



UNIVERSIDAD DE ALMERÍA

Grado en Ingeniería Informática

Metodología de la Programación

2016-2017



Tema 4. Pilas, Colas y Colas de Prioridad



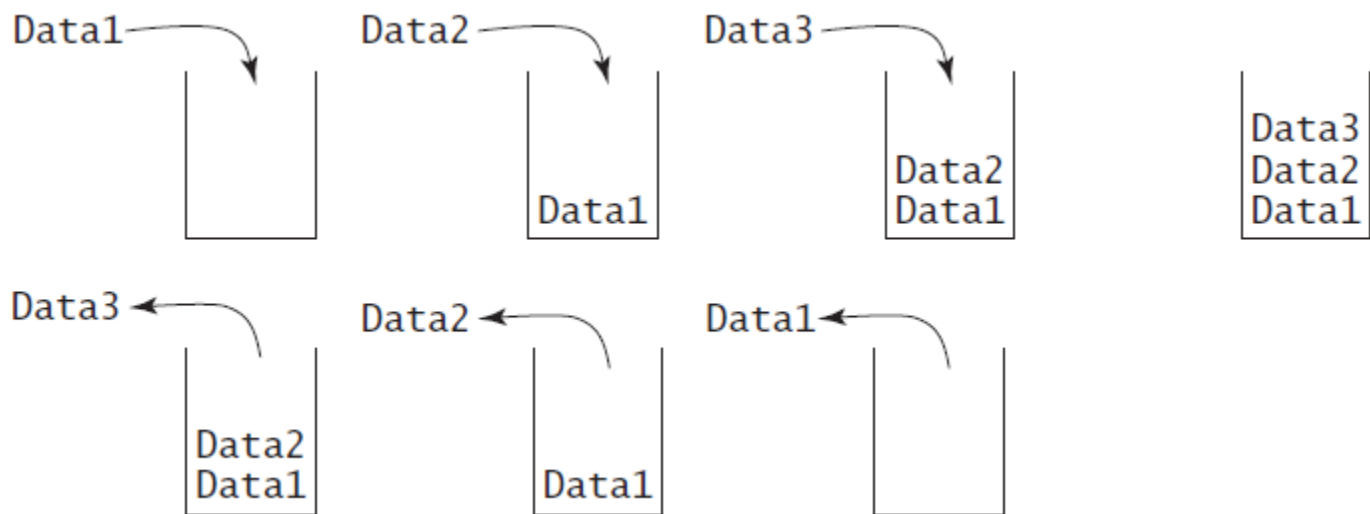
- Introducción
- Implementación de pilas usando ArrayList
- Implementación de colas usando LinkedList
- Montículos
- Implementación de colas de prioridad

Introducción

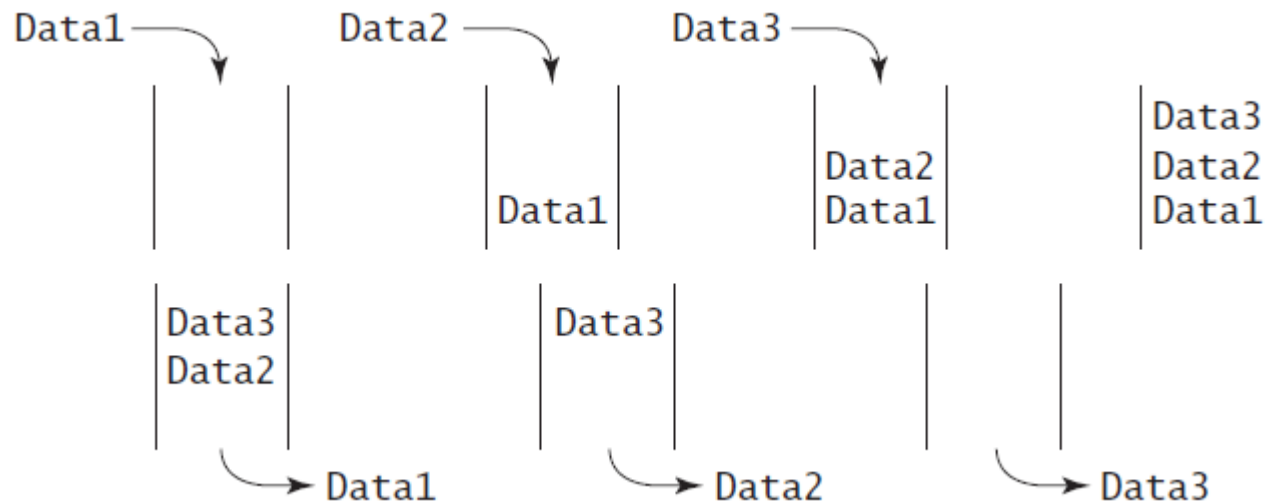
En este capítulo estudiaremos las pilas, colas y colas de prioridad. Las pilas se pueden implementar usando ArrayList y las colas usando LinkedList.

Una **pila** puede verse como un tipo especial de lista donde los elementos se insertan, eliminan y acceden solo por el final (**top**) como se muestra en la figura.

Una **cola** representa una lista de espera. Se puede ver como un tipo especial de lista donde los elementos se insertan en el extremo de la cola (**tail**) y se eliminan y acceden por el principio (**head**) como se muestra en la figura.



Una pila mantiene los datos: primero en entrar, último en salir



Una cola mantiene los datos: primero en entrar, primero en salir

Puesto que las operaciones de inserción y eliminación en una pila se hacen solo por el final, es más eficiente implementar una pila con una lista con array que con una lista con una estructura enlazada.

Puesto que las operaciones de eliminación se hacen por el principio de la lista, es más eficiente implementar una cola usando una lista enlazada que una lista con array. En este capítulo implementaremos la **pila** usando **ArrayList** y una **cola** usando **LinkedList**.

Hay dos maneras de diseñar las clases pila y cola:

- Usando **herencia**: podemos definir **la clase pila por extensión de ArrayList** y **la clase cola por extensión de LinkedList**. Se muestra en la figura. (a)
- Usando **composición**: podemos definir **un arraylist como una propiedad en la clase pila** y **un linkedlist como una propiedad en la clase cola**. Se muestra en la figura. (b)











(a) Usando herencia



(b) Usando composición

Implementación de la pila usando ArrayList

Ambos diseños está bien, aunque usar composición es mejor porque permite definir la clase pila y cola completamente nueva, sin heredar métodos innecesarios e inadecuados de ArrayList y LinkedList. La implementación de la clase **pila usando composición** se muestra a continuación proporcionando el diagrama UML y la clase **GenericStack.java**.

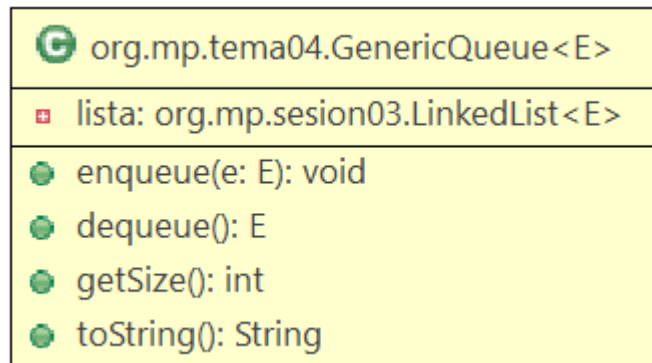
 org.mp.tema04.GenericStack<E>
 lista: org.mp.sesion03.ArrayList<E>
 getSize(): int
 peek(): E
 push(o: E): void
 pop(): E
 isEmpty(): boolean
 toString(): String

Devuelve el número de elementos de la pila.
Devuelve el elemento de la cima, no lo borra
Almacena un elemento en la cima de la pila.
Elimina un elemento de la cima y lo devuelve.
Devuelve verdadero si la pila está vacía.
toString.

Se crea una lista enlazada usando arrays (ArrayList) para almacenar elementos en la pila. Usa el método **push** para añadir un elemento a la pila y el método **pop** para eliminar un elemento de la pila. El método **getSize()** devuelve el número de elementos de la pila y el método **isEmpty** devuelve verdadero si la pila está vacía y falso en caso contrario.

Implementación de una cola usando LinkedList

La implementación de la clase **cola** usando **composición** se muestra a continuación proporcionando el diagrama UML y la clase **GenericQueue.java**.



Añade un elemento a la cola.
Elimina un elemento de la cola.
Devuelve el número de elementos de la cola.
toString.

Se crea una lista enlazada usando una estructura enlazada (LinkedList) para almacenar elementos en la cola. El método **enqueue(e)** añade el elemento **e** por la cola (tail) de la cola. El método **dequeue()** elimina un elemento por la cabeza (head) de la cola y devuelve el elemento eliminado. El método **getSize()** devuelve el número de elementos de la cola.

A continuación se de un ejemplo de creación de una cola usando **GenericQueue** y de una pila usando **GenericStack**.

```
package org.mp.tema04;

public class GenericStack<E> {
    private org.mp.sesion03.ArrayList<E>
        lista = new org.mp.sesion03.ArrayList<E>();

    public int getSize() {
        return lista.size();
    }

    public E peek() {
        return lista.get(getSize() - 1);
    }

    public void push(E o) {
        lista.add(o);
    }

    public E pop() {
        E o = lista.get(getSize() - 1);
        lista.remove(getSize() - 1);
        return o;
    }

    public boolean isEmpty() {
        return lista.isEmpty();
    }

    @Override
    public String toString() {
        return "Pila: " + lista.toString();
    }
}
```

```
package org.mp.tema04;

public class GenericQueue<E> {
    private org.mp.sesion03.LinkedList<E> lista
        = new org.mp.sesion03.LinkedList<E>();

    public void enqueue(E e) {
        lista.addLast(e);
    }

    public E dequeue() {
        return lista.removeFirst();
    }

    public int getSize() {
        return lista.size();
    }

    @Override
    public String toString() {
        return "Cola: " + lista.toString();
    }
}
```



```
package org.mp.sesion04;

public class TestStackQueue {
    public static void main(String[] args) {
        // Create a stack
        GenericStack<String> stack = new GenericStack<String>();
        // Add elements to the stack
        stack.push("Antonio"); // Push it to the stack
        System.out.println("(1) " + stack);
        stack.push("Susana"); // Push it to the the stack
        System.out.println("(2) " + stack);
        stack.push("Juan"); // Push it to the stack
        stack.push("Miguel"); // Push it to the stack
        System.out.println("(3) " + stack);
        // Remove elements from the stack
        System.out.println("(4) " + stack.pop());
        System.out.println("(5) " + stack.pop());
        System.out.println("(6) " + stack);

        // Create a queue
        GenericQueue<String> queue = new GenericQueue<String>();
        // Add elements to the queue
        queue.enqueue("Antonio"); // Add it to the queue
        System.out.println("(7) " + queue);
        queue.enqueue("Susana"); // Add it to the queue
        System.out.println("(8) " + queue);
        queue.enqueue("Juan"); // Add it to the queue
        queue.enqueue("Miguel"); // Add it to the queue
        System.out.println("(9) " + queue);
        // Remove elements from the queue
        System.out.println("(10) " + queue.dequeue());
        System.out.println("(11) " + queue.dequeue());
        System.out.println("(12) " + queue);
    }
}
```

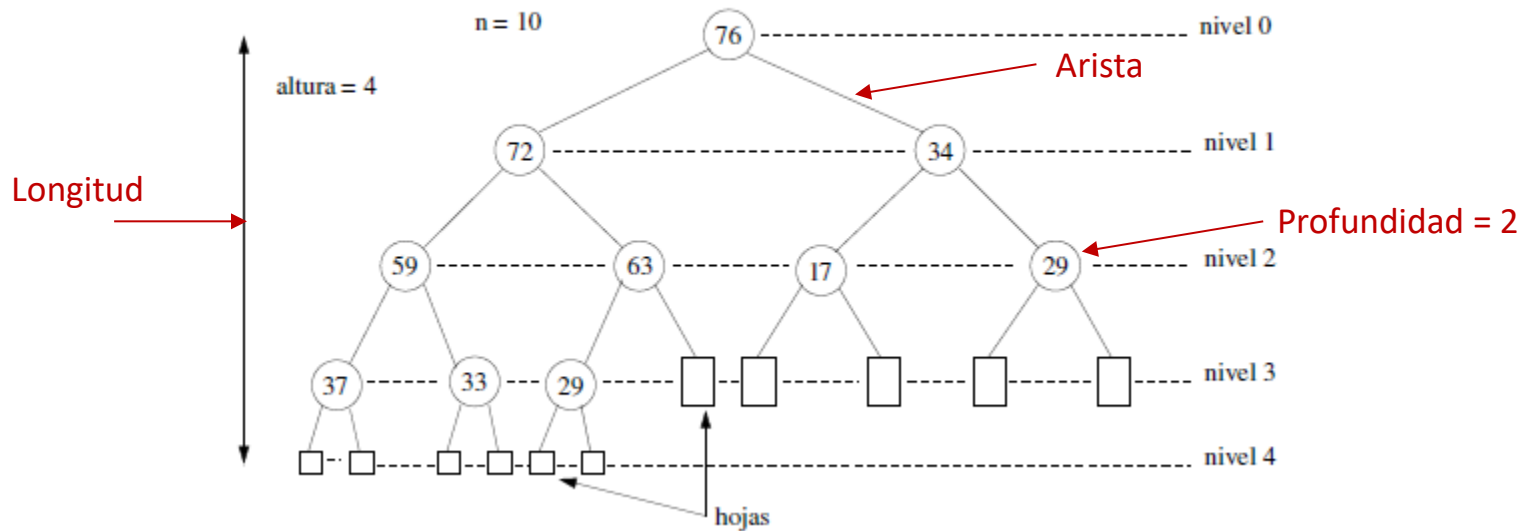


Salida

```
(1) Pila: [Antonio]
(2) Pila: [Antonio, Susana]
(3) Pila: [Antonio, Susana, Juan, Miguel]
(4) Miguel
(5) Juan
(6) Pila: [Antonio, Susana]
(7) Cola: [Antonio]
(8) Cola: [Antonio, Susana]
(9) Cola: [Antonio, Susana, Juan, Miguel]
(10) Antonio
(11) Susana
(12) Cola: [Juan, Miguel]
```

Montículo. Heap

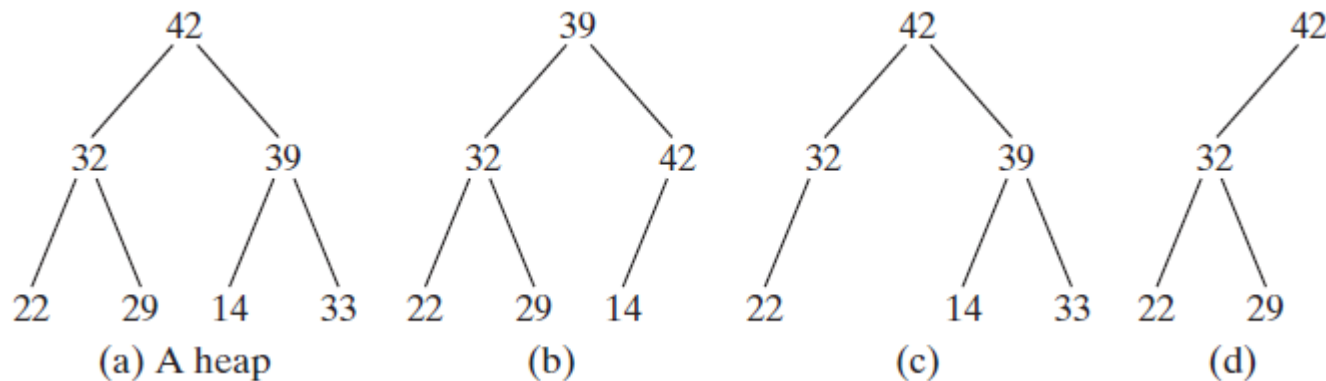
Un árbol binario es una estructura jerárquica. O bien está vacío o se compone de un elemento, llamado raíz (root) y de dos árboles binarios distintos, llamados *subárbol izquierdo* y *subárbol derecho*. La *longitud* del camino es el número de aristas en el camino. La *profundidad* de un nodo es la longitud del camino desde la raíz al nodo.



Un *montículo binario* es un árbol binario con las propiedades siguientes:

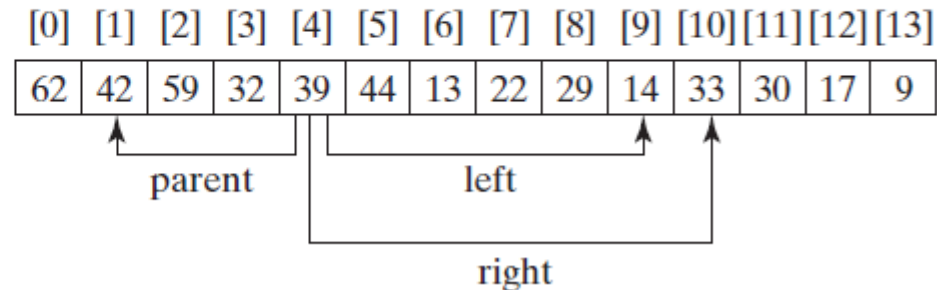
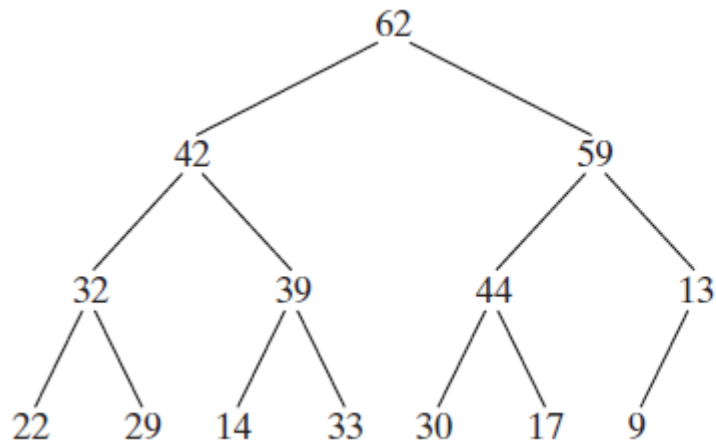
- Propiedad de la forma: Es un árbol binario completo.
- Propiedad del montículo: Cada nodo es mayor o igual que cualquiera de sus hijos.

Un árbol binario es *completo* si cada uno de sus niveles está lleno, excepto el último nivel que puede no estar lleno y todas las hojas en el último nivel se sitúan más a la izquierda. En el siguiente ejemplo los árboles en (a) y (b) están completos pero los (c) y (d) no. Además, (a) es un montículo pero (b) no lo es porque la raíz (39) es menor que su hijo derecho (42).



Un *montículo* se puede almacenar en un **ArrayList** o en un array si el tamaño del montículo se conoce de antemano. A continuación se muestra cómo se puede usar un array para almacenar un montículo. La raíz está la posición 0, y sus dos hijos están en las posiciones 1 y 2. Para un nodo en la posición i , su hijo izquierdo está en la posición $2i + 1$, su hijo derecho está en la posición $2i + 2$, y su padre está en $(i - 1)/2$.

Por ejemplo, el nodo para el elemento 39 está en la posición 4, por tanto, su hijo izquierdo (elemento 14) está en la posición 9, $2 \times 4 + 1$. Su hijo derecho está en la posición 10, $2 \times 4 + 2$. Su padre (elemento 42) está en 1, $(4 - 1)/2$.

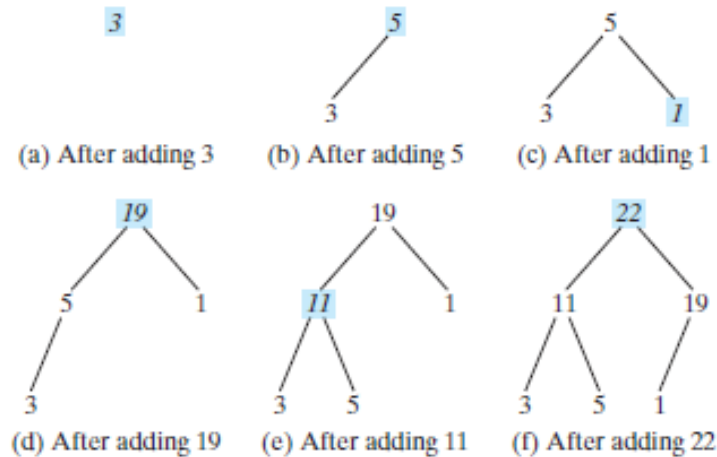


Añadir un nuevo nodo

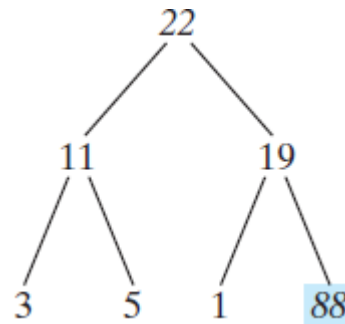
Para añadir un nuevo nodo al montículo, primero lo añadimos al final del montículo y luego reconstruimos el árbol como sigue:

```
Dejamos que el último nodo sea el actual;
while ( el nodo actual sea mayor que su padre) {
    Intercambiar el nodo actual con su padre;
    Ahora el nodo actual está en un nivel arriba;
}
```

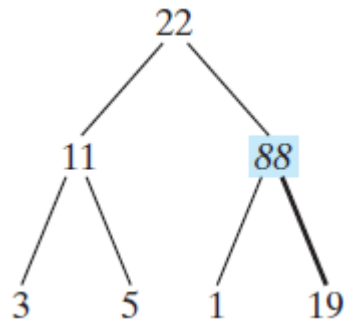
Supongamos un montículo inicialmente vacío. En la siguiente figura, se muestra cómo añadir los números 3, 5, 1, 19, 11 y 22 en ese orden.



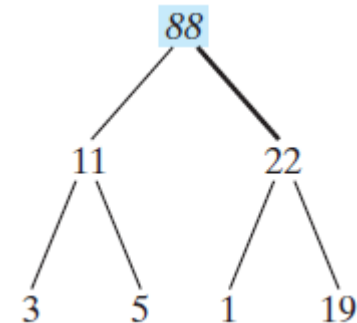
Consideramos ahora que añadimos 88 en el montículo. Situamos el nuevo nodo 88 al final del árbol, tal y como se muestra en la figura.



Intercambiamos 88 con 19



Intercambiamos 88 con 22

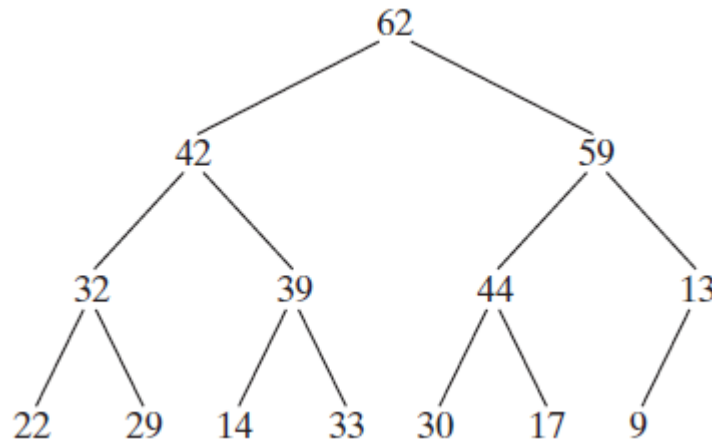


Eliminar la raíz

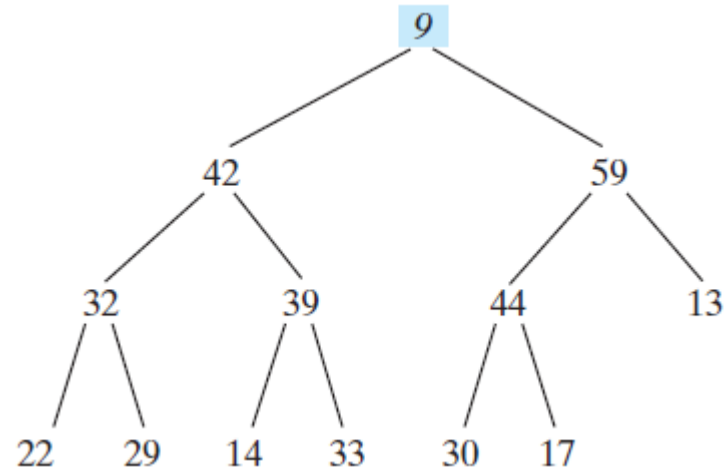
A menudo hay que eliminar la raíz de un montículo. Después de eliminado, el árbol debe reconstruirse para mantener la propiedad del montículo. El algoritmo para la reconstrucción puede describirse:

```
Mover el último nodo para sustituir la raíz;  
Dejar que la raíz sea el nodo actual  
while (el nodo actual tenga hijos y el nodo actual sea menor que  
    sus hijos) {  
    Intercambiar el nodo actual con el mayor de sus hijos;  
    Ahora el nodo actual está en un nivel abajo;  
}
```

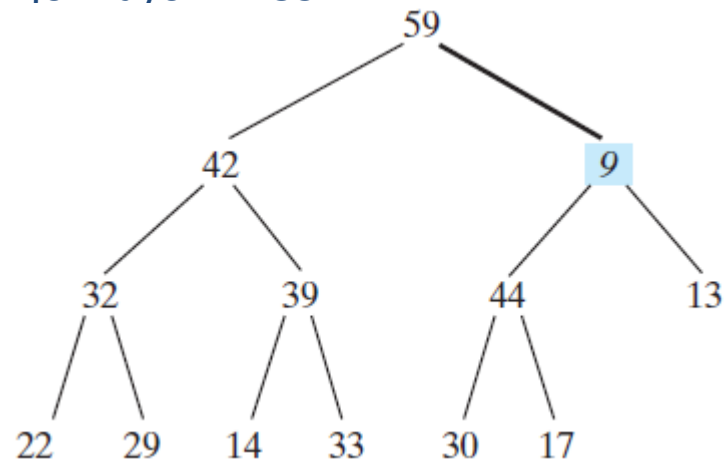
Sea:



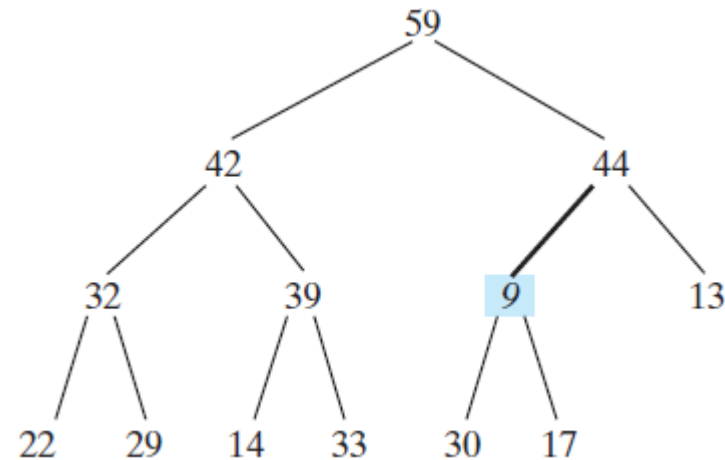
Eliminamos la raíz y sustituimos por el último nodo => movemos 9 a la raíz



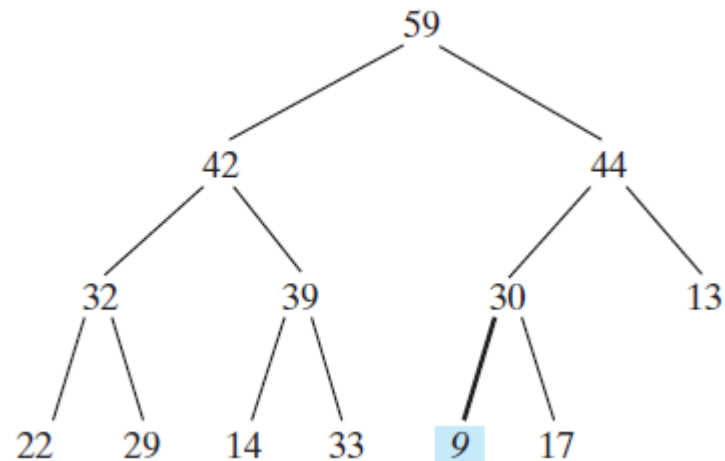
Intercambiamos 9 por su hijo mayor => 59




Después de intercambiar 9 por su hijo mayor $\Rightarrow 44$









Después de intercambiamos 9 por su hijo mayor $\Rightarrow 30$



La clase Heap

 org.mp.tema04.Heap<E extends Comparable<E>>

 lista: org.mp.sesion03.ArrayList<E>

-  Heap(): void
-  Heap(objects: E[]): void
-  add(newObject: E): void
-  remove(): E
-  getSize(): int

Crea un montículo por defecto vacío.

Crea un montículo con los objetos especificados.

Añade un nuevo objeto en el montículo.

Elimina la raíz del montículo y lo devuelve.

Devuelve el tamaño del montículo.

```
1 package org.mp.tema04;
2 public class Heap<E extends Comparable<E>> {
3     private org.mp.sesion03.ArrayList<E> lista = new org.mp.sesion03.ArrayList<E>();
4
5     /** Create a default heap */
6     public Heap() {
7     }
8
9     /** Create a heap from an array of objects */
10    public Heap(E[] objects) {
11        for (int i = 0; i < objects.length; i++)
12            add(objects[i]);
13    }
14
15    /** Add a new object into the heap */
16    public void add(E newObject) {
17        lista.add(newObject); // Append to the heap
18        int currentIndex = lista.size() - 1; // The index of the last node
19
20        while (currentIndex > 0) {
21            int parentIndex = (currentIndex - 1) / 2;
22            // Swap if the current object is greater than its parent
23            if (lista.get(currentIndex).compareTo(
24                lista.get(parentIndex)) > 0) {
25                E temp = lista.get(currentIndex);
26                lista.set(currentIndex, lista.get(parentIndex));
27                lista.set(parentIndex, temp);
28            }
29            else
30                break; // the tree is a heap now
31        }
```

```

32     currentIndex = parentIndex;
33 }
34 }
35
36 /** Remove the root from the heap */
37 public E remove() {
38     if (lista.size() == 0) return null;
39
40     E removedObject = lista.get(0);
41     lista.set(0, lista.get(lista.size() - 1));
42     lista.remove(lista.size() - 1);
43
44     int currentIndex = 0;
45     while (currentIndex < lista.size()) {
46         int leftChildIndex = 2 * currentIndex + 1;
47         int rightChildIndex = 2 * currentIndex + 2;
48
49         // Find the maximum between two children
50         if (leftChildIndex >= lista.size()) break; // The tree is a heap
51         int maxIndex = leftChildIndex;
52         if (rightChildIndex < lista.size()) {
53             if (lista.get(maxIndex).compareTo(
54                 lista.get(rightChildIndex)) < 0) {
55                 maxIndex = rightChildIndex;
56             }
57         }
58
59         // Swap if the current node is less than the maximum
60         if (lista.get(currentIndex).compareTo(
61             lista.get(maxIndex)) < 0) {
62             E temp = lista.get(maxIndex);
63             lista.set(maxIndex, lista.get(currentIndex));

```

```

64         lista.set(currentIndex, temp);
65         currentIndex = maxIndex;
66     }
67     else
68         break; // The tree is a heap
69 }
70
71     return removedObject;
72 }
73
74 /** Get the number of nodes in the tree */
75 public int getSize() {
76     return lista.size();
77 }
78 }


```


Implementación de una cola de prioridad


Una cola ordinaria es una estructura en la que el primer elemento en entrar es el primero en salir. Los elementos se añaden por el final y se eliminan por el principio. En una *cola de prioridad*, a los elementos se les asigna una prioridad. Cuando se accede a los elementos, el elemento con la prioridad más alta se elimina el primero.


Por ejemplo, en urgencias en un hospital se le asigna una prioridad al paciente, el paciente con prioridad más alta es tratado el primero.


Una cola de prioridad se pueden implementar usando un montículo en el cual la raíz es el objeto con la prioridad más alta en la cola.

 org.mp.tema04.PriorityQueue<E extends Comparable<E>>

 heap: Heap<E>

 enqueue(newObject: E): void

 dequeue(): E

 getSize(): int

Añade un elemento a la cola.

Elimina un elemento de la cola.

Devuelve el número de elementos de la cola.

```
package org.mp.tema04;

public class PriorityQueue<E extends Comparable<E>> {
    private Heap<E> heap = new Heap<E>();

    public void enqueue(E newObject) {
        heap.add(newObject);
    }

    public E dequeue() {
        return heap.remove();
    }

    public int getSize() {
        return heap.getSize();
    }
}
```



Salida

Carmen (prioridad:7) Antonio (prioridad:5) Juan (prioridad:2) Ana (prioridad:1)

```
1 package org.mp.tema04;
2 public class TestPriorityQueue {
3     public static void main(String[] args) {
4         Paciente paciente1 = new Paciente("Juan", 2);
5         Paciente paciente2 = new Paciente("Ana", 1);
6         Paciente paciente3 = new Paciente("Antonio", 5);
7         Paciente paciente4 = new Paciente("Carmen", 7);
8
9         PriorityQueue<Paciente> colaPrioridad
10            = new PriorityQueue<Paciente>();
11         colaPrioridad.enqueue(paciente1);
12         colaPrioridad.enqueue(paciente2);
13         colaPrioridad.enqueue(paciente3);
14         colaPrioridad.enqueue(paciente4);
15
16         System.out.println();
17         while (colaPrioridad.getSize() > 0)
18             System.out.print(colaPrioridad.dequeue() + " ");
19     }
20
21     static class Paciente implements Comparable<Paciente> {
22         private String nombre;
23         private int prioridad;
24
25         public Paciente(String name, int priority) {
26             this.nombre = name;
27             this.prioridad = priority;
28         }
29
30         @Override
31         public String toString() {
32             return nombre + "(prioridad:" + prioridad + ")";
33         }
34
35         public int compareTo(Paciente o) {
36             return this.prioridad - o.prioridad;
37         }
38     }
39 }
```

Resumen

- Hemos aprendido cómo implementar pilas y colas.
- Hemos aprendido cómo definir e implementar la clase montículo *heap* y cómo insertar y eliminar elementos de/al montículo.
- Hemos aprendido cómo implementar una cola de prioridad usando un montículo.



¡MUCHAS GRACIAS!



UNIVERSIDAD DE ALMERÍA

