

Estructuras de Datos

Clase 14 – Colas con prioridad



Dr. Sergio A. Gómez
<http://cs.uns.edu.ar/~sag>



Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

ADT Cola con prioridad

- Una cola con prioridad almacena una colección de elementos que soporta:
 - Inserción de elementos arbitraria
 - Eliminación de elementos en orden de prioridad (el elemento con 1era prioridad puede ser eliminado en cualquier momento)
- Nota: Una cola con prioridad almacena sus elementos de acuerdo a su prioridad relativa y no expone una noción de “posición” a sus clientes.

Prioridad: Atributo de un individuo que sirve para pesar al individuo en un conjunto de individuos.

Ejemplo: Promedio de un alumno para otorgar becas

Ejemplo: Gravedad de la situación de un paciente en una sala de espera para decidir a quién atender primero

Ejemplo: Tiempo esperado de ejecución de un proceso para decidir a qué proceso darle la CPU en un ambiente multiprogramación (la política de asignación de la CPU mediante la política de *shortest-job first* minimiza la suma de los tiempos de tiempo de espera de todos los procesos).

Comparación de prioridades con órdenes totales

- Una cola con prioridad necesita un criterio de comparación \leq que sea un *orden total* para poder resolver siempre la comparación entre prioridades.
- Sea S un conjunto y \leq una relación binaria en S , entonces (S, \leq) es un orden total ssi:
 - Reflexivo: para todo k en S , vale $k \leq k$
 - Antisimétrico: para todo k_1, k_2 en S , vale que si $k_1 \leq k_2$ y $k_2 \leq k_1$ entonces $k_1 = k_2$
 - Transitivo: para todo k_1, k_2, k_3 en S , vale que si $k_1 \leq k_2$ y $k_2 \leq k_3$ entonces $k_1 \leq k_3$
- Nota: Si (S, \leq) es un orden total, todos los pares de elementos de S son comparables entre sí mediante \leq .
- Nota: \leq en los números enteros y reales y \leq para cadenas de texto (comparación alfabética u orden lexicográfico) son órdenes totales.

Cola con prioridad

- Una cola con prioridad es un colección de elementos, llamados *valores*, los cuales tienen asociada una *prioridad* que es provista en el momento que el elemento es insertado.
- Un par prioridad-valor insertado en un cola con prioridad se llama una *entrada*.
- Operaciones fundamentales de una cola con prioridad P:
 - insert(k, x): Inserta un valor x con prioridad k en P
 - removeMin(): Retorna y remueve de P una entrada con la prioridad más pequeña

Estructuras de datos - Dr. Sergio A. Gómez

5

Entradas

Problema: ¿Cómo asociar prioridades con valores?

```
public interface Entry<K,V> {
    public K getKey();           // Retorna la prioridad de la entrada
    public V getValue();        // Retorna el valor de la entrada
}
public class Entrada<K,V> implements Entry<K,V> {
    private K clave;
    private V valor;

    public Entrada(K k, V v) { clave = k; valor = v; }
    public K getKey() { return clave; }
    public V getValue() { return valor; }
    public void setKey(K k) { clave = k; }
    public void setValue(V v) { valor = v; }
    public String toString() {
        return "(" + getKey() + "," + getValue() + ")";
    }
}
```

Estructuras de datos - Dr. Sergio A. Gómez

6

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente: "Estructuras de Datos. Notas de Clase". Sergio A. Gómez. Universidad Nacional del Sur. (c) 2013-2019.

Comparador por defecto

El comparador por defecto delega su comportamiento en el comportamiento de la operación `compareTo` del tipo básico `E`:

```
public class DefaultComparator<E extends Comparable<E>>
    implements java.util.Comparator<E> {
    public int compare( E a, E b ) {
        return a.compareTo( b );
    }
}
```

ADT Cola con Prioridad

Dada una cola con prioridad `P`:

- **size():** Retorna el número de entradas en `P`.
- **isEmpty():** Testea si `P` es vacía
- **min():** Retorna (pero no remueve) una entrada de `P` con la prioridad más pequeña; ocurre un error si `P` está vacía.
- **insert(k,x):** Inserta en `P` una entrada con prioridad `k` y valor `x`; ocurre un error si `k` es inválida (e.g. `k` es nula).
- **removeMin():** Remueve de `P` y retorna una entrada con la prioridad más pequeña; ocurre una condición de error si `P` está vacía.

Interfaz Cola con prioridad en Java

/** K representa el tipo de la prioridad del objeto de tipo V almacenado en la cola con prioridad*/

```
public interface PriorityQueue<K,V> {

    /** Retorna el número de ítems en la cola con prioridad. */
    public int size();

    /** Retorna si la cola con prioridad está vacía. */
    public boolean isEmpty();

    /** Retorna pero no elimina una entrada con minima prioridad. */
    public Entry<K,V> min() throws EmptyPriorityQueueException;

    /**Inserta un par clave-valor y retorna la entrada creada.*/
    public Entry<K,V> insert(K key, V value) throws InvalidKeyException;

    /** Remueve y retorna una entrada con minima prioridad. */
    public Entry<K,V> removeMin() throws EmptyPriorityQueueException;
}
```

Estructuras de datos - Dr. Sergio A. Gómez

11

```
public class Principal {
    public static void main(String[] args) {
        // Creo una cola con prioridad implementada con un Heap
        // con prioridades de tipo entero y valores de tipo string.
        // El constructor recibe el tamaño y el comparador de prioridades.
        PriorityQueue<Integer, String> cola = new Heap<Integer, String>( 20,
            new DefaultComparator<Integer>() );
        try {
            cola.insert(40, "Sergio"); // Inserto a Sergio con prioridad 40.
            cola.insert(30, "Martin"); // Inserto a Martín con prioridad 30.
            cola.insert(15, "Matias"); // Inserto a Matías con prioridad 15.
            cola.insert(5, "Carlos"); // Inserto a Carlos con prioridad 5.
            cola.insert(100, "Marta"); // Inserto a Marta con prioridad 100.
            // Imprimo la entrada con mínima prioridad: (5, Carlos).
            System.out.println( "Min: " + cola.min() );
            // Vacío la cola: puede lanzar EmptyPriorityQueueException
            while ( !cola.isEmpty() ){
                Entry<Integer,String> e = cola.removeMin();
                System.out.println( "Entrada: " + e);
            } // Salen las prioridades: 5, 15, 30, 40 y 100 en ese orden.
        } catch(InvalidKeyException e) { e.printStackTrace();
        } catch(EmptyPriorityQueueException e) { e.printStackTrace(); }
    }
}
```

Estructuras de datos - Dr. Sergio A. Gómez

12

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente:
 "Estructuras de Datos. Notas de Clase". Sergio A. Gómez. Universidad Nacional del Sur. (c) 2013-2019.

Implementación de cola con prioridad con listas

- Lista no ordenada:
 - ✓ insert: Se inserta al principio de la lista
 - ✓ min, removeMin: Para hallar el mínimo o removerlo es necesario recorrer toda la lista
- Lista ordenada:
 - ✓ insert: Para insertar en forma ordenada es necesario recorrer toda la lista en el peor caso
 - ✓ min, removeMin: El mínimo es el primer elemento de la lista.

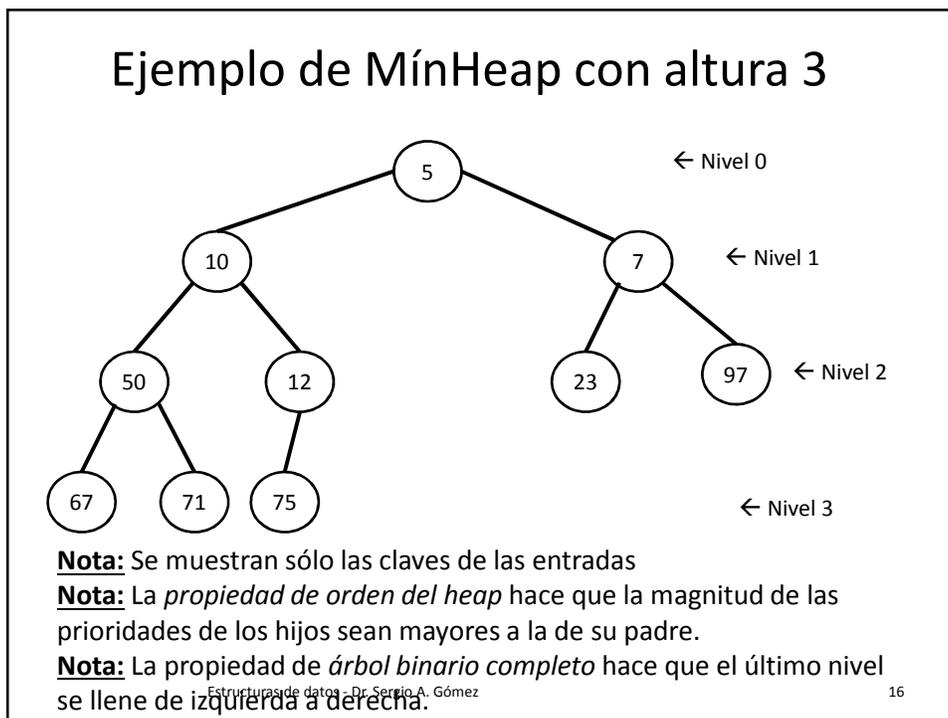
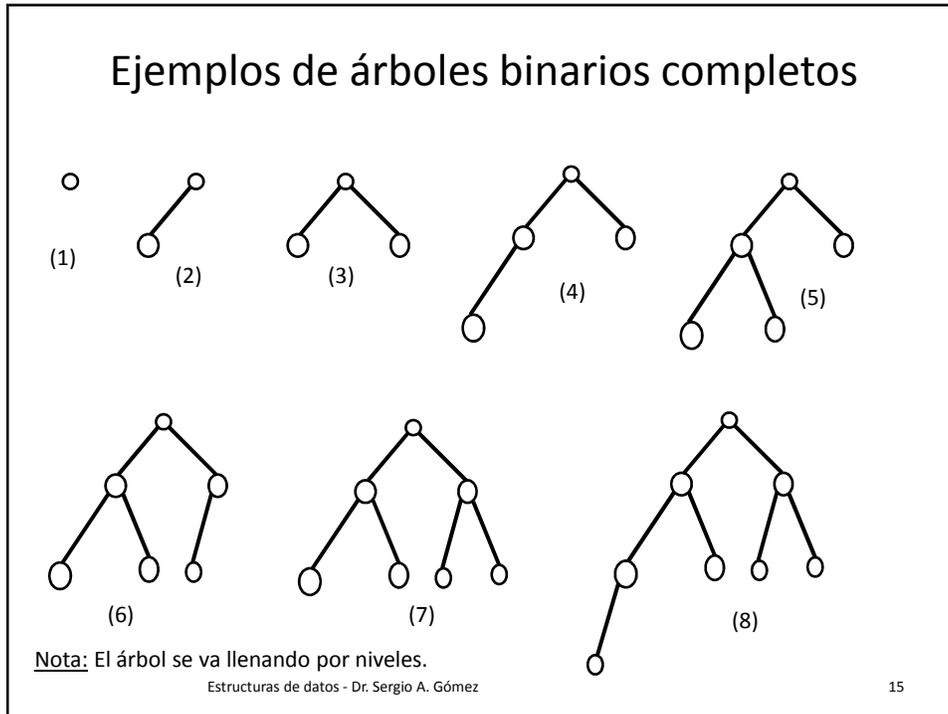
Método	Lista no ordenada	Lista ordenada
size, isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Nota: Ver fragmentos de código 8.7 y 8.8 de [GT].

Cola con prioridad implementada con Heap

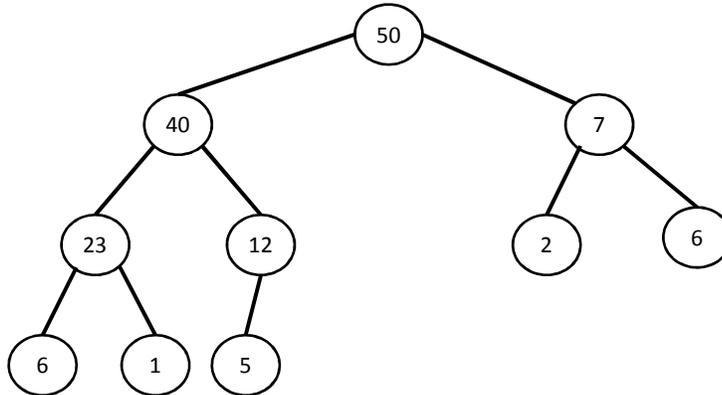
Un **(mín)heap** es un árbol binario que almacena una colección de entradas en sus nodos y satisface dos propiedades adicionales:

- **Propiedad de orden del heap (árbol parcialmente ordenado):** En un heap T , para cada nodo v distinto de la raíz, la clave almacenada en v es mayor o igual que la clave almacenada en el padre de v .
- **Propiedad de árbol binario completo:** Un heap T con altura h es un árbol binario completo si los nodos de los niveles $0, 1, 2, \dots, h-1$ tienen el máximo número de nodos posibles y en el nivel $h-1$ todos los nodos internos están a la izquierda de las hojas y si hay un nodo con un hijo, éste debe ser un hijo izquierdo (*y el nodo debiera ser el nodo interno de más a la derecha*).



El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente: "Estructuras de Datos. Notas de Clase". Sergio A. Gómez. Universidad Nacional del Sur. (c) 2013-2019.

Ejemplo de MáxHeap



Nota: Se muestran sólo las claves de las entradas.

Nota: Las magnitudes de las prioridades de los hijos son menores a las de su padre.

Nota: Para lograr esto en el código que veremos tengo que personalizar el comportamiento del comparador de prioridades.

Estructuras de datos - Dr. Sergio A. Gómez

17

Método insert(k,x)

`cola.insert(20)`



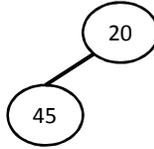
Estructuras de datos - Dr. Sergio A. Gómez

18

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente: "Estructuras de Datos. Notas de Clase". Sergio A. Gómez. Universidad Nacional del Sur. (c) 2013-2019.

Método insert(k,x)

```
cola.insert( 20 )  
cola.insert( 45 )
```

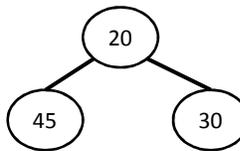


Estructuras de datos - Dr. Sergio A. Gómez

19

Método insert(k,x)

```
cola.insert( 20 )  
cola.insert( 45 )  
cola.insert( 30 )
```



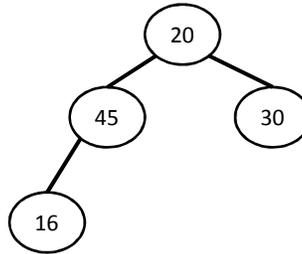
Estructuras de datos - Dr. Sergio A. Gómez

20

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente:
"Estructuras de Datos. Notas de Clase". Sergio A. Gómez. Universidad Nacional del Sur. (c) 2013-2019.

Método insert(k,x)

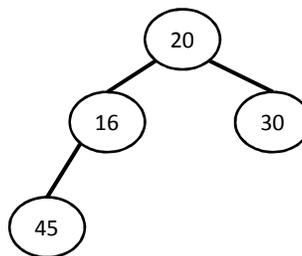
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
```



16 es menor a 45 y viola la propiedad de orden parcial => hay que intercambiarlos

Método insert(k,x)

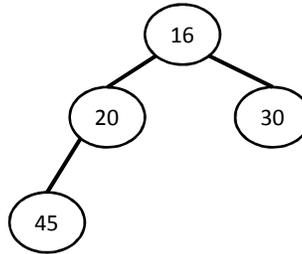
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
```



16 es menor a 20 y viola la propiedad de orden parcial => hay que intercambiarlos

Método insert(k,x)

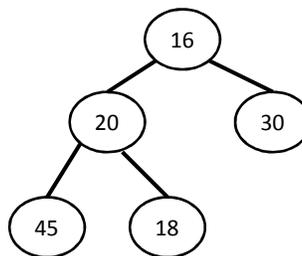
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
```



16 ya llegó a la raíz => ya terminé la inserción

Método insert(k,x)

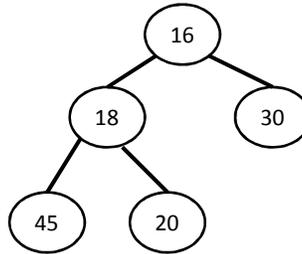
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
```



18 es menor que 20 => hay que intercambiarlos

Método insert(k,x)

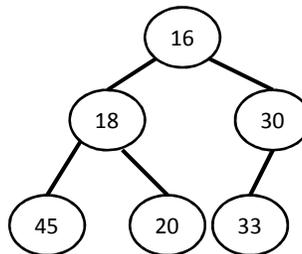
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
```



18 es mayor que 16 => terminé

Método insert(k,x)

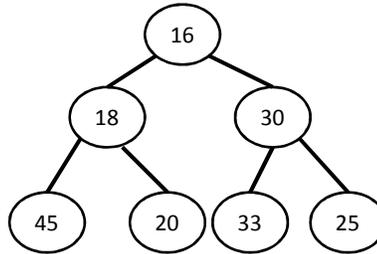
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
```



33 es mayor que 30 => terminé

Método insert(k,x)

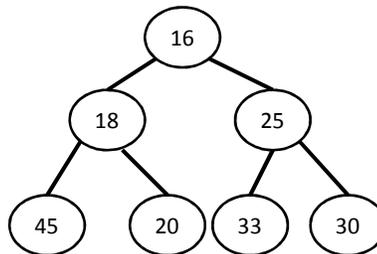
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
```



25 es menor que 30 => los intercambio

Método insert(k,x)

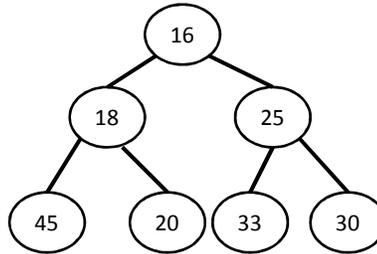
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
```



25 es mayor que 16 => terminé

Método removeMin()

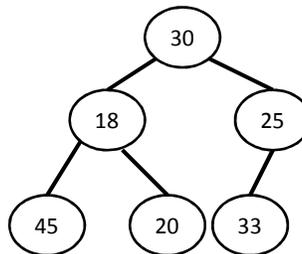
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin()
```



e será 16 porque está en la raíz
La raíz se reemplaza con la hoja más profunda y más a la derecha (es decir el último nodo del recorrido por niveles)

Método removeMin()

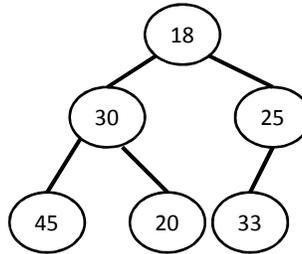
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
```



30 es mayor que 18 y que 25
Intercambio 30 por el menor de sus hijos (18)

Método removeMin()

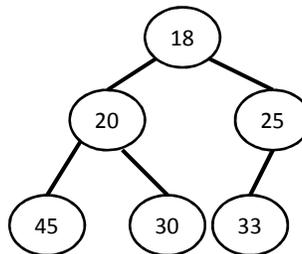
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
```



30 es mayor que 20
Intercambio 30 por el menor de sus hijos (20)

Método removeMin()

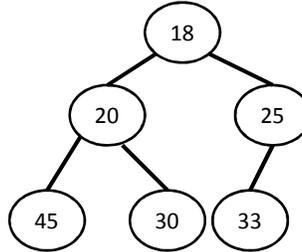
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
```



30 llegó a una hoja => terminé

Método removeMin()

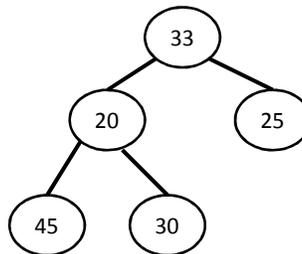
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
```



Reemplazo 18 por 33 (hoja más profunda y más a la derecha)

Método removeMin()

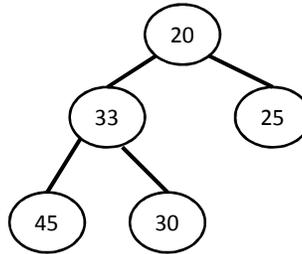
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
```



33 es mayor a 20 y 25 => intercambio 33 con el menor de sus hijos (el 20)

Método removeMin()

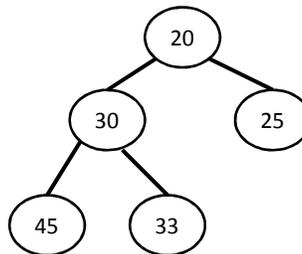
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
```



33 es mayor a 30 => intercambio 33 con el menor de sus hijos (el 30)

Método removeMin()

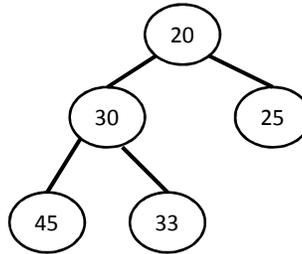
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
```



33 llegó a una hoja => terminé

Método removeMin()

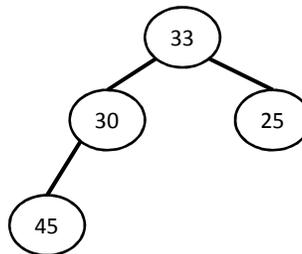
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
```



Reemplazo 20 por último nodo (el 33)

Método removeMin()

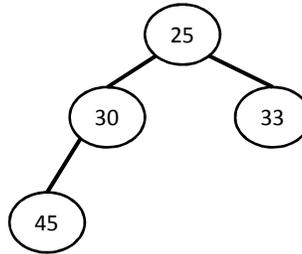
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
```



33 es mayor a 30 y a 25 => reemplazo 33 por el 25

Método removeMin()

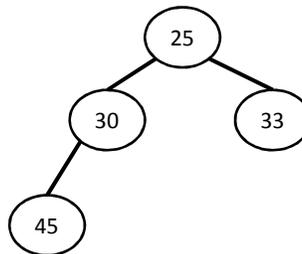
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
```



33 llegó a una hoja => terminé

Método removeMin()

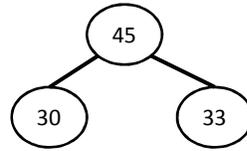
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
e ← cola.removeMin() // 25
```



Reemplazo 25 por último nodo (el 45)

Método removeMin()

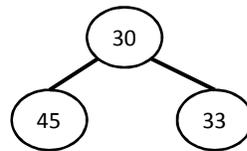
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
e ← cola.removeMin() // 25
```



45 es mayor a 30 y a 33 => Lo intercambio por el menor (el 30)

Método removeMin()

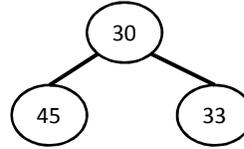
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
e ← cola.removeMin() // 25
```



45 llegó a una hoja => terminé

Método removeMin()

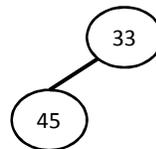
```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
e ← cola.removeMin() // 25
e ← cola.removeMin() // 30
```



Reemplazo el 30 por el último nodo del listado por niveles

Método removeMin()

```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
e ← cola.removeMin() // 25
e ← cola.removeMin() // 30
```



33 es menor a 45 => terminé

Método removeMin()

```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
e ← cola.removeMin() // 25
e ← cola.removeMin() // 30
e ← cola.removeMin() // 33
```



45

Reemplazo 33 por 45 => terminé.

Método removeMin()

```
cola.insert( 20 )
cola.insert( 45 )
cola.insert( 30 )
cola.insert( 16 )
cola.insert( 18 )
cola.insert( 33 )
cola.insert( 25 )
e ← cola.removeMin() // 16
e ← cola.removeMin() // 18
e ← cola.removeMin() // 20
e ← cola.removeMin() // 25
e ← cola.removeMin() // 30
e ← cola.removeMin() // 33
e ← cola.removeMin() // 45
```

Elimino el 45 => terminé porque el árbol quedó vacío.

Altura del heap

- **Propiedad:** Un heap T con n entradas tiene una altura $h = \lfloor \log n \rfloor$.
- **Justificación:** Como T es completo, la cantidad n de nodos mínima se da con nivel $h-1$ lleno y hay un nodo en nivel h :

$$n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h.$$

Luego, $h \leq \log_2 n$.

La cantidad n de nodos es máxima cuando el nivel h está lleno:

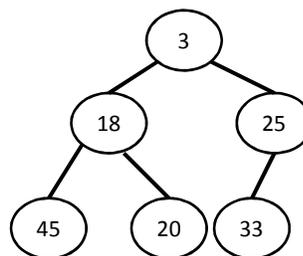
$$n \leq 1 + 2 + 4 + \dots + 2^h = 2^{h+1} - 1$$

Luego, $h \geq \log_2(n + 1) - 1$.

Por lo tanto, como h es entero, entonces $h = \lfloor \log n \rfloor$.

Representación con arreglos del árbol binario

Hijo_izquierdo(i) = $2i$
 Hijo_derecho(i) = $2i+1$
 Padre(i) = $i \text{ div } 2$



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
árbol		3	18	25	45	20	33							

Nota: La componente 0 del arreglo no se usa.

Nota: Las componentes del arreglo representan el listado por niveles del árbol.

Implementación en Java

```

public class Heap<K,V> implements PriorityQueue<K,V>{

    protected Entrada<K,V> [] elems;
    protected Comparator<K> comp;
    protected int size;

    private class Entrada<K,V> implements Entry<K,V> //Clase anidada
    {
        private K clave;    private V valor;
        public Entrada(K clave, V valor) {
            this.clave = clave;
            this.valor = valor;
        }
        public K getKey() { return clave; }
        public V getValue() { return valor; }
        public String toString() {
            return "(" + clave + ", " + valor + ")"; }
    }
}

```

Estructuras de datos - Dr. Sergio A. Gómez

49

```

public Heap(int maxElems, Comparator<K> comp ) {
    // Ojo: ¡¡Mirar bien cómo se hace la creación del arreglo!!
    // Creo un arreglo de maxElems entradas
    elems = (Entrada<K,V> []) new Entrada[maxElems];
    this.comp = comp; // Me guardo el comparador del cliente
    size = 0; // Digo que el árbol está vacío porque no tiene entradas
}

public int size() {
    return size; // Size es la cantidad de entradas del árbol
}

public boolean isEmpty() {
    return size == 0; // El árbol está vacío cuando no tiene entradas
}

```

Estructuras de datos - Dr. Sergio A. Gómez

50

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente:
 "Estructuras de Datos. Notas de Clase". Sergio A. Gómez. Universidad Nacional del Sur. (c) 2013-2019.

```

public Entry<K,V> min() throws EmptyPriorityQueueException
{
    if (isEmpty())
        throw new EmptyPriorityQueueException();
    return elems[1];
    // Recuerde que la componente 0 del arreglo no se usa
}

```

Estructuras de datos - Dr. Sergio A. Gómez

51

```

public Entry<K,V> insert(K key, V value) throws InvalidKeyException {
    Entrada<K,V> entrada = new Entrada<K,V>(key, value); // Creo una entrada nueva
    elems[++size] = entrada; // Incremento size y pongo la entrada nueva al final del arreglo

    // Burbujeo para arriba.
    int i = size; // seteo indice i de la posicion corriente en arreglo que es la última
    boolean seguir = true; // Bandera para saber cuándo encontré la ubicación de entrada
    while ( i>1 && seguir ) {
        Entrada <K,V> elemActual = elems[i]; // obtengo entrada i-ésima
        Entrada <K,V> elemPadre = elems[i/2]; // obtengo el padre de la entrada i-ésima
        if( comp.compare(elemActual.getKey(), elemPadre.getKey()) < 0) {
            Entrada<K,V> aux = elems[i]; // Intercambio entradas si están desordenadas
            elems[i] = elems[i/2];
            elems[i/2] = aux;
            i /= 2; // Reinicializo i con el índice de su padre
        } else // Si no pude intercambiar => la entrada ya estaba ordenada
            seguir = false; // Aviso que terminé
    } // fin while
    return entrada;
}

```

$T_{\text{insert}}(n) = O(h) = O(\log_2(n))$ si n es la cantidad de nodos del heap this y h su altura.

Resolución de Problemas y Algoritmos - Dr.
Sergio A. Gómez

52

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente:
 “Estructuras de Datos. Notas de Clase”. Sergio A. Gómez. Universidad Nacional del Sur. (c) 2013-2019.

```

public Entry<K,V> removeMin() throws EmptyPriorityQueueException {
    Entry<K,V> entrada = min(); // Salvo valor a retornar.
    if( size == 1 ) { elems[1] = null; size = 0; return entrada; }
    else {
        // Paso la última entrada a la raíz y la borro del final del arreglo y decremento size:
        elems[1] = elems[size]; elems[size] = null; size--;
        // Burbujeo la nueva raíz hacia abajo buscando su ubicación correcta:
        int i = 1; // i es mi ubicación corriente (Me ubico en la raíz)
        boolean seguir = true; // Bandera para saber cuándo terminar
        while ( seguir ) {
            // Calculo la posición de los hijos izquierdo y derecho de i; y veo si existen realmente:
            int hi = i*2;      int hd = i*2+1;
            boolean tieneHijolzquierdo = hi <= size();    boolean tieneHijoDerecho = hd <= size();
            if( !tieneHijolzquierdo ) seguir = false; // Si no hay hijo izquierdo, llegué a una hoja
            else {
                int m; // En m voy a computar la posición del mínimo de los hijos de i:
                if( tieneHijoDerecho ) {
                    // Calculo cuál es el menor de los hijos usando el comparador de prioridades:
                    if( comp.compare( elems[hi].getKey(), elems[hd].getKey()) < 0 ) m = hi;
                    else m = hd;
                } else m = hi; // Si hay hijo izquierdo y no hay hijo derecho, el mínimo es el izq.
            } // Fin else
        } // Fin while
    } // Fin else
}

```

53

```

// Me fijo si hay que intercambiar el actual con el menor de sus hijos:
if( comp.compare(elems[i].getKey(), elems[m].getKey()) > 0 ) {
    Entrada<K,V> aux = elems[i]; // Intercambio la entrada i con la m
    elems[i] = elems[m];
    elems[m] = aux;
    i = m; // Reinicializo i para en la siguiente iteración actualizar a partir de posición m.
} else seguir = false; // Si la comparación de entrada i con la m dio bien, termino.
} // Fin while

return entrada;
} // Fin método removeMin

```

El método tiene la complejidad del bucle while, que en el peor escenario realiza tantas iteraciones como altura h tiene árbol (el comparador funciona en orden 1 y los accesos al arreglo se realizan en orden 1; $\min()$ también tiene orden 1). Recuerde que probamos que h es del orden de logaritmo base 2 de la cantidad de nodos del árbol.

Entonces, $T_{\text{removeMin}}(n) = O(h) = O(\log_2(n))$

54

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente: "Estructuras de Datos. Notas de Clase". Sergio A. Gómez. Universidad Nacional del Sur. (c) 2013-2019.

Aplicación: Heap Sort

- Objetivo: Ordenar un arreglo A de N enteros en forma ascendente
- Estrategia: Insertar los n elementos del arreglo en un heap inicialmente vacío y luego eliminarlos de a uno y almacenarlos en el arreglo.
- Algoritmo HeapSort(a, n)


```
cola ← new ColaConPrioridad()
para i ← 0..n-1 hacer
    cola.insert( a[i] )
para i ← 0..n-1 hacer
    a[i] ← cola.removeMin()
```

Resolución de Problemas y Algoritmos - Dr. Sergio A. Gómez

55

Complejidad temporal de Heap Sort

Tamaño de la entrada:

n = cantidad de componentes de a

Algoritmo HeapSort(a, n)

cola ← new ColaConPrioridad() c_1

para i ← 0..n-1 hacer Realiza n iteraciones

cola.insert(a[i]) la iteración i cuesta $c_2 \log_2(i)$

para i ← 0..n-1 hacer Realiza n iteraciones

a[i] ← cola.removeMin() la iteración i cuesta $c_3 \log_2(n-i)$

Complejidad:

$$T_{\text{heapsort}}(n) = c_1 + c_2 n \log_2(n) + c_3 n \log_2(n) = O(n \log_2(n))$$

$\text{SPACE}_{\text{heapsort}}(n) = O(n)$ porque usa una estructura auxiliar (la heap) de tamaño n

Recordar que $\text{SPACE}_A(n)$ es la cantidad de memoria extra que usa el algoritmo A para resolver el problema.

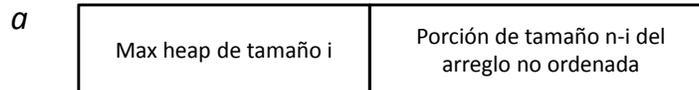
Estructuras de Datos - Dr. Sergio A. Gómez

56

El uso total o parcial de este material está permitido siempre que se haga mención explícita de su fuente: "Estructuras de Datos. Notas de Clase". Sergio A. Gómez. Universidad Nacional del Sur. (c) 2013-2019.

Heap sort in place

En lugar de usar una cola con prioridades externa (de tamaño n) al arreglo a , se puede usar una porción del mismo arreglo a para implementar la cola con prioridades y así no usar memoria adicional.



Paso 1: para $i=0$ hasta $n-1$ insertar $a[i]$ en la maxheap

Paso 2: para $i=n-1$ hasta 0 eliminar el máximo elemento de la maxheap y ubicarlo en $a[i]$.

Complejidad: $T_{\text{heapsortinplace}}(n) = O(n \log_2(n))$

$SPACE_{\text{heapsortinplace}}(n) = O(1)$ porque no usa estructuras auxiliares

Bibliografía

- Capítulo 8 de M. Goodrich & R. Tamassia, Data Structures and Algorithms in Java. Fourth Edition, John Wiley & Sons, 2006.