

Capítulo 7

Heap

7.1 Introducción

Una *cola de prioridad* es una estructura de datos apropiada cuando la tarea más importante es localizar y/o eliminar el elemento con valor menor de una colección.

Ejemplo “limpiar la casa”, se puede posponer sin que sea imperativa su realización, sin embargo, “estudiar para el examen que será mañana” tiene que hacerse ya, y lo demás espera. Es decir se puede ordenar de acuerdo a su importancia o quizá también en el tiempo que requieren para terminar, su beneficio a largo plazo o bien lo divertida que son. En computación, procesos activos, la cola de impresión.

Existen varias posibilidades de implementar una cola de prioridad:

1. Utilizar una lista, así se insertan los elementos al inicio, con lo cual la inserción es rápida, luego se recorre la lista para encontrar el elemento menor, esto puede requerir $O(n)$ pasos.
2. Con una lista ordenada, aunque la localización y eliminación es rápida el problema es en la inserción de datos pues puede ser del orden de $O(n)$.
3. Con un árbol binario de búsqueda. El problema es que para encontrar el menor elemento hay que recorrer todo el lado izquierdo.

El nombre es poco afortunado pues no es una cola en el sentido FIFO.

Definiciones:

Un *árbol binario completo* es un árbol binario en el que todos los nodos a excepción de las hojas tienen dos hijos, y se llena de izquierda a derecha. De manera formal se dice que es aquel en que existe un entero positivo k tal que:

1. Toda hoja está nivel k o $k + 1$, y
2. Si un nodo en el árbol tiene un descendiente derecho en el nivel $k + 1$ entonces todos sus hijos izquierdos que son hojas están a nivel $k + 1$.

Ejemplos:

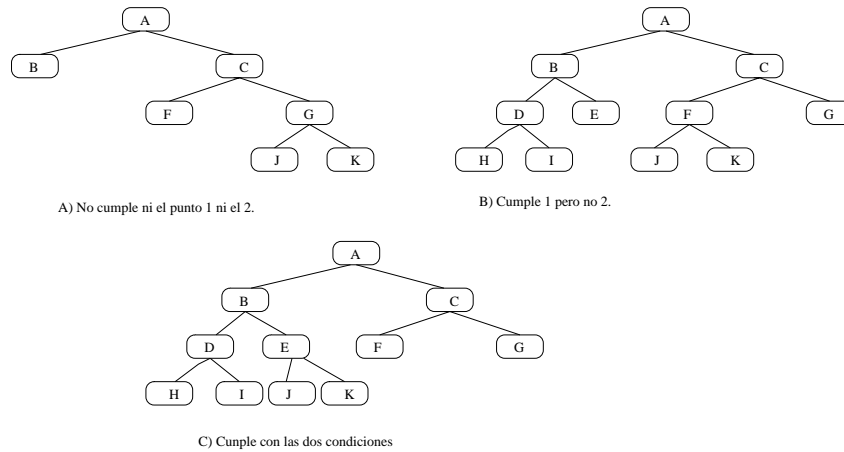


Figura 7.1: Árboles completos.

Un árbol binario completo proporciona el máximo número de hojas con la mínima trayectoria a ellas.

Una propiedad importante es que un árbol binario completo puede representarse eficientemente en un arreglo. La raíz se tiene en la localidad 0, el hijo izquierdo está en la posición $2n + 1$ y el hijo derecho está en la $2n + 2$.

Dado el índice de un nodo es fácil descubrir su padre o hijos. No debe haber huecos en el arreglo. La trayectoria más larga de la raíz a una hoja en un abc es $\lceil \log n \rceil$ (por arriba).

El arreglo correspondiente al árbol binario completo de la figura anterior es:

Un *heap* es un árbol binario completo en el cual el valor de cada nodo es menor o igual que el de sus hijos. Por tanto el nodo raíz es el menor elemento del árbol (o sub-árbol según el caso).

Un arreglo ordenado es un heap pero al revés no es cierto.

Ejemplo:

A continuación la implementación del heap:

```
import java.util.Comparator;

/**
 * Programa que crea y trabaja con un heap.
 * @version Noviembre 2004
 * @author Amparo López Gaona.
 * Deberá trabajar con un arregloDinamico o bien con un Vector.
 */
class Heap {
    private Object [] elementos;
    private Comparator prueba;
    private int nElementos = 0;

    public Heap(Comparator c) {
        prueba = c;
        elementos = new Object[20];
        nElementos = 0;
    }

    public Heap(Comparator c, Object [] datos, int tam) {
        prueba = c;
        crearHeap(datos, tam);
    }

    public boolean estáVacio() {
        return nElementos == 0;
    }

    public int tamaño() {
```

```
        return    nElementos;
    }

    public Object obtenPrimero() {
        return elementos[0];
    }

    public void imprimeHeap() {
        for (int i = 0 ; i < tamaño(); i++)
            System.out.print(elementos[i]+" ");
        System.out.println();
    }
}
```

Para insertar se debe hacer un hueco, es decir un elemento se debe mover al final del arreglo, para mantener la propiedad de ser un abc, pero puede violar la propiedad de heap si éste es menor que el padre. Una solución es colocar el nuevo elemento en la última localidad y luego moverlo repetidamente intercambiando el nodo con su padre hasta tener un heap. Este método es de orden $O(\log n)$.

```
public void agregaElemento(Object valor) {
    int posicion = nElementos, // Va agregar el dato al final
        posPadre = (nElementos - 1)/2;
    Object valorPadre = (posPadre >= 0) ? elementos[posPadre] : null;

    elementos[posicion] = valor;
    while((posicion > 0) && (prueba.compare(valor, valorPadre)) < 0) {
        elementos[posicion] = valorPadre;
        posicion = posPadre;
        posPadre = (posicion - 1)/2;
        if (posPadre >= 0)
            valorPadre = elementos[posPadre];
    }
    elementos[posicion] = valor;
    nElementos++;
}
```

En esta estructura el único elemento que se puede borrar es el primero y esto lo hace el método `eliminaPrimero`, sustituyendo la raíz por el último

elemento del árbol. Sin embargo, esto puede alterar la condición de ser un heap con lo cual se va recorriendo el árbol hacia abajo para encontrar su lugar vía el método `ajustarHeap`.

```
public void eliminaPrimero() {
    int ultimaPos = tamaño() - 1;

    if (ultimaPos != 0) // Elimina el primer elemento
        elementos[0] = elementos[ultimaPos];

    nElementos--;
    ajustarHeap(0,nElementos);
}
```

El método `ajustarHeap` se encarga de restablecer la propiedad de ser un heap, para ello se examinan los dos hijos del nodo que está en la posición indicada (`pos`), en la variable `posHijo` se tienen la posición del menor de los dos hijos. Si este elemento es mayor que el valor en cuestión, ya tenemos un heap, en otro caso el menor hijo es menor también que el valor en cuestión y por tanto se deben intercambiar ambos valores. Al hacer esto la nueva raíz es menor que ambos elementos. Ahora es necesario continuar examinando del subárbol cuya raíz es el elemento a donde se modificó el valor en el intercambio anterior. La posición del hijo menor es el nuevo valor de la variable `pos` y el ciclo continua. El proceso termina cuando el valor en cuestión encuentra su ubicación (siendo el menor de los dos hijos) o bien se llegó a un lugar sin hijos.

```
private void ajustarHeap(int pos, int tamaño) {
    Object valor = elementos[pos];
    while (pos < tamaño) {
        int posHijo = pos * 2 + 1;
        if (posHijo < tamaño) { //Encuentra ls pos del hijo menor.
            if (((posHijo + 1) < tamaño) &&
                cmp.compare(elementos[posHijo+1], elementos[posHijo]) < 0)
                posHijo++;
            if (cmp.compare(valor, elementos[posHijo]) < 0) {
                elementos[pos] = valor;
                return;
            }
        }
        pos = posHijo;
    }
}
```

```
        } else {
            elementos[pos] = elementos[posHijo];
            pos = posHijo;
        }
    } else {
        elementos[pos] = valor;
        return;
    }
}
}
```

Ahora se verá cómo crear un heap a partir de un arreglo de objetos, para ello se llama al método `ajustarHeap` con la mitad de los elementos. Se empieza desde la mitad pues de la mitad más uno todos son hojas y por definición una hoja es un heap.

```
private void crearHeap(Object [] a) {
    int max = nElementos;

    for (int i = max/2; i >=0 ; i--)
        ajustarHeap(i,max);
}
```

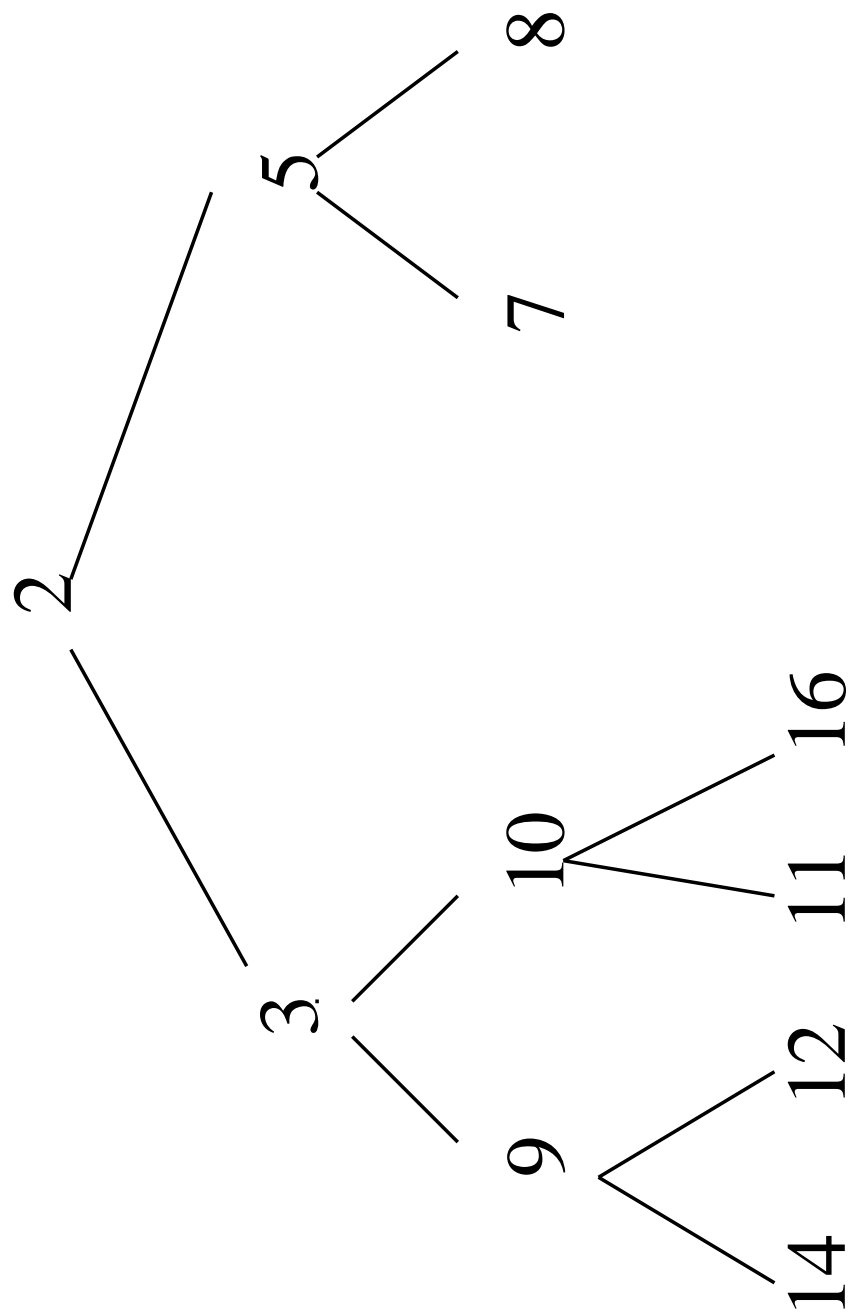
Este método privado podría ser el cuerpo de un constructor a partir de un arreglo. Faltaría también un constructor de copia.

Como caso curioso, se presenta un método para ordenar los elementos de un arreglo usando un heap. Para ello el elemento de la raíz (que es el menor del arreglo) se intercambia con el último y el tamaño del heap se reduce en 1.

```
public void ordenar(Object [] a) {
    crearHeap(a);

    for (int i = nElementos -1; i > 0; i--) {
        Object tmp = elementos[i];
        elementos[i] = elementos[0] ;
        elementos[0] = tmp ;
        ajustarHeap(0,i);
    }
}
```

G	H	I	J	K
6	7	8	9	10

**Figura 7.3:** Ejemplo de heap.