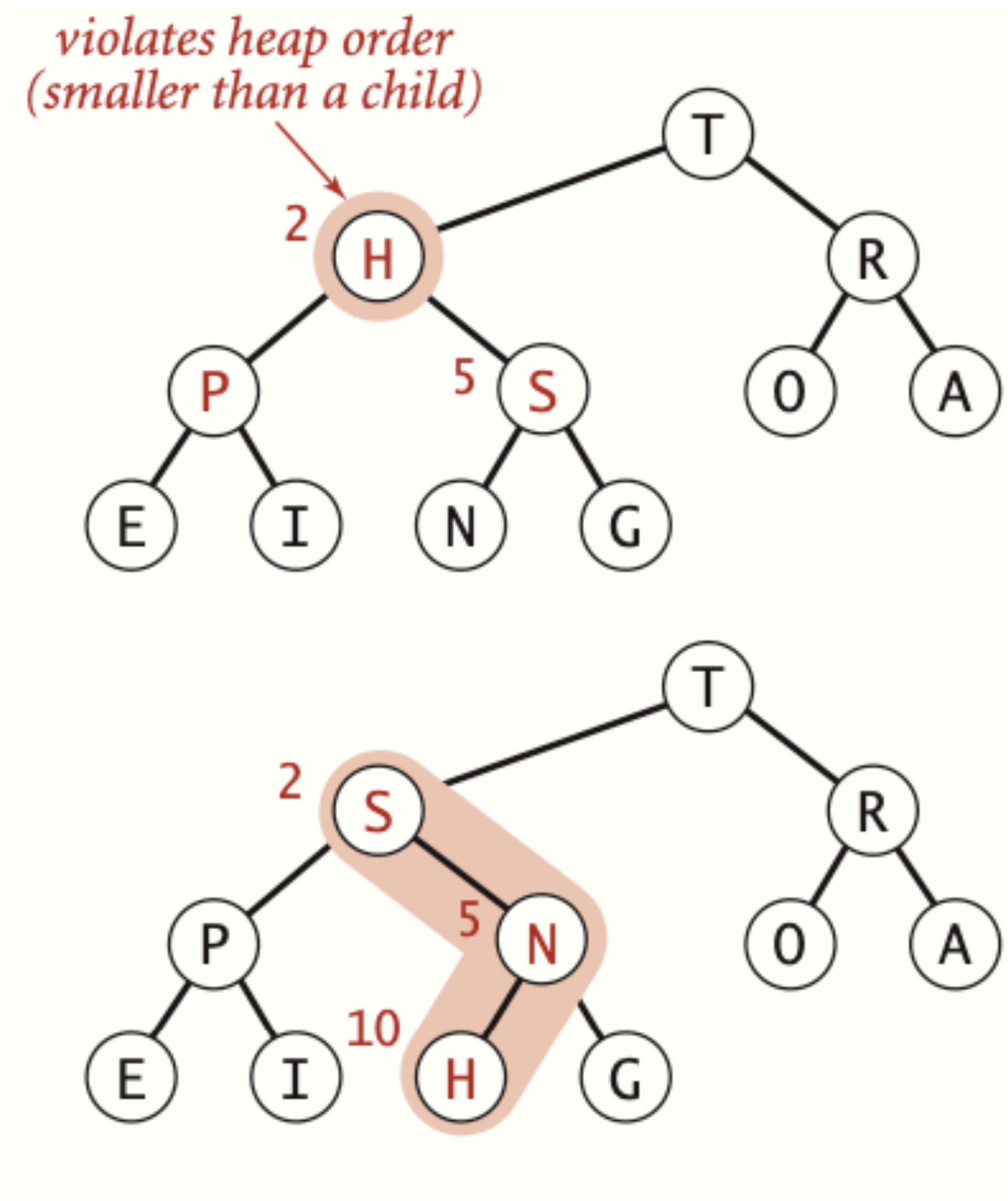
Restaurar invariante de heaps: swim

```java
/**
 * @pre 1<=k<=size()
 * @post Swaps the value of the node at position k with its
 *   parents until the heap order invariant is satisfied.
 */
private void swim(int k) {
    while (k > 1 && less(queue[k/2], queue[k])) {
        exch(queue, k/2, k);
        k = k/2;
    }
}
```

- Si la altura del heap es h, swim hace a lo sumo $h - 1$ comparaciones
  - El peor caso es comparar todos los valores de los nodos desde desde una hoja hasta la raíz
- Esto es, swim es $O(h) = O(log_2 n)$

Restaurar invariante de heaps: sink

- Las operaciones de heaps pueden violar temporalmente el invariante de orden de los valores del heap
- Para restaurarlo existen dos operaciones: swim y sink
- sink(k) toma la posición de un nodo que posiblemente viola la propiedad, y va intercambiando el valor del nodo con el mayor de sus hijos hasta llevar el valor a la posición correcta
  - Opera de arriba hacia abajo (top-down)
- De esta manera restaura el invariante de orden del heap



Top-down reheapify (sink)

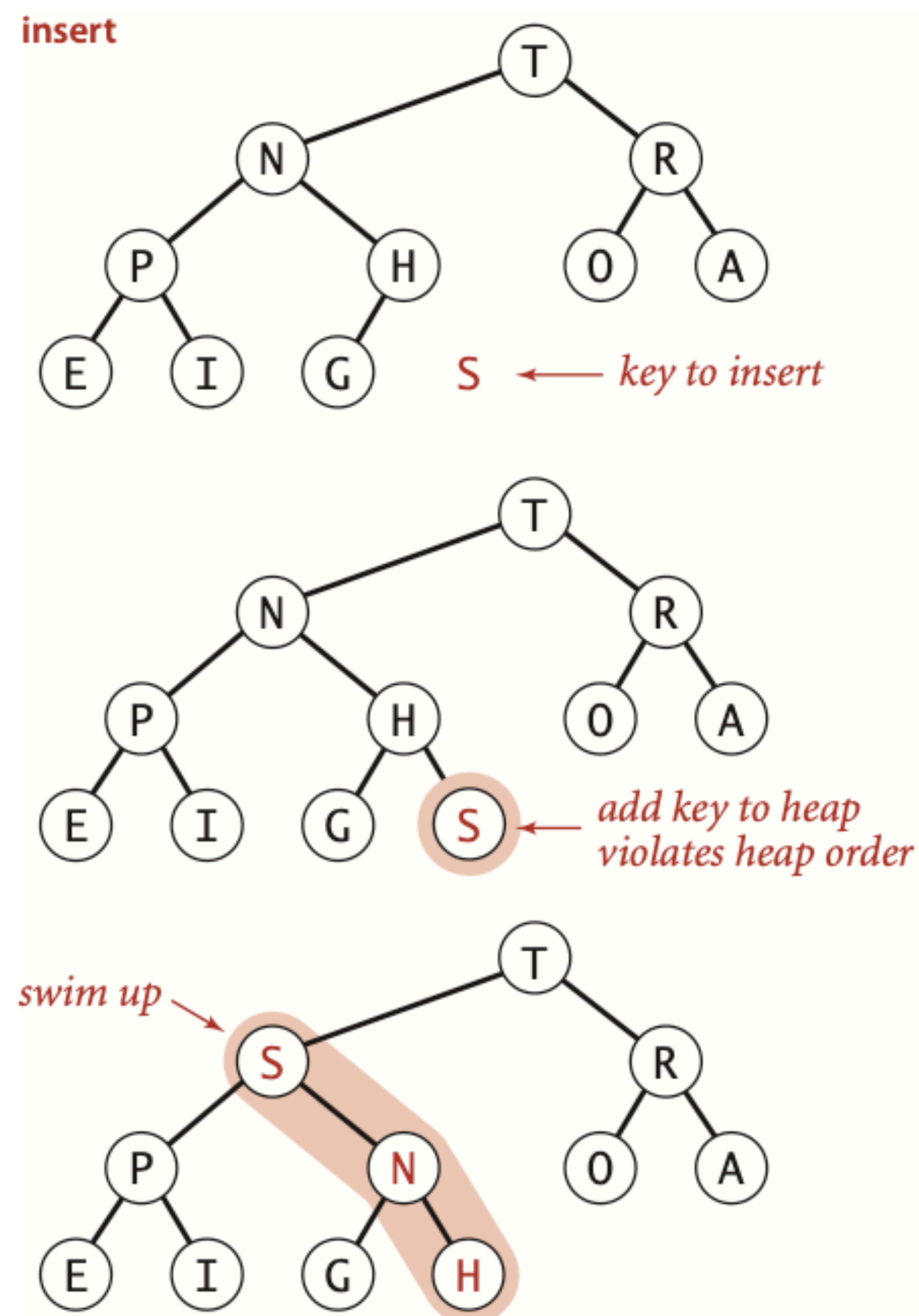
Restaurar invariante de heaps: sink

```java
/**
 * @pre 1<=k<=size()
 * @post Swaps the value of the node at position k with its
 *   descendants until the heap order invariant is satisfied.
 */
private void sink(int k) {
    while (2*k <= size) {
        int j = 2*k;
        if (j < size && less(queue[j], queue[j+1]))
            j++;
        if (!less(queue[k], queue[j]))
            break;
        exch(queue, k, j);
        k = j;
    }
}
```

- Si la altura del heap es h, sink hace $2 * (h - 1)$ comparaciones
    - El peor caso es comparar todas las raíces con los dos hijos, en el camino desde la raíz hasta una hoja
- Esto es, sink es $O(h) = O(log_2 n)$

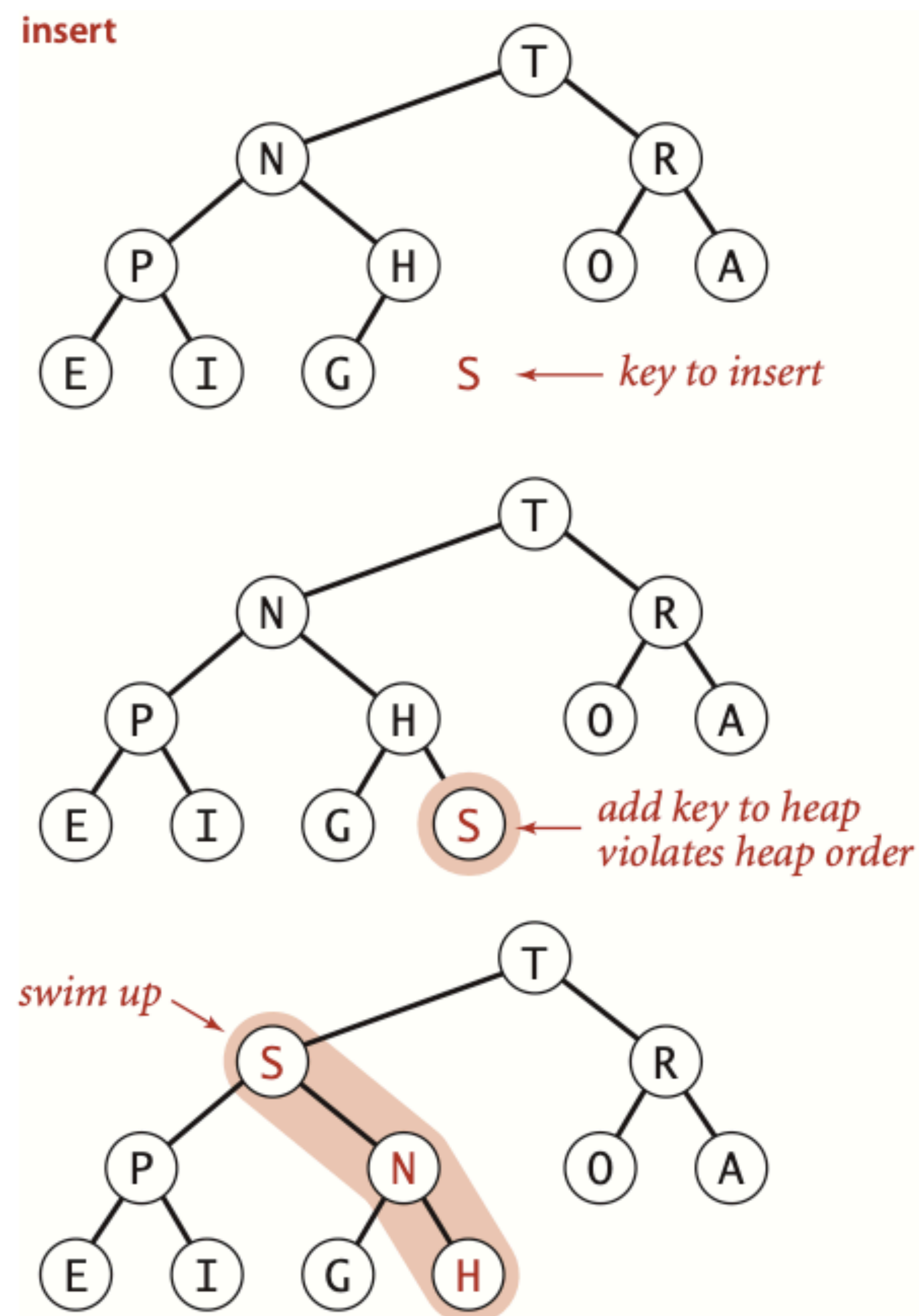
Insertar en un heap

- Se inserta en la hoja más a la derecha
- Luego se intercambia esa clave hacia arriba usando swim, hasta encontrar su lugar en el árbol
- Insertar en la hoja más a la derecha toma tiempo constante
- swim es O(log n), por lo que insertar es O(log n) (si el arreglo es lo suficientemente grande)



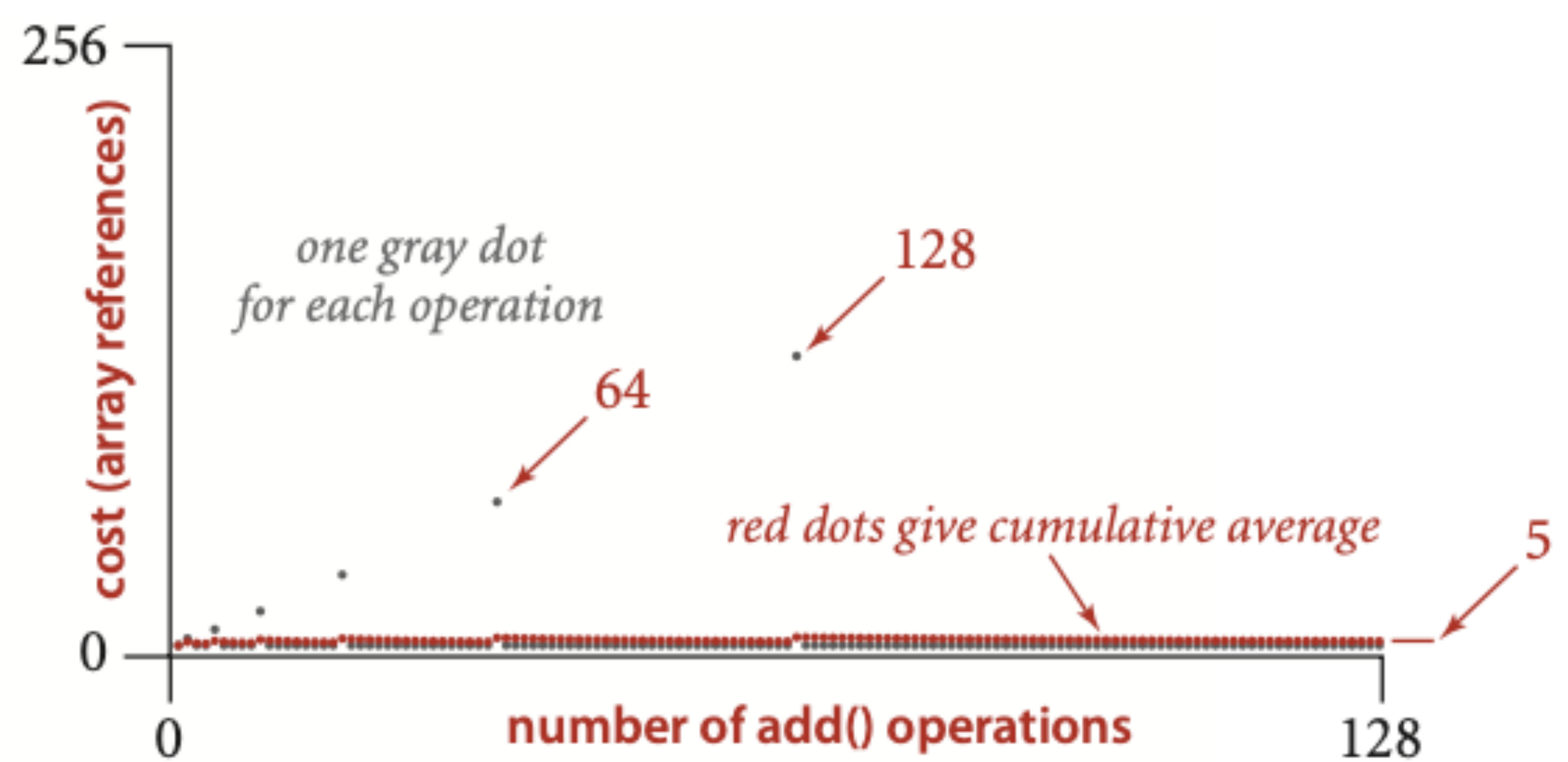
Insertar en un heap

```java
/**
 * @post Inserts x to the queue.
 */
public void insert(T x) {
    // double size of array if necessary
    if (size == queue.length - 1)
        resize(2 * queue.length);
    // add x, and percolate it up to maintain
    // heap invariant
    queue[++size] = x;
    swim(size);
}
```



Análisis amortizado de tiempo de ejecución

- En algunos casos, el tiempo de ejecución en el peor caso puede dar una cota demasiado gruesa
- Un ejemplo típico es cuando tenemos estructuras implementadas con arreglos, y vamos duplicando el tamaño del arreglo a medida que se necesita para poder almacenar más elementos
- Por ej., el peor caso de push para pilas implementadas con arreglos sería O(n), ya que debemos crear un arreglo de tamaño 2*n, y copiar los n elementos del arreglo previo al nuevo arreglo
- Sin embargo, si analizamos experimentalmente la cantidad de elementos accedidos al hacer n operaciones de push consecutivas, obtenemos un gráfico como el de la Figura
- Tenemos algunas pocas operaciones muy caras, pero la mayoría son muy baratas
- Es decir, en promedio, la cantidad de accesos a arreglos por cada operación de push es constante (si hacemos n veces push)



Análisis amortizado de tiempo de ejecución

- El análisis amortizado consiste en evaluar el tiempo promedio de realizar n operaciones
  - Es muy útil para obtener cotas más precisas para casos como el anterior
- Analicemos el tiempo amortizado de hacer n operaciones de push consecutivas
- Por simplicidad, tomemos $n = 2^k$ para algún k, ya que este es el peor caso del análisis amortizado: la última operación duplica el arreglo, y deja muchos espacios sin usar
  - Luego, $logn = k$
- Asumamos que el tamaño inicial del arreglo es 2 (si es más grande no cambia el análisis asintótico), y contemos solo los accesos a arreglos (el resto son operaciones constantes)
- Cada uno de los n push hacen 1 acceso al arreglo
  - Esto suma n accesos
- Cada vez que duplicamos el tamaño del arreglo de k' a 2*k' sumamos 2*k' accesos para inicializar el nuevo arreglo
  - Esto suma $4 + 8 + 16 + \ldots + 2 * 2^k$ accesos
- Luego, n operaciones tardan $5n - 4$; son O(n)
- Si n push son O(n), el costo amortizado de cada push es O(1)

$$\boxed{\begin{aligned}
& n + \sum_{j=2}^{logn+1} 2^j \\
&= n + \sum_{j=0}^{logn+1} 2^j - 2^0 - 2^1 \\
&= n + (2^{logn+2} - 1) - 3 \\
&= n + (4n - 1) - 3 \\
&= 5n - 4 \\
&\in O(n)
\end{aligned}}$$

Análisis amortizado de tiempo de ejecución

- El análisis amortizado consiste en evaluar el tiempo promedio de realizar n operaciones
  - Es muy útil para obtener cotas más precisas para casos como el anterior
- Analicemos el tiempo amortizado de hacer n operaciones de push consecutivas
- Por simplicidad, tomemos $n = 2^k$ para algún k, ya que este es el peor caso del análisis amortizado: la última operación duplica el arreglo, y deja muchos espacio
  - Luego, $logn = k$
- Asumamos que el tamaño inicial del arreglo es 2 (si es más grande no car asintótico), y contemos solo los accesos a arreglos (el resto son operaciones constantes)
- Cada uno de los n push hacen 1 acceso al arreglo
  - Esto suma n accesos
- Cada vez que duplicamos el tamaño del arreglo de k' a 2*k' sumamos 2*k' accesos para inicializar el nuevo arreglo
  - Esto suma $4 + 8 + 16 + \ldots + 2 * 2^k$ accesos
- Luego, n operaciones tardan $5n - 4$; son O(n)
- Si n push son O(n), el costo amortizado de cada push es O(1)

Recordar que:

$$\sum_{j=0}^{n} 2^j = 2^{n+1} - 1$$

$$
\begin{aligned}
& n + \sum_{j=2}^{logn+1} 2^j \\
&= n + \sum_{j=0}^{logn+1} 2^j - 2^0 - 2^1 \\
&= n + (2^{logn+2} - 1) - 3 \\
&= n + (4n - 1) - 3 \\
&= 5n - 4 \\
&\in O(n)
\end{aligned}
$$

Análisis amortizado de tiempo de ejecución

- Un análisis similar permite mostrar que n inserciones en un heap tienen costo amortizado O(n*log n)
- Tomemos $n = 2^k$ para algún k, ya que este es el peor caso del análisis amortizado
  - Luego, $logn = k$
- Asumamos que el tamaño inicial del arreglo es 2, y contemos solo los accesos a arreglos
- Cada uno de los n insert hacen log n accesos al arreglo
  - Esto suma n*log n accesos
- Cada vez que duplicamos el tamaño del arreglo de k' a 2*k' sumamos 2*k' accesos para inicializar el nuevo arreglo
  - Esto suma $4 + 8 + 16 + \ldots + 2 * 2^k$ accesos
- La Figura demuestra que n operaciones tardan son O(n*log n)
- Si n insert son O(n*log n), el costo amortizado de cada insert es O(log n)

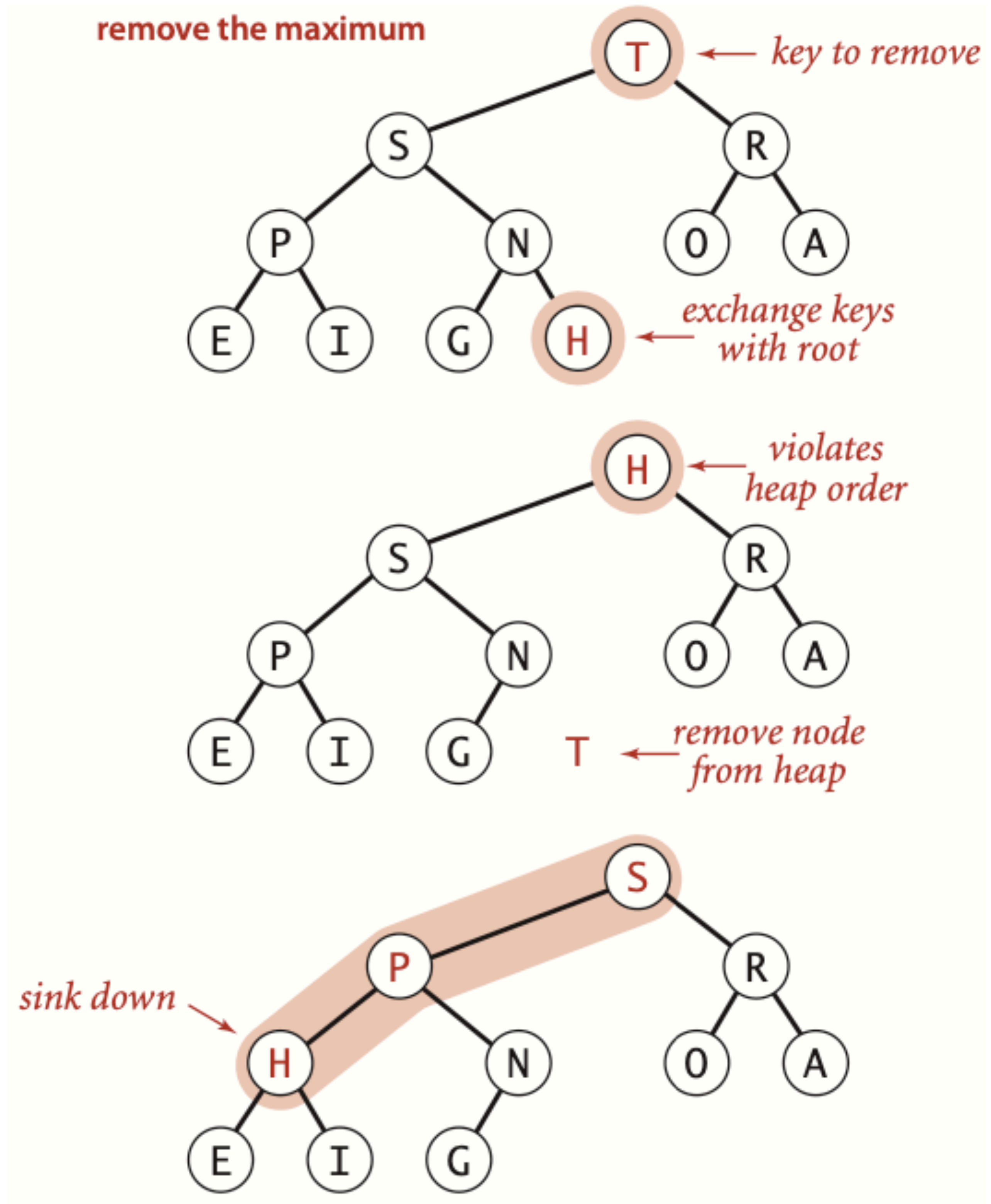
$$n * logn + \sum_{j=2}^{logn+1} 2^j$$

$$= n * logn + 4n - 4$$

Esto es: $O(n * logn)$

Eliminar el máximo

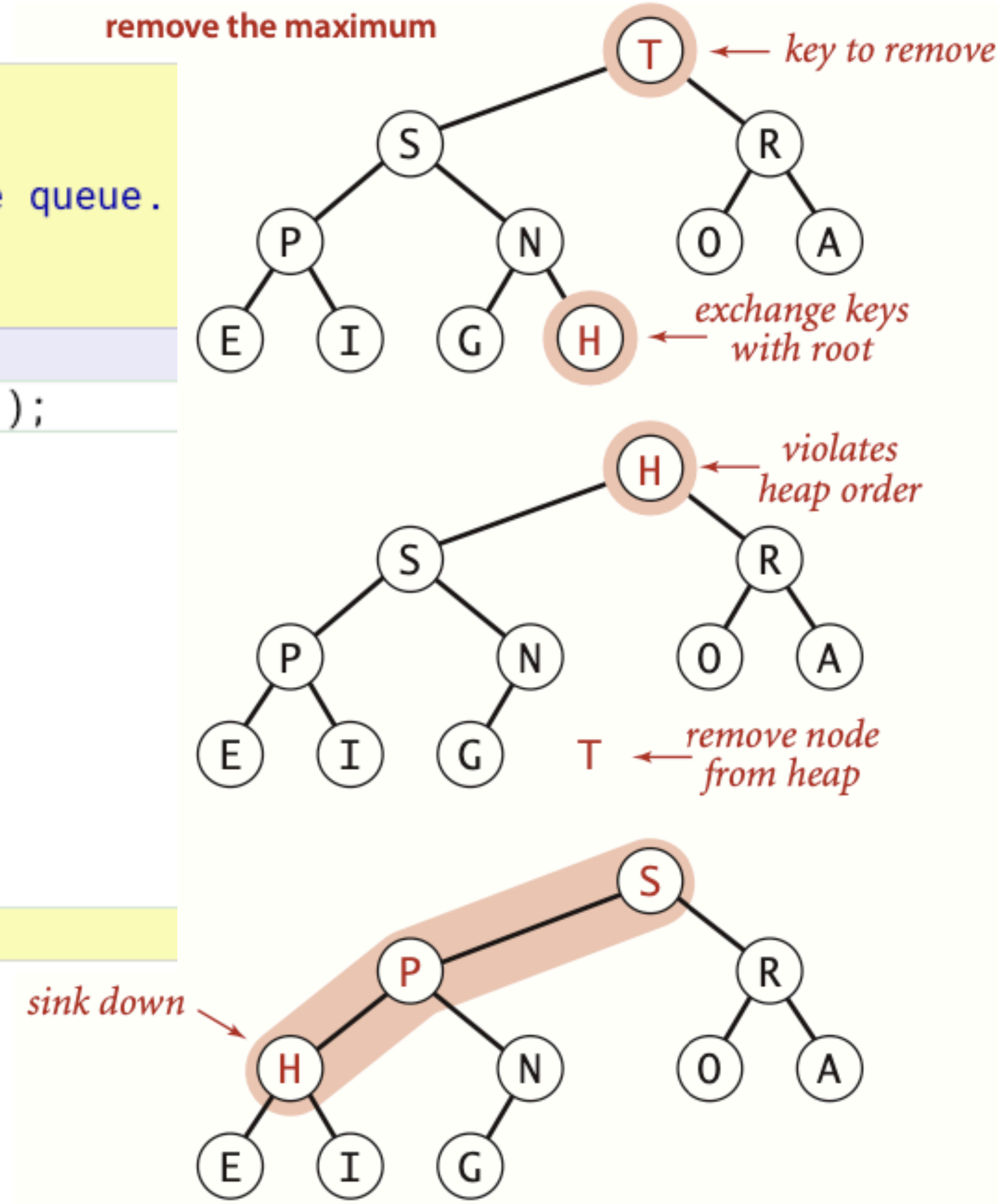
- El máximo elemento de un heap siempre está en la raíz
- Para eliminar el máximo intercambiamos la raíz con la hoja de más a la derecha, y eliminamos el último elemento
  - Esto toma tiempo O(1)
- Luego intercambiamos hacia abajo la nueva raíz hasta que encuentre su lugar en el heap usando sink
- sink es O(log n), por lo que eliminar el máximo es O(log n)



Eliminar el máximo

```java
/**
 * @pre !isEmpty()
 * @post Deletes and returns the largest element of the queue.
 */
public T removeMax() {
    if (isEmpty())
        throw new NoSuchElementException("Empty queue.");

    T max = queue[1];
    // Exchange root and last element, and
    // remove last
    exch(queue, 1, size--);
    queue[size+1] = null;
    // Sink the root to restore heap invariant
    sink(1);
    return max;
}
```



Heaps: Más operaciones

```java
/**
 * @pre !isEmpty()
 * @post Returns the largest element of the queue.
 */
public T max() {
    if (isEmpty())
        throw new NoSuchElementException("Empty queue.");
    return queue[1];
}
```

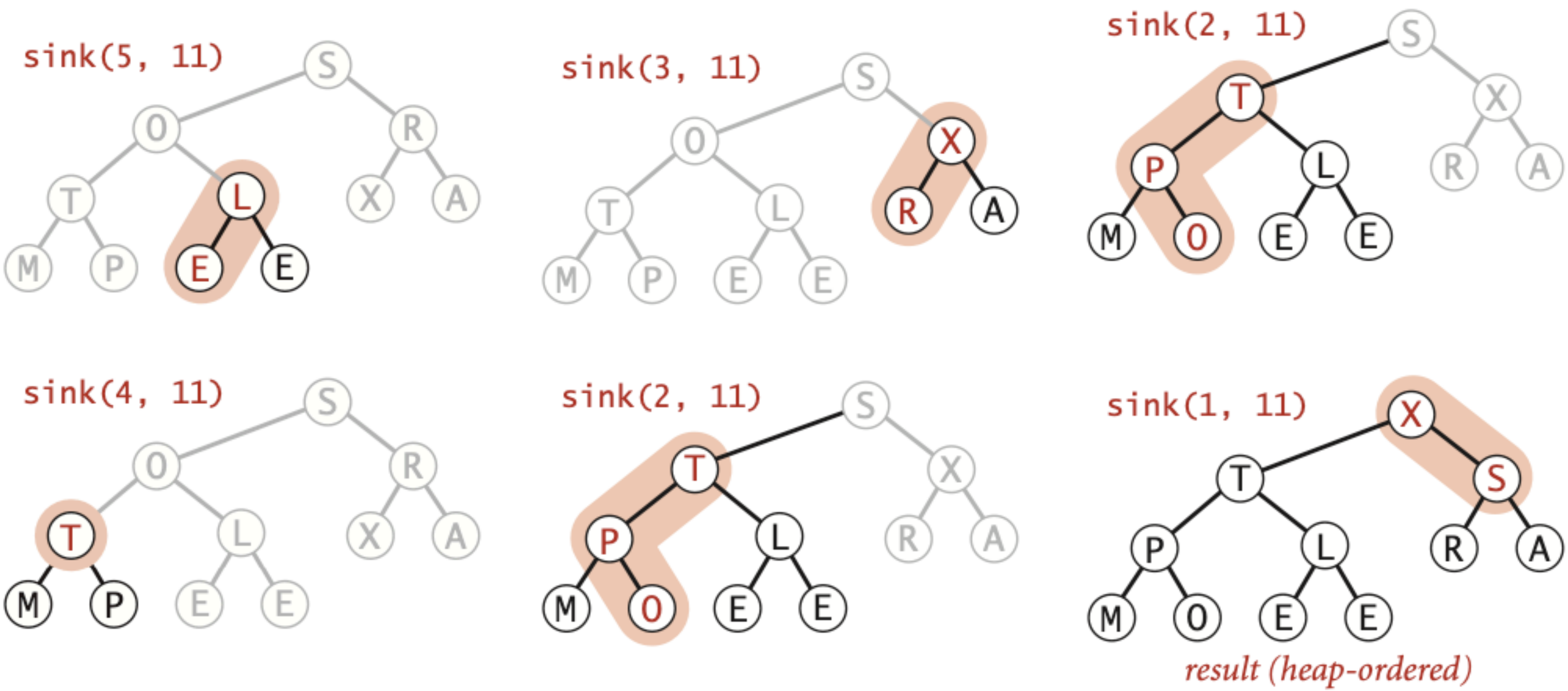
```java
/**
 * @post Returns the number of elements in the queue.
 *   More formally, it satisfies: result = #this.
 */
public int size() {
    return size;
}
```

```java
/**
 * @post Returns true iff the queue contains no elements.
 *   More formally, it satisfies: result = #this = 0.
 */
public boolean isEmpty() {
    return size == 0;
}
```

Construir un heap desde un arreglo

- Para implementar este método eficientemente, es suficiente con ir hundiendo los elementos que no son hojas a su posición correspondiente en el arreglo
- Es decir, hacemos sink(i), con i comenzando en size/2 hasta 1
- Se puede demostrar que este método es O(n)

Construir un heap desde un arreglo



Construir un heap desde un arreglo

```java
/**
 * @post Creates a priority queue with the elements
 *   of array keys.
 */
public HeapPriorityQueue(T[] keys) {
    size = keys.length;
    queue = (T[]) new Comparable[keys.length + 1];
    for (int i = 0; i < size; i++)
        queue[i+1] = keys[i];
    for (int k = size/2; k >= 1; k--)
        sink(k);
}
```

HeapSort

Podemos hacer un algoritmo de ordenación eficiente con heaps:

- Se construye un heap a partir del arreglo a ordenar
  - Vimos que esto es O(n)
- Hacemos una especie de selection sort, iterativamente sacamos el máximo del heap, y lo vamos insertando en un nuevo arreglo de atrás hacia adelante
  - Se repite n veces eliminar el máximo, que es O(log n).
  - En total, esto toma tiempo O(n*log n)
- El arreglo que se obtiene está ordenado

El tiempo de ejecución del algoritmo es O(n*log n), eficiente en la práctica
Esta versión requiere O(n) de espacio adicional

HeapSort

- Podemos evitar crear un arreglo auxiliar ordenando sobre el mismo arreglo, como se muestra en la figura
- Este código ordena las posiciones entre 1 y N del arreglo (y deja la posición 0 como viene)
- Puede ser una alternativa viable si tenemos restricciones de espacio
- Sigue siendo O(n*log n) ya que el último ciclo repite n veces sink, que es O(log n)

```java
/**
 * @post Rearranges the array elements a[1..N]
 *   in ascending order.
 */
public static void sort(Comparable[] a)
{
    int N = a.length-1;
    // Use sink on the first half of the array
    // to create a heap
    for (int k = N/2; k >= 1; k--)
        sink(a, k, N);
    // Repeatedly exchange the max with the
    // last element to get a sorted array
    while (N > 1)
    {
        exch(a, 1, N--);
        sink(a, 1, N);
    }
}
```

Una ejecución de HeapSort

N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
heap-ordered			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
sorted result			A	E	E	L	M	O	P	R	S	T	X

Función de abstracción

- En abstracto, vamos a representar las colas de prioridad como secuencias ordenadas $s = [e_1, e_2, \ldots, e_n]$, donde $e_1 \leq e_2 \leq \ldots \leq e_n$
- Primero vamos a copiar el heap en un arreglo auxiliar
- Para ordenar el arreglo auxiliar usaremos una especie de selection sort: iterativamente intercambiar la raíz (el elemento más grande) con la última posición no ordenada, y hundir la nueva raíz hacia abajo para restaurar la propiedad de heaps (y que la raíz vuelva a ser el máximo)

```java
/**
 * @post Rearranges the heap array a in ascending order.
 */
private Comparable[] sortHeapArray()
{
    Comparable[] a = Arrays.copyOfRange(queue, 0, size+1);
    int N = size;
    while (N > 1)
    {
        exch(a, 1, N--);
        sink(a, 1, N);
    }
    return a;
}
```

Función de abstracción

- Una vez que tenemos el arreglo ordenado, mostramos ese arreglo (sin incluir el primer elemento no usado del arreglo)

```
/**
 * @post Returns a string representation of the queue. Implements
 *   the abstraction function. Hence, it represents the queue as a
 *   sequence "[o1, o2,..., on]" where elements appear in ascending
 *   order (o1 <= o2 <= ... <= on).
 */
public String toString() {
    Comparable[] sorted = sortHeapArray();
    String res = "[";
    for (int i = 1; i <= size; i++)
    {
        res += sorted[i].toString();
        if (i <= size-1)
           res += ", ";
    }
    res += "]";
    return res;
}
```

Otras Operaciones sobre Heaps

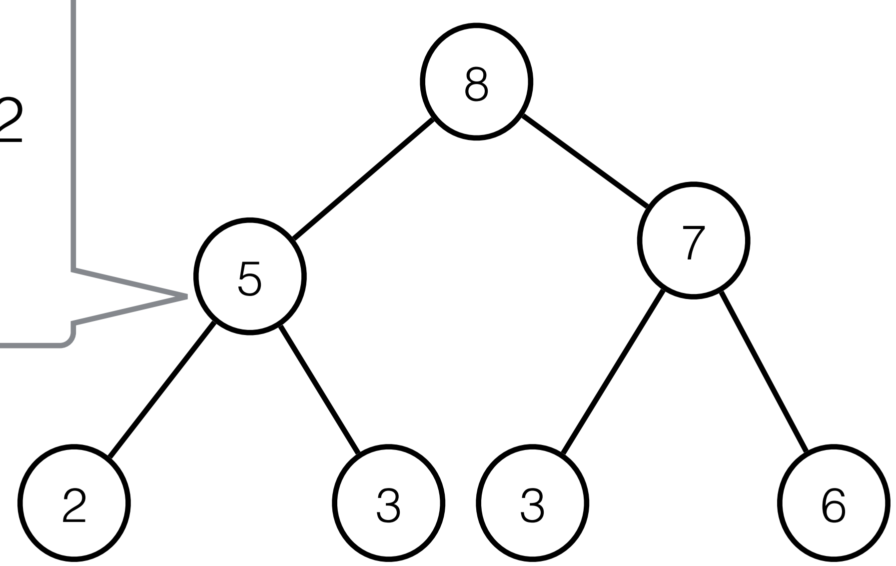
Hay otras operaciones sobre heaps que se pueden hacer eficientemente:

increaseKey(int pos, Number delta)

Se puede hacer en O(log n)

Incrementa la clave del elemento en la posición pos, en delta

Si sumamos 6 al elemento en pos = 2 (5) tenemos que subirlo con swim



Otras Operaciones sobre Heaps

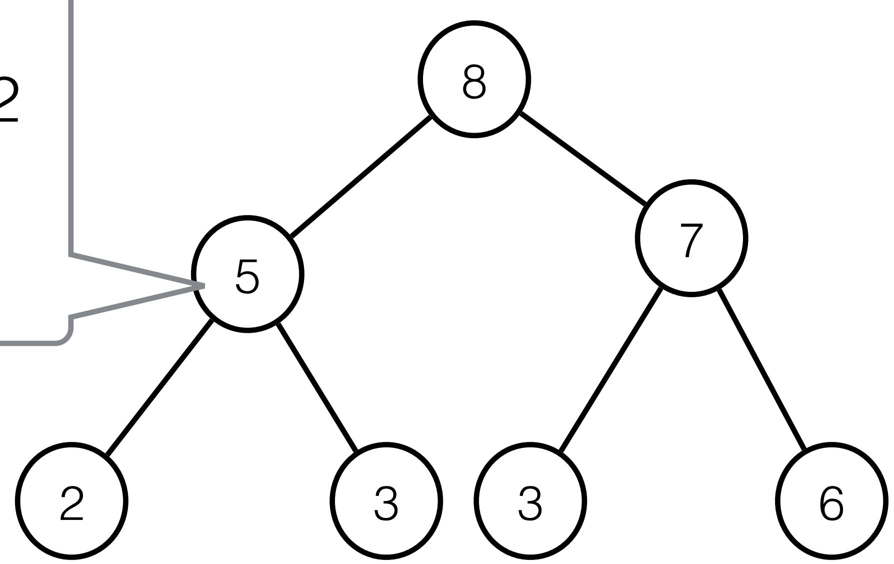
Hay otras operaciones sobre heaps que se pueden hacer eficientemente:

decreaseKey(int pos, Number delta)

Se puede hacer en O(log n)

Decrementa la clave del elemento en la posición pos, en delta

Si restamos 3 al elemento en pos = 2 (5), tenemos que bajarlo con sink



Actividades

- Leer el capítulo 6 del libro "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009
- Ó leer el capítulo 2.4 del libro "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016

Bibliografía

- "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016
- "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009
- "Data Structures and Algorithms". A. Aho, J. Hopcroft, J. Ullman. Addison-Wesley. 1983