

# Soluciones - práctico 3


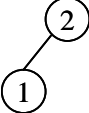
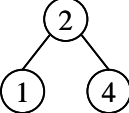
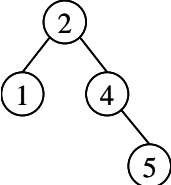
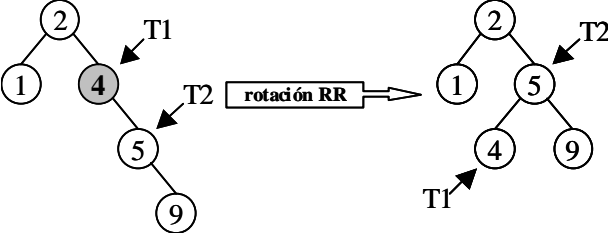
Árboles balanceados (AVL)

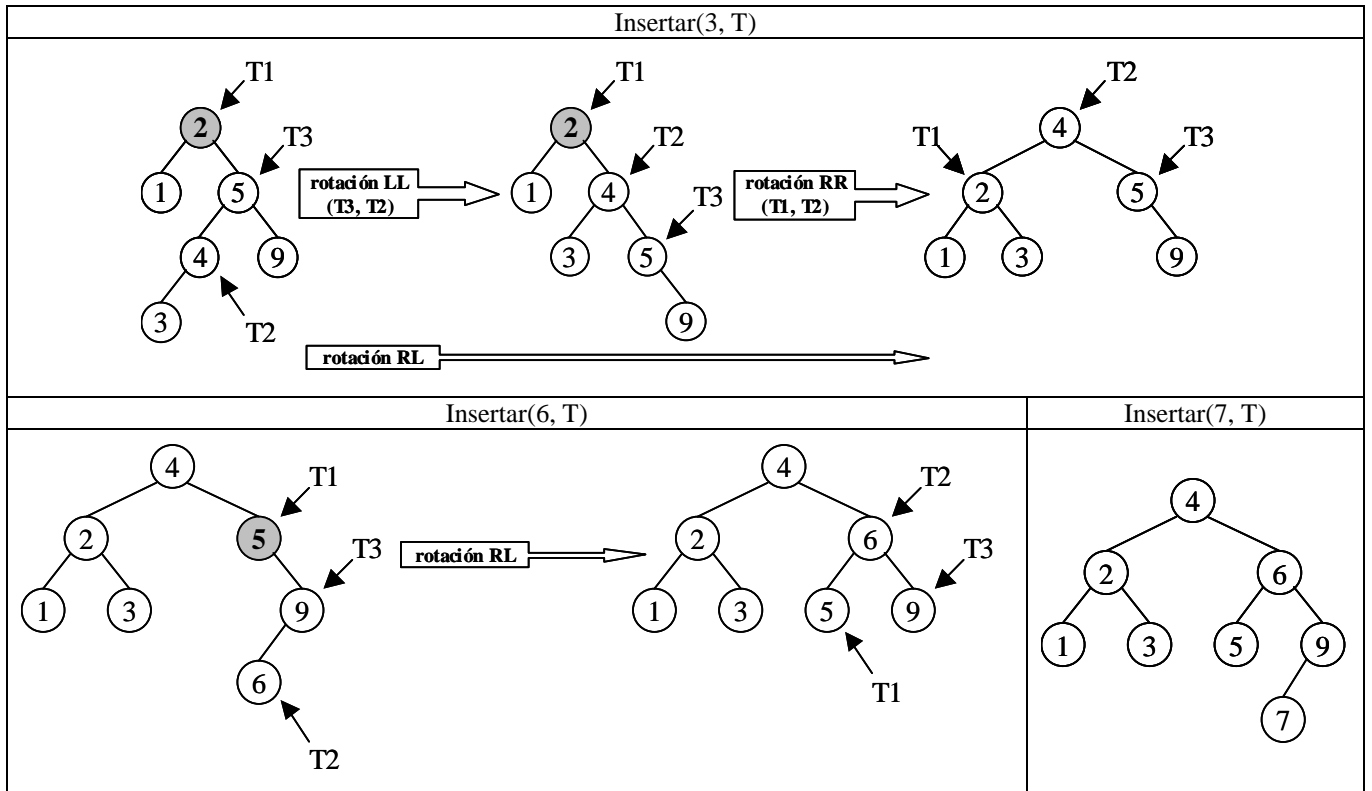
Tablas de dispersión (Hash)

Colas de prioridad (Heap)

## ÁRBOLES BALANCEADOS (AVL)

### EJERCICIO 2 (I)

Insertar(2, T)	Insertar(1, T)	Insertar(4, T)	Insertar(5, T)
			
Insertar(9, T)			
			



# Tablas de dispersión (Hash)

## EJERCICIO 5 (I)

### PARTES 1 Y 2.

```
// hashCerrado.h

#include <stdio.h>
#include <math.h>

// Tipo opaco, definido en hashCerrado.cpp
struct nodoHashCerrado;

typedef nodoHashCerrado* hashCerrado;

// CONSTRUCTORAS

void crear (hashCerrado &h, int tamanio, int resColision);
// Crea la tabla, resColision indica la manera en que es buscado el
// siguiente indice en caso de que ocurra una colision, para utilizar
// resolución lineal de colisiones, ingresar 1, y para utilizar resolución
// cuadrática de colisiones, ingresar 2.

void insertar (hashCerrado &h, int clave);
// Si la tabla esta llena salgo de la ejecucion del programa.
// Precondicion: El elemento no existe en la tabla

// SELECTORAS

bool pertenece (hashCerrado &h, int clave);
// Retorna true si la tabla contiene a clave.

// DESTRUCTORAS

void borrar (hashCerrado &h, int clave);
// Quita el elemento clave de la tabla h
// Precondicion: El elemento existe en la tabla
```

```

// hashCerrado.cpp

#include "hashCerrado.h"
#include <assert.h>

// DEFINICION DE TIPOS

struct bucket{
    int clave;
    bool borrado;
};

struct nodoHashCerrado{
    bucket** tabla;
    int tamano, cantidadElementos, resColision;
};

// FUNCIONES

int funcionDispersion (int clave, int tamano){
    int dispersion = clave % tamano;
    return dispersion;
}

// resColision = 1 para resolución lineal de colisiones.
// resColision = 2 para resolución cuadrática de colisiones.
void crear (hashCerrado &h, int tamano, int resColision){
    h = new nodoHashCerrado;

    h->tamano = tamano;
    h->cantidadElementos = 0;
    h->resColision = resColision;
    h->tabla = new bucket*[tamano];

    for (int i = 0; i < tamano; i++){
        h->tabla[i] = NULL;
    }
}

// Funcion auxiliar para obtener el proximo indice donde se
// debe ingresar la informacion en caso de que haya colision
int proximoIndice (int primerClave, int iteracion, int tamano, int
resColision){
    return (primerClave + (int) pow (iteracion, resColision)) % tamano;
}

void insertar (hashCerrado &h, int clave){
    // Evita loop infinito cuando la tabla esta llena
    int iteracion = 0;
    int indice = funcionDispersion (clave, h->tamano);
    int primerClave = indice;

    while (h->tabla[indice] != NULL && !h->tabla[indice]->borrado){
        iteracion++;
        // Error fatal, termina el programa
        assert (iteracion != h->tamano);
        indice = proximoIndice (primerClave, iteracion, h->tamano,
            h->resColision);
    }
    if (h->tabla[indice] == NULL){
        h->tabla[indice] = new bucket;
    }
}

```

```

    }

    h->tabla[indice]->clave = clave;
    h->tabla[indice]->borrado = false;
    h->cantidadElementos++;
}

bool pertenece (hashCerrado &h, int clave){
    int iteracion = 0;
    int indice = funcionDispersion (clave, h->tamano);
    int primerClave = indice;
    bool encuentre = false;

    while (!encontre && iteracion < h->tamano &&
           h->tabla[indice] != NULL){
        if (!h->tabla[indice].borrado &&
            h->tabla[indice]->clave == clave){
            encuentre = true;
        }
        else{
            indice = proximoIndice (primerClave, ++iteracion,
                                   h->tamano, h->resColision);
        }
    }

    return (encontre);
}

void borrar (hashCerrado &h, int clave){
    int iteracion = 0;
    int indice = funcionDispersion (clave, h->tamano);
    int primerClave = indice;
    bool encuentre = false;

    while (!encontre && iteracion < h->tamano &&
           h->tabla[indice] != NULL) {
        if (!h->tabla[indice].borrado &&
            h->tabla[indice]->clave == clave){
            encuentre = true;
        }
        else{
            indice = proximoIndice (primerClave, ++iteracion,
                                   h->tamano, h->resColision);
        }
    }
    if (encontre)
        h->tabla[indice]->borrado=true;
}

```

**EJERCICIO 8**

Se resuelve este ejercicio tomando una política de resolución de colisiones lineal ( $f(i) = i$ ).

El resultado luego de insertar 23, 48, 35, 4, 10 en una tabla de 5 buckets con función de dispersión  $h(i) = i \% 5$  es:

Iniciamente se cuenta con la tabla vacía:

0	1	2	3	4
Libre	Libre	Libre	Libre	Libre

Inserción 23:

Primero se calcula  $h(23) = 23 \% 5 = 3$ .

$h(23)$  es una posición libre y por tanto se almacena 23 en la casilla 3.

0	1	2	3	4
Libre	Libre	Libre	23	Libre
Libre	Libre	Libre	Ocupado	Libre

Inserción 48:

La posición  $h(48) = 3$  se encuentra ocupada.

Resolviendo las colisiones con redispersión lineal se obtiene la nueva ubicación de 48.

Obteniendo  $h_i(48) = (h(48) + f(i)) \% 5 = (h(48) + i) \% 5$ .

Se prueba con  $i = 1$  y se obtiene  $h_1(48) = 4$  que es una posición libre y por tanto se almacena 48 en la casilla 4.

0	1	2	3	4
Libre	Libre	Libre	23	48
Libre	Libre	Libre	Ocupado	Ocupado

Inserción 35:

Se calcula  $h(35) = 0$ , y como la casilla 0 se encuentra libre, se almacena 35 en dicha posición.

0	1	2	3	4
35	Libre	Libre	23	48
Ocupado	Libre	Libre	Ocupado	Ocupado

Inserción 4:

Se calcula  $h(4) = 4$ .

La casilla 4 se encuentra ocupada y por tanto se calcula  $h_i(4) = (h(4) + i) \% 5$ ;

Se prueba con  $i = 1$  y se obtiene  $h_1(4) = 0$  que también es una casilla ocupada.

Se prueba entonces con  $i = 2$  y se obtiene  $h_2(4) = 1$  que es una casilla libre y por tanto se procede a la inserción del valor 4 en la casilla 1.

0	1	2	3	4
35	4	Libre	23	48
Ocupado	Ocupado	Libre	Ocupado	Ocupado

Inserción 10:

Se calcula  $h(10) = 0$ .

La casilla 0 se encuentra ocupada y por tanto se calcula  $h_i(10) = (h(10) + i) \% 5$ ;

Se prueba con  $i = 1$  y se obtiene  $h_1(10) = 1$  que también es una casilla ocupada.

Se prueba entonces con  $i = 2$  y se obtiene  $h_2(10) = 2$  que es una casilla libre y por tanto se procede a la inserción del valor 10 en la casilla 2.

0	1	2	3	4
35	4	10	23	48
Ocupado	Ocupado	Ocupado	Ocupado	Ocupado

## Colas de prioridad (Heap)

### EJERCICIO 17

La implementación de una cola de prioridad mediante un *heap* permite encontrar el elemento mínimo en tiempo constante y borrar e insertar un elemento en  $O(\log(n))$  (donde  $n$  es la cantidad de elementos en la estructura). Explicar como modificar la estructura de *heap* y las operaciones de cola de prioridad para proveer la implementación de una cola de prioridad que tiene las siguientes características:

1. Un elemento se puede insertar en tiempo  $O(\log(n))$ .
2. El máximo y el mínimo se pueden borrar en tiempo  $O(\log(n))$ .
3. El máximo o el mínimo se pueden encontrar en tiempo constante.

Sugerencia: ordenar los elementos de modo que los nodos en los niveles pares tengan valor mayor que sus descendientes y los nodos en los niveles impares tengan valores menores que los de cualquiera de sus descendientes. Esta estructura se denomina comúnmente *min-max-heap*.

Un ejemplo de la estructura de un *minmax heap* se puede ver en la Figura 1.

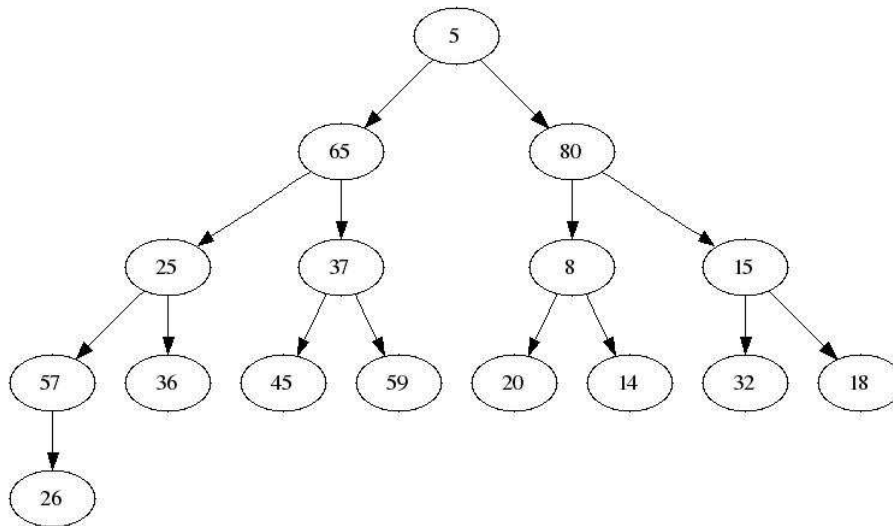


Figura 1 : minmax heap

#### 1.

La inserción en un *minmax heap* es similar a la de un *heap*. Primero se coloca el nuevo elemento como la última hoja y luego se reordena esa rama del árbol para que se cumplan las condiciones de ordenación. Sin embargo, estas condiciones son algo diferentes. Mientras que en un *heap* las relaciones de ordenación se cumplen entre un nodo y sus *hijos*, en un *minmax heap* se deben cumplir entre un nodo y sus *nietos*. Se dice que para los niveles impares un nodo tiene que ser menor que sus *nietos* y para los niveles pares tiene que ser mayor que sus *nietos*.

Para dejar el elemento insertado en la posición adecuada se debe hacer un filtrado ascendente en el *minmax heap* por los niveles pares o impares. Suponiendo que el elemento es insertado en un nivel impar (la explicación es análoga si el nivel es par), inicialmente se lo compara con su *abuelo* (si existe), en caso de que sea menor se lo intercambia y se continua el filtrado ascendente. En caso de que sea mayor (como en el ejemplo de la Figura 2) se lo compara con su *padre*, si es mayor se lo intercambia y se continua el filtrado con su *abuelo* hasta que el elemento quede en una posición donde se cumplan las condiciones del *minman heap*.



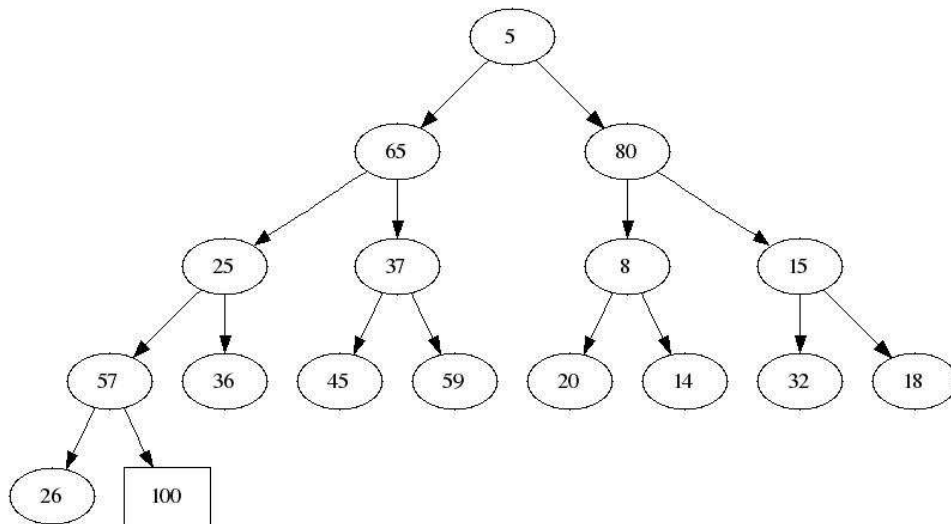


Figura 2 : insercion del 100

Siguiendo con el ejemplo, el 100 es demasiado grande para estar ahí, así que debe subir por los niveles pares. Como está originalmente en un nivel impar primero se intercambia con su *padre*, el 57 (Figura 3). Luego se va intercambiando con su *abuelo* hasta que se restablezca la ordenación de la estructura.

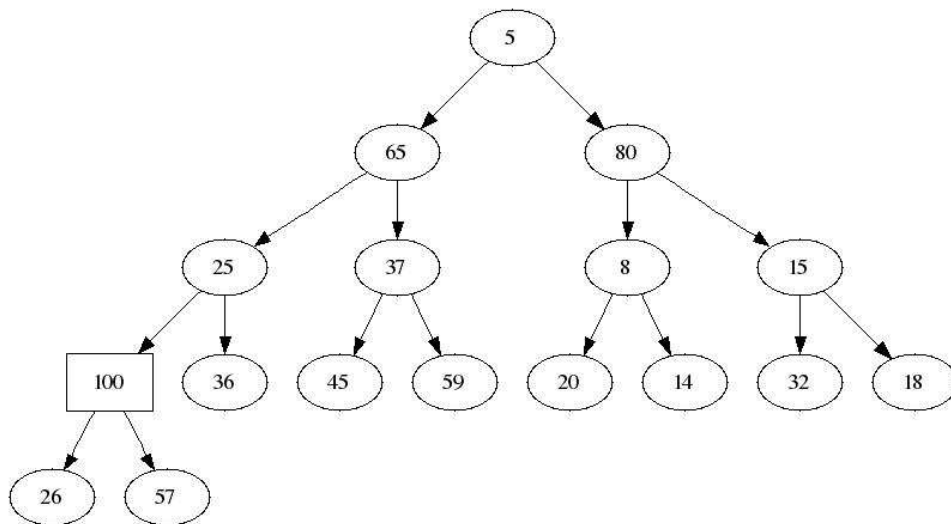


Figura 3 : insercion del 100 - primer intercambio

La Figura 4 muestra la ordenación luego de la finalizada la inserción.

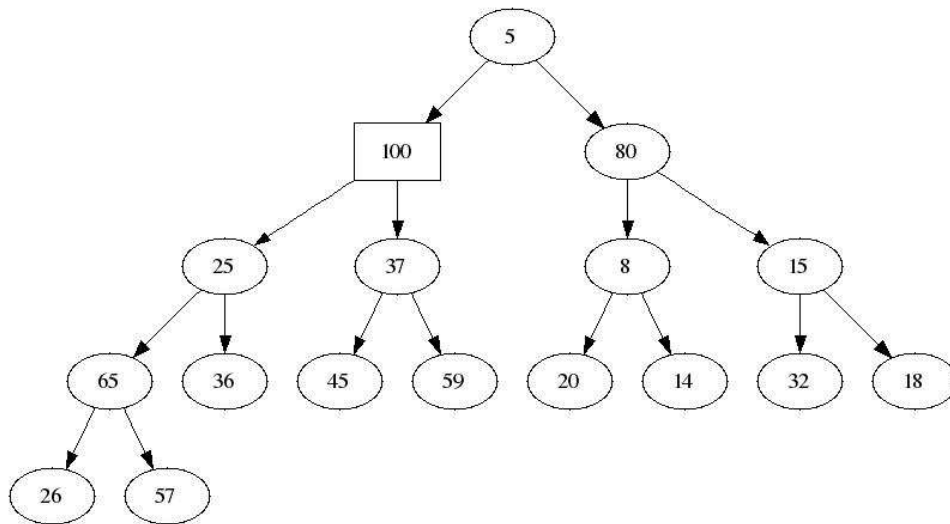


Figura 4 : insercion del 100 - minmax heap ordenado

El orden de la operación es logarítmico porque cada intercambio es  $O(1)$  y se hacen a lo sumo  $\{\text{altura del árbol}\}$  intercambios. Como el árbol es parcialmente completo la altura es  $\log(n)$ .

## 2.

Para borrar el máximo o el mínimo de la estructura se mueve la última hoja al lugar que quedó libre y luego se hace un filtrado descendente hasta restablecer el orden del árbol. Si se borró el mínimo de la estructura la última hoja se mueve a la posición de la raíz, que está en un nivel impar, así que el elemento tendría que ser menor a todos sus descendientes. Si se borró el máximo la situación es inversa. Para asegurar la condición de ordenación para el elemento hay que compararlo con los *nietos* (y con los *hijos* que son hojas).

Por ejemplo, si en el *minmax heap* inicial se elimina el máximo, la estructura queda como en la Figura 5, y se procede a re acomodar el 26.

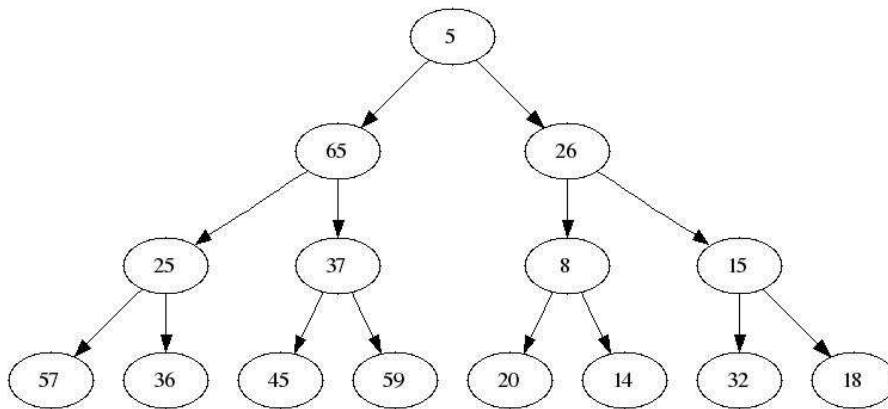


Figura 5 : eliminacion del maximo

Como el 26 está en un nivel par. La condición de ordenación no se satisface porque 26 es menor que 32. Se intercambian el 26 y el 32 y queda restablecido el orden del árbol. La estructura final se puede observar en la Figura 6.

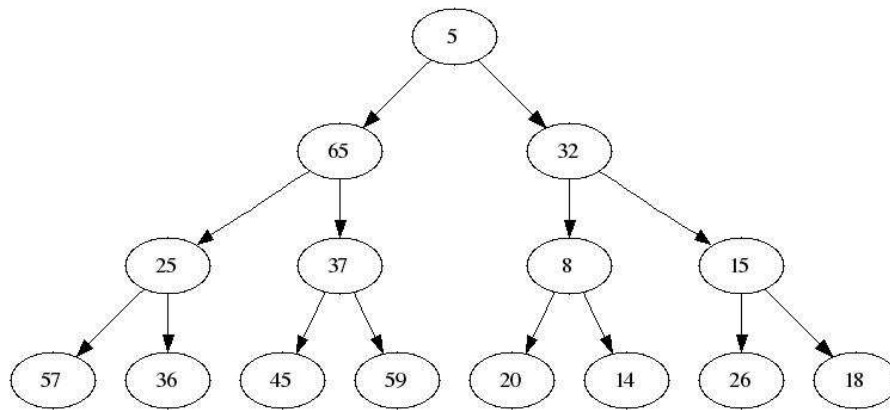


Figura 6 : minmax heap ordenado

El orden es nuevamente  $\log(n)$  porque el trabajo es una sucesión de pasos simples de  $O(1)$  a lo largo de una rama del árbol.

### 3.

Encontrar el máximo o mínimo es  $O(1)$ . El mínimo está en la raíz, y el máximo es uno de sus hijos. En el array subyacente a la estructura estos elementos son los tres primeros.