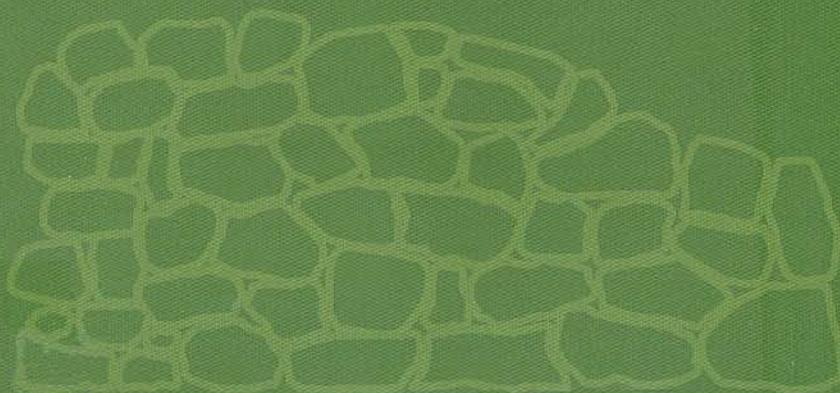


TD

INFORMÁTICA

# Ejercicios resueltos sobre *Programación* *y estructuras de datos*

Sergio Luján Mora, Antonio Ferrández Rodríguez,  
Jesús Peral Cortés y Antonio Requena Jiménez



PUBLICACIONES UNIVERSIDAD DE ALICANTE

SERGIO LUJÁN MORA, ANTONIO FERRÁNDEZ RODRÍGUEZ,  
JESÚS PERAL CORTÉS Y ANTONIO REQUENA JIMÉNEZ

EJERCICIOS RESUELTOS SOBRE  
*PROGRAMACIÓN Y ESTRUCTURAS*  
*DE DATOS*

Este libro ha sido debidamente examinado y valorado por evaluadores ajenos a la Universidad de Alicante, con el fin de garantizar la calidad científica del mismo.

Publicaciones de la Universidad de Alicante  
Campus de San Vicente s/n  
03690 San Vicente del Raspeig  
[publicaciones@ua.es](mailto:publicaciones@ua.es)  
<http://publicaciones.ua.es>  
Teléfono: 965 903 480

© Sergio Luján Mora, Antonio Ferrández Rodríguez,  
Jesús Peral Cortés y Antonio Requena Jiménez, 2014  
© de la presente edición: Universidad de Alicante

ISBN: 978-84-9717-295-0  
Depósito legal: A 123-2014

Diseño de cubiertas: candela ink  
Composición: Marten Kwickelenberg  
Impresión y encuadernación:  
Imprenta Comercial



Esta editorial es miembro de la UNE, lo que garantiza la difusión y comercialización de sus publicaciones a nivel nacional e internacional.

Reservados todos los derechos. Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra sólo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, [www.cedro.org](http://www.cedro.org)) si necesita fotocopiar o escanear algún fragmento de esta obra.

## ÍNDICE

<b>1. Introducción .....</b>	11
1.1. Introducción.....	13
1.2. Programación y Estructuras de Datos.....	14
1.3. Descripción de la asignatura.....	15
1.4. Temario de la asignatura .....	16
1.5. Estructura del libro .....	20
1.6. Cómo utilizar este libro.....	21
1.7. Agradecimientos .....	22
<b>2. Introducción a los tipos abstractos de datos .....</b>	23
2.1. Tipos abstractos de datos.....	24
2.1.1. Preguntas.....	24
2.1.2. Ejercicios.....	28
2.2. Complejidad .....	42
2.2.1. Preguntas.....	42
2.2.2. Ejercicios.....	43
<b>3. El lenguaje C++.....</b>	45
3.1. Preguntas .....	46
3.2. Ejercicios .....	48



<b>4. Tipos lineales</b>	55	6.1.1. Preguntas	112
4.1. Preguntas	56	6.1.2. Ejercicios	116
4.2. Ejercicios	57	6.2. Unión-búsqueda	117
		6.2.1. Preguntas	117
<b>5. Tipo árbol</b>	65	6.3. Cola de prioridad	119
5.1. Árbol	66	6.3.1. Preguntas	119
5.1.1. Preguntas	66	6.3.2. Ejercicios	122
5.2. Árbol binario	68		
5.2.1. Preguntas	68	<b>7. Tipo grafo</b>	137
5.2.2. Ejercicios	70	7.1. Preguntas	138
5.3. Árbol binario de búsqueda	73	7.2. Ejercicios	139
5.3.1. Preguntas	73		
5.4. Árbol enhebrado	74	<b>8. Exámenes</b>	149
5.4.1. Preguntas	74	8.1. Examen 1	151
5.5. ÁrbolAVL	75	8.1.1. Test	151
5.5.1. Preguntas	75	8.1.2. Ejercicios	153
5.5.2. Ejercicios	79	8.2. Examen 2	157
5.6. Árbol 2-3	87	8.2.1. Test	157
5.6.1. Preguntas	87	8.2.2. Ejercicios	158
5.6.2. Ejercicios	89	8.3. Examen 3	161
5.7. Árbol 2-3-4	91	8.3.1. Test	161
5.7.1. Preguntas	91	8.3.2. Ejercicios	162
5.7.2. Ejercicios	92	8.4. Examen 4	167
5.8. Árbol rojo-negro	102	8.4.1. Test	167
5.8.1. Preguntas	102	8.4.2. Ejercicios	169
5.8.2. Ejercicios	104		
5.9. ÁrbolB	107	<b>9. Solución de las preguntas de tipo test de los exámenes</b>	173
5.9.1. Preguntas	107	9.1. Examen 1	175
5.9.2. Ejercicios	109	9.2. Examen 2	176
<b>6. Tipo conjunto</b>	111	9.3. Examen 3	177
6.1. Representación de conjuntos	112	9.4. Examen 4	178

Bibliografía recomendada .....	179
Sobre los autores .....	181
Índice alfabético .....	185

1

## Introducción

En este capítulo se realiza una introducción del libro y se presenta el contenido de cada uno de los capítulos. Además, también se explican las distintas formas de utilizar este libro.

## 1.1. Introducción

El problema fundamental en el diseño e instrumentación de grandes proyectos o aplicaciones informáticas es reducir su complejidad. Los atributos que permiten obtener esta característica deseable son los siguientes: legibilidad, corrección, eficiencia, facilidad de mantenimiento y reutilización.

A través de la abstracción de datos se pueden obtener estos objetivos. Además, la abstracción de datos permitirá el cambio de representación de los datos sin afectar por ello a los programas que los utilicen. Es además particularmente importante porque simplifica la estructura de los programas que la usan debido a que presenta un interfaz de alto nivel.

El primer mecanismo de abstracción en programación fue el concepto de procedimiento. Un procedimiento hace cierta tarea o función, que será realizada en el programa principal a través de una llamada a dicho procedimiento. Para utilizarlo, el programador sólo se preocupa de qué hace y no cómo está instrumentado. Con el paso de los años el énfasis en el diseño de programas fue cambiando del diseño basado en procedimientos a la organización de los datos. Esto permitió la aparición durante la década de los ochenta de la programación orientada a objetos. En esta forma de programar, son los datos los que constituyen la jerarquía básica. Un dato puede estar ligado a otro a través de una relación de parentesco o de cualquier otro tipo, lo que da lugar a la aparición de una red semejante a la que suele formarse en las aplicaciones procedimentales con los programas.

Los tipos de datos se definen mediante entidades abstractas (modelo matemático) y operaciones que se encapsulan de forma tal que la única vía para acceder o para modificar las entidades es a través de las operaciones abstractas. A este encapsulamiento se le llama Tipo Abstracto de Dato (TAD), haciendo referencia con el calificativo 'abstracto', a la idea de que, cuando se especifica, lo único fundamental son las cualidades, es decir, el qué se puede hacer, y no cómo se hace. Informalmente, podríamos definir un tipo abstracto de datos como un ente que tiene la propiedad de instanciabilidad, más un conjunto de operaciones, las únicas aceptadas para acceder a la representación (privacidad). La especificación sirve para expresar las cualidades de un tipo abstracto de datos, es decir, qué debe producir, y no cómo lo hace. Es una descripción breve y precisa de

- Evaluar cada representación de un tipo abstracto de datos en función de su consumo de recursos (eficiencia espacial y eficiencia temporal).
- Desarrollar el hábito de trabajar en equipos de programación.

## 1.4. Temario de la asignatura

La asignatura está dividida en cinco módulos principales, que a su vez se dividen en 17 unidades que tratan aspectos generales de la programación y las estructuras de datos:

### Módulo I: Introducción a los tipos abstractos de datos. Los tipos lineales

*Unidad 0: Presentación y objetivos de la asignatura*

*Unidad 1: Introducción a los tipos abstractos de datos*

- 1.1 Introducción.
- 1.2 Especificación algebraica de los tipos abstractos de datos.
- 1.3 Implementación de tipos abstractos de datos.

*Unidad 2: Vectores*

- 2.1 Definición y especificación.
- 2.2 Representación secuencial.
- 2.3 Aplicaciones de vectores: matrices y polinomios.

*Unidad 3: Pilas*

- 3.1 Definición y especificación.
- 3.2 Representación secuencial y enlazada. Análisis de la eficiencia de las representaciones.
- 3.3 Aplicaciones de pilas: evaluación de expresiones y gestión de la llamada a procedimientos recursivos.

*Unidad 4: Colas*

- 4.1 Definición y especificación.
- 4.2 Representación secuencial y enlazada. Análisis de la eficiencia de las representaciones.

- 4.3 Aplicaciones de colas: simulación y gestión de colas de espera.

### Unidad 5: Listas

- 5.1 Listas de acceso por posición: definición y especificación.
- 5.2 Representación secuencial.
- 5.3 Representación enlazada: listas enlazadas circulares, listas doblemente enlazadas y listas multienlazadas.
- 5.4 Análisis de la eficiencia de las representaciones.
- 5.5 Aplicaciones de listas: matrices dispersas y representación de polinomios.

### Módulo II: La eficiencia de los algoritmos

*Unidad 6: Definiciones generales de nociones de complejidad algorítmica*

- 6.1 Noción de complejidad.
- 6.2 Cotas de complejidad.
- 6.3 Notación asintótica.
- 6.4 Obtención de cotas de complejidad. Ejemplos (algoritmos de búsqueda/ordenación en un vector, etc.).

### Módulo III: Tipo árbol

*Unidad 7: Definiciones generales de árboles*

*Unidad 8: Árboles binarios*

- 8.1 Definición y especificación.
- 8.2 Representación secuencial y enlazada. Análisis de la eficiencia de las representaciones.
- 8.3 Recorridos de un árbol binario: preorden, inorden, postorden y niveles.
- 8.4 Operaciones complementarias: profundidad, número de elementos y altura de un nodo.
- 8.5 Aplicaciones de árboles binarios: árboles para la evaluación de expresiones aritméticas y códigos de Huffman.

los datos y funciones de un módulo. La especificación nos va a permitir, por tanto separar la realización del uso. Es el enlace entre el usuario y el implementador.

La especificación formal de los Tipos Abstractos de Datos la vamos a plantear desde el punto de vista algebraico. En esta aproximación, la especificación de un TAD consta de dos partes: *sintaxis*, en la que se determinan los conjuntos de objetos que se van a utilizar y el perfil de las operaciones, y la *semántica*, es decir, un conjunto suficientemente completo de axiomas que especifique cómo interactúan las funciones. Además, es necesario un vocabulario que nos servirá para declarar los objetos que aparecen en la definición de la sintaxis y la semántica. Se denomina algebraica a esta aproximación por fundamentarse en la teoría de álgebras heterogéneas. Una vez obtenida la especificación del TAD, y la herramienta para poder instrumentarla, el programador decide qué estructura de datos utilizará para representar el tipo, y qué algoritmos utilizará para realizar las operaciones determinadas por la especificación.

## 1.2. Programación y Estructuras de Datos

Desde el curso 2011-2012, se imparte la asignatura “Programación y Estructuras de Datos” en la Universidad de Alicante. Esta asignatura obligatoria pertenece al plan de estudios de la titulación de Grado en Ingeniería Informática de la Universidad de Alicante y es impartida por el Departamento de Lenguajes y Sistemas Informáticos, adscrito a la Escuela Politécnica Superior.

La asignatura “Programación y Estructuras de Datos” equivale a la asignatura “Programación y Estructuras de Datos” del antiguo plan de estudios de 2001 de las titulaciones de Informática de la Universidad de Alicante.

Este libro pretende ser un material de apoyo para los alumnos de “Programación y Estructuras de Datos” para realizar preguntas tipo test y ejercicios del temario de la asignatura que servirán de preparación a las pruebas de evaluación de conocimientos a las que se tendrán que enfrentar al final de la asignatura.

## 1.3. Descripción de la asignatura

La asignatura “Programación y Estructuras de Datos” es obligatoria y de duración semestral, con una carga docente de 6 créditos ECTS (*European Credit Transfer System*), repartidos entre 3 créditos de teoría y 3 de prácticas. Se imparte en el segundo curso del Grado en Ingeniería Informática.

La asignatura “Programación y Estructuras de Datos” tiene una dependencia con la asignatura “Programación 2” y se debe coordinar con las asignaturas “Programación 3” y “Análisis y Diseño de Algoritmos”.

Los objetivos generales de la asignatura “Programación y Estructuras de Datos” son los siguientes:

- Que el alumno conozca:
  - Los mecanismos de abstracción y su importancia para la resolución de problemas.
  - Los tipos de datos más usuales en programación, sus representaciones más comunes y su utilidad.
- Que el alumno comprenda:
  - La necesidad de separación entre los niveles de especificación, implementación y uso.
  - La necesidad de adaptar la representación interna del tipo abstracto de datos (TAD) a los requerimientos de la aplicación a resolver.
- Que el alumno sea capaz de:
  - Distinguir entre las representaciones alternativas de una abstracción de datos y razonar sobre la solución escogida en cuanto a coste computacional se refiere.
  - Aplicar los tipos abstractos de datos básicos aprendidos a problemas prácticos reales.
  - Organizar un determinado volumen de datos de la forma más racional posible en función de los requerimientos del problema a resolver.
  - Crear nuevos tipos abstractos de datos, o que pueda elegir otra representación de los mismos para adaptarlos a una aplicación determinada.

*Unidad 9: Árboles n-arios y árboles generales*

- 9.1** Definición y especificación.
- 9.2** Representación secuencial y enlazada. Análisis de la eficiencia de las representaciones.
- 9.3** Recorridos.
- 9.4** Aplicaciones: árboles genealógicos.

*Unidad 10: Árboles binarios de búsqueda*

- 10.1** Definición y especificación.
- 10.2** Operaciones básicas: inserción, búsqueda y eliminación. Análisis de la eficiencia de las operaciones.
- 10.3** Árbol binario de búsqueda equilibrado en altura (AVL). Algoritmos de inserción y eliminación. Análisis de su eficiencia.
- 10.4** Aplicaciones.

*Unidad 11: Árboles n-arios de búsqueda*

- 11.1** Definición y especificación.
- 11.2** Árboles n-arios de búsqueda: árboles 2-3 y árboles 2-3-4.
- 11.3** Aplicación: utilización en la organización interna de bases de datos.

**Módulo IV: Tipo conjunto***Unidad 12: El conjunto*

- 12.1** Definición y especificación.
- 12.2** Representación: secuencial y enlazada basada en listas. Análisis de la eficiencia de las representaciones.
- 12.3** Clasificación de los conjuntos según las operaciones.

*Unidad 13: El diccionario*

- 13.1** Definición y especificación.

- 13.2** Representación mediante tablas de dispersión (hashing). Dispersión cerrada y abierta; análisis comparativo entre ambas. Operaciones del diccionario con métodos de dispersión.
- 13.3** Otras representaciones: árbol 2-3 y árbol 2-3-4.

*Unidad 14: Colas de prioridad*

- 14.1** Definición y especificación.
- 14.2** Definición de árbol parcialmente ordenado. Representaciones: listas ordenadas y árboles parcialmente ordenados.
- 14.3** El montículo (heap) para representar árboles parcialmente ordenados. Tipos de montículos.
- 14.4** Algoritmos de inserción y borrado del elemento de prioridad mínima (máxima). Análisis de su eficiencia.

*Unidad 15: Otros tipos de conjuntos*

- 15.1** El trie. Definición e implementación. Algoritmos de búsqueda, inserción y borrado. Análisis de su eficiencia.

**Módulo V: Tipo grafo***Unidad 16: Grafos*

- 16.1** Definición de grafo y terminología. Especificación.
- 16.2** Algunas representaciones: matriz de adyacencia y lista de adyacencia. Análisis de la eficiencia de las representaciones.
- 16.3** Grafos dirigidos. Grafos acíclicos dirigidos. Grafos no dirigidos.
- 16.4** Recorridos en un grafo: profundidad y anchura. Comparación de los métodos.

*Unidad 17: Aplicaciones con grafos*

- 17.1** Clasificación de arcos. Bosque extendido en profundidad y en anchura de un grafo.
- 17.2** Coloreado de grafos y ordenación topológica.

## 1.5. Estructura del libro

Los capítulos principales del libro (del 2 al 7) se corresponden con los módulos del temario de la asignatura que se ha presentado en la sección anterior. Además, hay un capítulo adicional dedicado al lenguaje de programación C++, que se emplea en la realización de las prácticas de la asignatura. En estos seis capítulos, las preguntas están organizadas según el tema que tratan.

En el Capítulo 2 (**Introducción a los tipos abstractos de datos**) se incluyen preguntas y ejercicios sobre los tipos abstractos de datos y la complejidad.

En el Capítulo 3 (**El lenguaje C++**) se incluyen preguntas y ejercicios sobre el lenguaje de programación C++.

En el Capítulo 4 (**Tipos lineales**) se incluyen preguntas y ejercicios sobre los tipos lineales más importantes: la pila, la cola y la lista.

En el Capítulo 5 (**Tipo árbol**) se incluyen preguntas y ejercicios sobre diferentes tipos de árboles: árbol binario, árbol binario de búsqueda, árbol AVL, árbol 2-3, etc.

En el Capítulo 6 (**Tipo conjunto**) se incluyen preguntas y ejercicios sobre la representación de los conjuntos, el tipo unión-búsqueda y la cola de prioridad.

Por último, en el Capítulo 7 (**Tipo grafo**) se incluyen preguntas y ejercicios sobre los grafos dirigidos y no dirigidos.

Por otro lado, en el Capítulo 8 (**Exámenes**), se proponen cuatro exámenes sin solución similares a los empleados en la asignatura "Programación y Estructuras de Datos". Para cada pregunta, ya sea de tipo test o de tipo ejercicio, se incluye una referencia (número de pregunta y página) a la explicación de la solución de la pregunta que aparece en los capítulos anteriores.

En el Capítulo 9 (**Solución de las preguntas de tipo test de los exámenes**), se incluye la solución (verdadero o falso) a las preguntas de tipo test de los exámenes propuestos en el Capítulo 8. Además, también se incluye una referencia a la explicación de la solución de la pregunta.

El libro se completa con un apéndice con la bibliografía recomendada y un índice alfabético que facilita la búsqueda de información.

Las preguntas planteadas para cada tema son:

- Introducción a los tipos abstractos de datos: 14 preguntas, 10 ejercicios.

- El lenguaje C++: 9 preguntas, 2 ejercicios.

- Tipos lineales: 7 preguntas, 8 ejercicios.

- Tipo árbol: 58 preguntas, 12 ejercicios.

- Tipo conjunto: 39 preguntas, 7 ejercicios.

- Tipo grafo: 11 preguntas, 5 ejercicios.

En total, este libro contiene 138 preguntas y 44 ejercicios.

## 1.6. Cómo utilizar este libro

Como ya se ha comentado, este libro pretende ser un material complementario que ayude a los alumnos a la preparación de los distintos exámenes de la asignatura. Teniendo en mente este objetivo, el libro se puede emplear de diversas formas:

- Para hacer ejercicios de forma individual. Utilizando el índice del libro (que se corresponde con el temario de la asignatura) el alumno puede seleccionar un tipo abstracto de datos concreto (por ejemplo, el árbol 2-3-4) y realizar tantos las preguntas como los ejercicios exclusivos de este tipo de datos.
- Para hacer exámenes tipo. Esta opción del libro permite al alumno realizar un examen global de todos los contenidos de la asignatura (tanto de test como de ejercicios). Sólo se presenta el enunciado del examen con los correspondientes enlaces a las soluciones; esto permite al alumno realizar un examen "en situación real" sin tener las soluciones delante.
- Cada una de las preguntas y ejercicios incluye su correspondiente solución, lo que permite al alumno realizar una autoevaluación de sus conocimientos adquiridos.

## 1.7. Agradecimientos

Queremos dar las gracias a todos los profesores que han impartido docencia en esta asignatura, cuya aportación a los contenidos de la misma ha sido fundamental. Asimismo, queremos agradecer a Francisco José Sánchez Alcaraz por su colaboración en los gráficos de este libro.

# 2

## Introducción a los tipos abstractos de datos

Los tipos de datos se definen mediante entidades abstractas (modelo matemático) y operaciones que se encapsulan de forma tal que la única vía para acceder o para modificar las entidades es a través de las operaciones abstractas. Esta idea ya estaba implícita en el lenguaje de programación SIMULA 67, en el que la designación sintáctica *clase* denota una colección de esas operaciones. A este encapsulamiento se le llama *Tipo Abstracto de Dato (TAD)*, haciendo referencia con el calificativo *abstracto*, a la idea de que, cuando se especifica, lo único fundamental son las cualidades, es decir, el *qué* se puede hacer, y no *cómo* se hace. Informalmente, podríamos definir un *tipo abstracto de datos* como un ente que tiene la propiedad de *instanciabilidad*, más un conjunto de operaciones, las únicas aceptadas para acceder a la representación (*privacidad*).

A través de la *abstracción de datos* se pueden obtener los objetivos de legibilidad, corrección, eficiencia, facilidad de mantenimiento y reutilización, todos ellos fundamentales para reducir la complejidad en el diseño e instrumentación de grandes proyectos o aplicaciones informáticas. Además, la abstracción de datos permitirá el cambio de representación de los datos sin afectar por ello a los programas que los utilicen. Es además particularmente importante porque simplifica la estructura de los programas que la usan debido a que presenta un interfaz de alto nivel.

## 2.1. Tipos abstractos de datos

### 2.1.1. Preguntas

(P.1) Sea el siguiente TAD:

```
MÓDULO NATURALEXAMEN
TIPO natural
OPERACIONES
  uno: → natural;
  siguiente: natural → natural;
  sumar: natural, natural → natural;
FMÓDULO
```

**Si  $N$  es un natural, entonces**

$$N = \text{sumar}(\text{uno}, \text{siguiente}(\text{uno}))$$

**es un uso sintácticamente incorrecto de la operación  $\text{sumar}$ .**

Falso.  $N = \text{sumar}(\text{uno}, \text{siguiente}(\text{uno}))$  es sintácticamente correcto, ya que:

- $\text{sumar}$  recibe dos argumentos de tipo natural y  $\text{uno}$  y  $\text{siguiente}$  devuelven un natural.
- $\text{sumar}$  devuelve un natural, lo que coincide con el tipo de  $N$ .

(P.2) Una operación consultora devuelve un valor del tipo definido.

Falso. Por definición, las operaciones consultoras de una especificación algebraica son aquéllas que devuelven un valor distinto del tipo definido.

(P.3) Para el tratamiento de errores en la especificación, se añaden funciones constantes que devuelven un valor del tipo que causa el error.

Verdadero. Las operaciones de una especificación algebraica siempre tienen que devolver un valor del tipo esperado en la sintaxis. Para aquellas situaciones donde se produce un error y no podemos devolver un valor, empleamos las funciones para el tratamiento de errores que devuelven un valor que indica una situación de error.

empleamos las funciones para el tratamiento de errores que devuelven un valor que indica una situación de error. Por ejemplo, en la especifica-

ción algebraica del TIPO natural definimos  $\text{errornat}$  que será utilizado en la ecuación  $\text{pred}(\text{cero}) = \text{errornat}$

Por ejemplo, en la especificación algebraica del tipo natural, no existe el predecesor de cero; para esta situación definimos  $\text{error}_{\text{naturales}}$  que es un valor que indica una situación de error:

$$\text{pred}(\text{cero}) = \text{error}_{\text{naturales}}$$

(P.4) Para definir la semántica de una operación de un tipo de datos sólo se pueden utilizar las operaciones constructoras generadoras.

Falso. Para definir la semántica de una operación se pueden utilizar operaciones constructoras (generadoras o modificadoras) y operaciones consultoras.

(P.5) Sea el siguiente TAD:

```
MÓDULO NATURALEXAMEN TIPO natural OPERACIONES
  uno: → natural;
  siguiente: natural → natural;
  sumar: natural, natural → natural;
FMÓDULO
```

**$\text{siguiente}(\text{siguiente}(\text{uno}))$  y  $\text{sumar}(\text{uno}, \text{siguiente}(\text{uno}))$  denotan el mismo valor dependiendo de las ecuaciones que se definen para estas operaciones.**

Verdadero. Una especificación algebraica para que sea completa debe tener definidas la sintaxis y la semántica para todas las operaciones. La semántica de una operación es la que realmente define su funcionamiento; dependiendo de las ecuaciones que definamos para estas operaciones podemos hacer que dos expresiones cualesquiera denoten el mismo valor.

Si elegimos las ecuaciones correctas para las operaciones  $\text{siguiente}$  y  $\text{sumar}$  las dos expresiones

$\text{siguiente}(\text{siguiente}(\text{uno}))$  y  $\text{sumar}(\text{uno}, \text{siguiente}(\text{uno}))$  denotan el mismo valor.

(P.6) La operación  $\text{BorrarItem}$  tiene la siguiente sintaxis y semántica:

```
BorrarItem: LISTA, ITEM → LISTA
BorrarItem(IC(L1, j), i) =
    si ( i == j ) entonces L1
    sino IC (BorrarItem(L1, i), j)
    fsi
```

**Esta operación borra la primera ocurrencia del elemento (item) que se encuentra en la lista.**

Verdadero. Esta operación tiene 2 argumentos: la lista y el elemento a borrar. Cuando encuentra la primera ocurrencia del elemento a borrar en la lista, la borra y no hace una llamada recursiva para borrar las siguientes ocurrencias.

Si no encuentra el elemento a borrar en la cabeza de la lista, inserta la cabeza de la lista en el resultado y hace una llamada recursiva con el resto de la lista.

**(P.7) Longitud : LISTA → NATURAL**

**Si L es una lista, a es un elemento (item) de la lista, entonces**

$a = \text{Longitud}(L)$

**es un uso sintácticamente correcto de la operación.**

Falso. Esta operación recibe una lista y devuelve un número natural (la longitud de la lista). La expresión es incorrecta porque estamos asignando a un elemento ( $a$ ) un número natural ( $\text{Longitud}(L)$ ).

**(P.8) Las operaciones auxiliares de un TAD también se les llama ocultas o privadas.**

Verdadero. Por definición las operaciones auxiliares se introducen en una especificación para facilitar su escritura y legibilidad. Son invisibles para los usuarios del TAD y también se les llama ocultas o privadas.

**(P.9) La especificación algebraica de la siguiente operación eliminaría todas las claves repetidas:**

```
VAR C: Conjunto; x, y: Item;
Eliminarp(Crear, y) ↔ Crear
Eliminarp(Insertar(C, x), y) ↔
    si(x == y) entonces C
    sino Insertar(Eliminar(C, y), x)
    fsi
```

Falso. Esta operación tiene 2 argumentos: el conjunto y la clave a borrar. Cuando encuentra la primera ocurrencia de la clave a borrar en el conjunto, la borra pero no hace una llamada recursiva para borrar las siguientes ocurrencias. Para que borrara todas las ocurrencias la segunda ecuación debería ser:

```
Eliminar(Insertar(C, x), y) ↔
    si(x == y) entonces Eliminar(C, y)
    sino Insertar(Eliminar(C, y), x)
    fsi
```

**(P.10) La especificación algebraica de la siguiente operación permite la inserción de claves repetidas:**

```
VAR C: ConjuntoConClavesRepetidas; x, y: Item;
Insertar(Insertar(C, x), y) ↔
    si (x == y) entonces Insertar(C, x)
    sino Insertar(Insertar(C, y), x)
    fsi
```

Falso. En la ecuación se comprueba si la clave a insertar es igual a alguna clave del conjunto para no insertarla. La ecuación correcta para permitir la inserción de claves repetidas sería:

```
Insertar(Insertar(C, x), y) ↔ Insertar(Insertar(C, y), x)
```

**(P.11) Dada una especificación de un TAD, sólo existe una implementación posible.**

Falso. Tal y como está definido un Tipo Abstracto de Datos (TAD), el calificativo de "Abstracto" es porque la manipulación de los datos sólo depende del comportamiento descrito en su especificación (qué hace) y es independiente de su implementación (cómo se hace). Una especificación permite múltiples implementaciones.

**(P.12) La operación BorrarItem tiene la siguiente sintaxis y semántica:**

```
BorrarItem: LISTA, ITEM → LISTA
BorrarItem(Crear, i) = Crear
BorrarItem(IC(L1, j), i) =
    si (i == j) entonces L1
```

```

sino IC(BorrarItem(L1, i), j)
fsi

```

Esta operación borra todas las ocurrencias del elemento (*item*) que se encuentra en la lista.

Falso. Esta operación tiene 2 argumentos: la lista y el elemento a borrar. Cuando encuentra la primera ocurrencia del elemento a borrar en la lista, la borra pero no hace una llamada recursiva para borrar las siguientes ocurrencias. Para que borrara todas las ocurrencias la segunda ecuación debería ser:

```

BorrarItem(IC(L1, j), i) =
  si (i == j) entonces BorrarItem(L1, i)
  sino IC(BorrarItem(L1, i), j)
  fsi

```

## 2.1.2. Ejercicios

(E.1) Sea la sintaxis y semántica de las siguientes operaciones del tipo árbol binario (AB) cuyos elementos son números naturales:

```

VAR i, d: AB; x: Natural;

Crea_AB () → AB

Enraizar(AB, Natural, AB) → AB

Raiz(AB) → Natural
  Raiz(Crea_AB()) = Error_Natural()
  Raiz(Enraizar(i, x, d)) = x

EsVacio(AB) → booleano
  EsVacio(Crea_AB()) = CIERTO
  EsVacio(Enraizar(i, x, d)) = FALSO

HijoIz(AB) → AB
  HijoIz(Crea_AB()) = Crea_AB()
  HijoIz(Enraizar(i, x, d)) = i

HijoDe(AB) → AB
  HijoDe(Crea_AB()) = Crea_AB()
  HijoDe(Enraizar(i, x, d)) = d

```

Se define la operación *Suma\_AB* que recibe un árbol binario y devuelve un *Natural* que representa la suma de las etiquetas de todos los nodos del árbol:  $\text{Suma\_AB}(AB) \rightarrow \text{Natural}$ .

A) Escribir únicamente con dos ecuaciones la semántica de esta operación.

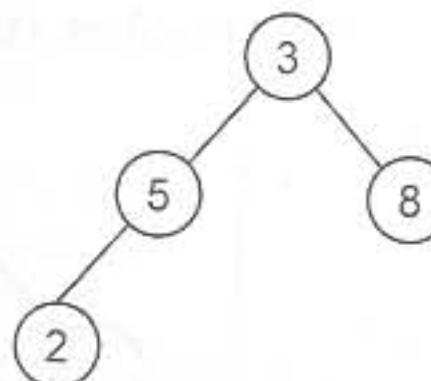
B) Se define la siguiente operación *Examen(AB)*. Explicar qué hace esta operación detallando el comportamiento de las dos ecuaciones que aparecen a continuación:

```

Examen(AB) → AB
  Examen(Crea_AB()) = Crea_AB()
  Examen(Enraizar(i, x, d)) =
    Enraizar(Examen(i), x + Suma_AB(i) + Suma_AB(d),
              Examen(d))

```

C) Dibujar el árbol binario resultante tras aplicar la operación *Examen* sobre el árbol binario:



SOLUCIÓN:

A) Cuando el árbol está vacío, la suma vale 0; cuando el árbol no está vacío, a la etiqueta del nodo actual le sumamos la suma de las etiquetas del subárbol izquierdo más la suma de las etiquetas del subárbol derecho.

```

Suma_AB(Crea_AB()) = 0
Suma_AB(Enraizar(i, x, d)) = x + Suma_AB(i) + Suma_AB(d)

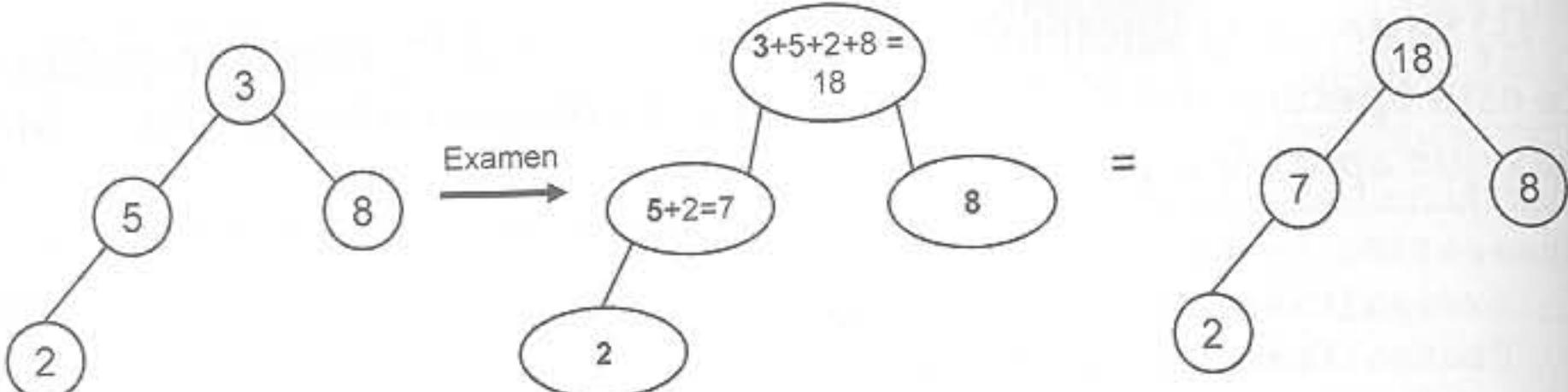
```

B) *Examen(AB)* es una operación que recibe un árbol binario *A* y devuelve otro árbol binario *B*:

- Si el árbol *A* está vacío, entonces el árbol *B* que devuelve también está vacío.
- Si el árbol *A* no está vacío, devuelve el árbol *B* con la misma estructura que el árbol *A*. Cada nodo de *B* contiene una etiqueta que es la suma de la etiqueta en la misma posición en *A* más la suma de

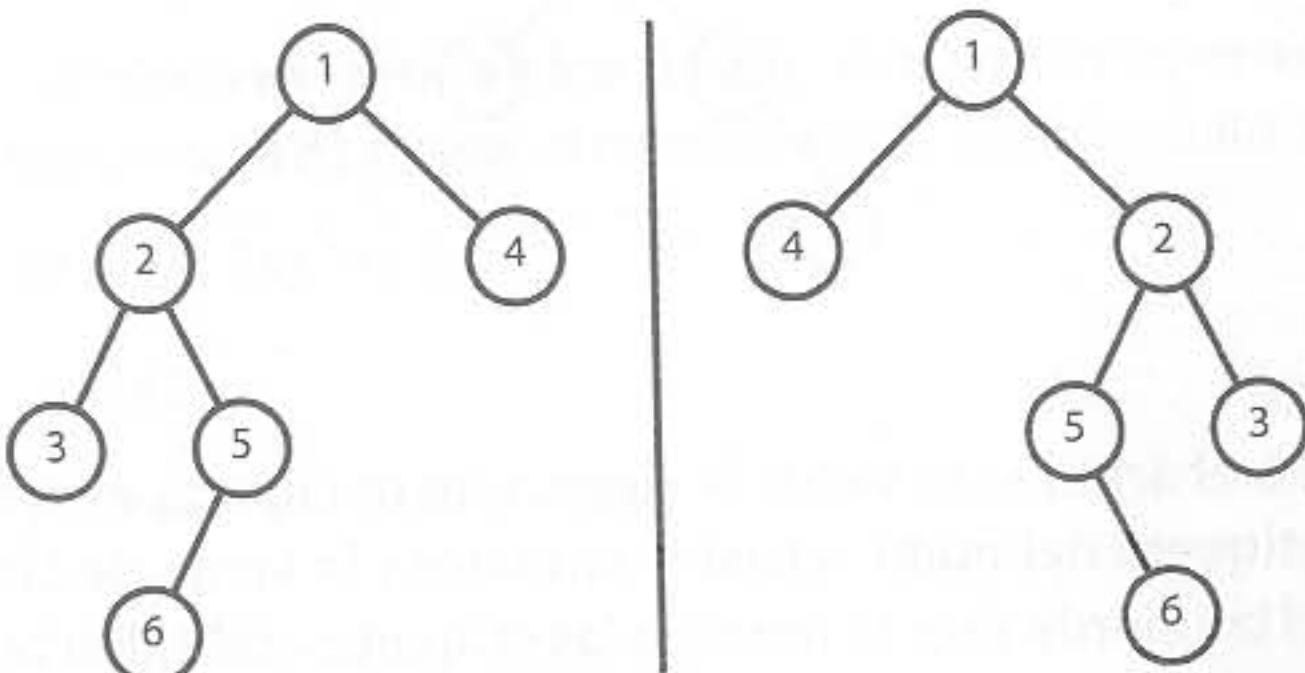
todas las etiquetas del subárbol izquierdo de ese nodo en  $A$  más la suma de todas las etiquetas del subárbol derecho de ese nodo en  $A$ .

C) El árbol resultante es:



(E.2) Sea un árbol binario cuyas etiquetas son números naturales. Especificar la sintaxis y la semántica de la operación *simetricos* que comprueba que 2 árboles binarios son simétricos.

NOTA: Éste es un ejemplo de 2 árboles simétricos.



### SOLUCIÓN:

Dos árboles  $a$  y  $b$  son simétricos si contienen el mismo elemento en la raíz y el hijo izquierdo de  $a$  es simétrico del hijo derecho de  $b$  y el hijo derecho de  $a$  es simétrico del hijo izquierdo de  $b$ . A partir de esta idea podemos plantear una ecuación recursiva para la operación *simetricos*. Sólo nos queda plantear los casos base: consideramos que dos árboles vacíos son simétricos y un árbol vacío con otro árbol no vacío no son simétricos. Se muestran dos soluciones posibles:

a) SINTAXIS: *simetricos(AB, AB)* → booleano

### SEMANTICA:

```

VAR x, y: Natural; iz, de, left, right: AB;
simetricos(crea_ab(), crea_ab()) = VERDADERO
simetricos(crea_ab(), enraizar(iz, x, de)) = FALSO
simetricos(enraizar(iz, x, de), crea_ab()) = FALSO
simetricos(enraizar(iz, x, de), enraizar(left, y, right)) =
  (x == y) & simetricos(iz, right) & simetricos(de, left)
  
```

b) SINTAXIS: *simetricos(AB, AB)* → booleano

### SEMANTICA:

```

VAR x, y: Natural; iz, de, left, right: AB;
simetricos(enraizar(iz, x, de), enraizar(left, y, right)) =
  si (x == y) entonces
    si simetricos(iz, right) & simetricos(de, left) entonces
      VERDADERO
    sino
      FALSO
    fsi
  sino
    FALSO
  fsi
  
```

(E.3) Dada la siguiente especificación formal de operaciones de listas de acceso por posición:

```

crear() → lista
inscabeza(lista, item) → lista
longitud(lista) → natural
  
```

A) Explicar qué hace la operación  $X$ , cuya sintaxis y semántica aparecen a continuación:

SINTAXIS:  
 $X(\text{lista}) \rightarrow \text{lista}$

SEMÁNTICA:  
 $\text{VAR } i: \text{item}; l: \text{lista};$

```

X(crear()) = crear()
si(longitud(l) == 0) entonces
  X(inscabeza(l, i)) = crear()
sino
  X(inscabeza(l, i)) = inscabeza(X(l), i)
fsi
  
```

**B) Simplificar la siguiente expresión: ( $IC = \text{inscabeza}$ )**

$$X(IC(IC(IC(crear(), a), b), c), d))$$
**SOLUCIÓN:**

A) La operación  $X$  recibe una lista y devuelve una lista. La lista que devuelve es igual a la que recibe como parámetro pero sin el elemento de la última posición. Con más detalle:

- Si la lista que recibe está vacía, devuelve la lista vacía.
- Si la lista que recibe posee un único elemento, devuelve la lista vacía: elimina el único elemento de la lista, que también es el último elemento.
- Si la lista posee más de un elemento, realiza llamadas recursivas a  $X$  hasta que se llega a una lista formada por un único elemento (el caso anterior): devuelve la lista original sin el último elemento.

B) La simplificación es:

$$\begin{aligned} X(IC(IC(IC(crear(), a), b), c), d)) &= \\ IC(X(IC(IC(crear(), a), b), c)), d) &= \\ IC(IC(X(IC(crear(), a)), b), c), d) &= \\ IC(IC(IC(X(IC(crear(), a))), b), c), d) &= \\ IC(IC(crear(), b), c), d) \end{aligned}$$

**(E.4) Sea el conjunto de los números naturales con las operaciones de cero y sucesor:**

```

MÓDULO NATURAL
USA Bool
TIPO natural
OPERACIONES
  cero: → natural
  suc: natural → natural
  
```

Define la semántica de las nuevas operaciones  $==$  y  $<=$  que permitan una ordenación de los elementos del conjunto:

**SINTAXIS**

$==: \text{natural, natural} \rightarrow \text{Bool}$   
 $<=: \text{natural, natural} \rightarrow \text{Bool}$

**SOLUCIÓN:**

Para la operación  $==$  se planteará una ecuación recursiva: diremos que dos números naturales son iguales si son iguales sus antecesores.

Como sólo disponemos de las operaciones cero y sucesor, cualquier número natural será cero o sucesor de otro número natural. Consiguientemente, se plantearán 3 casos base:

- Cero es igual a cero.
- Cero no es igual a cualquier número.
- Cualquier número no es igual a cero.

En el caso general se especificará que dados 2 números distintos de cero (representados por sucesor de otro número), éstos serán iguales si lo son sus antecesores. Esta ecuación recursiva acabará por cualquiera de los 3 casos base, dependiendo de que sean iguales (caso base 1), de que el primer número sea más pequeño que el segundo (caso base 2) o de que el primer número sea mayor que el segundo (caso base 3).

En el caso de  $<=$  sólo cambia el segundo caso base, ya que cero es menor o igual que cualquier número.

```

VAR a, b, n: natural;

== (cero, cero) = VERDAD
== (cero, suc(n)) = FALSO
== (suc(n), cero) = FALSO
== (suc(a), suc(b)) = == (a, b)

<= (cero, cero) = VERDAD
<= (cero, suc(n)) = VERDAD
<= (suc(n), cero) = FALSO
<= (suc(a), suc(b)) = <= (a, b)
  
```

**(E.5) Completa en esta misma hoja las ecuaciones que aparecen a continuación y que expresan el comportamiento de las operaciones de concatenar, primera y ultima, y borraultimo (borra el último elemento de la lista) en una lista de acceso por posición. La sintaxis de las operaciones es la siguiente:**

```

VAR x: elemento; l1, l2: lista;
concatenar(lista, lista) → lista
concatenar(crear(), l1) = ...
concatenar(l1, crear()) = ...
concatenar(inscabeza(l1, x), l2) = ...

primera(lista) → posicion; ultima(lista) → posicion
primera(crear()) = ...
ultima(crear()) = ...
si esvacia(l1) entonces ultima(inscabeza(l1, x)) = ...
sino ultima(inscabeza(l1, x)) = ...
fsi

borraultimo(lista) → lista
borraultimo(crear()) = ...
si esvacia(l1) entonces borraultimo(inscabeza(l1, x)) = ...
sino borraultimo(inscabeza(l1, x)) = ...
fsi

```

**SOLUCIÓN:**

Para la operación *concatenar* se planteará una ecuación recursiva que concatenará la lista *l1* (a la que se le va quitando su primer elemento en cada llamada recursiva) y la lista *l2*; éste primer elemento de *l1* se debe almacenar en el resultado final, por lo que se realiza una llamada a *inscabeza* antes de la llamada recursiva. Se plantea un caso base para el fin de la recursividad: cuando se concatena una lista vacía y la lista *l1* devuelve *l1*. Por último, se plantea otro caso base, para el caso de que la segunda lista sea vacía.

En las operaciones *primera* y *ultima* cuando se aplican sobre una lista vacía devuelven un error del tipo que esperan estas operaciones, es decir, devuelven un error del tipo *posicion*. La operación *ultima* cuando se realiza sobre una lista con un único elemento es la misma que la *primera* (caso base). Si la lista no está vacía, se realiza una llamada recursiva a la lista original sin el primer elemento (este elemento no se debe almacenar en el resultado final, ya que se tiene que devolver una *posicion*); esta recursividad acaba cuando quede una lista con un elemento.

Para la operación *borraultimo* se plantean dos casos base: si se aplica *borraultimo* sobre una lista vacía se devuelve una lista vacía; si se aplica sobre una lista con un único elemento hay que borrar este elemento, por lo que devuelve la lista vacía. En la ecuación recursiva se van realizando llamadas a una lista sin su primer elemento hasta que lleguemos a una lista con un único elemento. En este caso, como estos elementos se deben

almacenar en el resultado final (ya que se tiene que devolver una lista con todos los elementos de la lista original excepto el último) se realiza una llamada a *inscabeza* previa a la llamada recursiva.

```

VAR x: elemento; l1, l2: lista;
concatenar(crear(), l1) = l1
concatenar(l1, crear()) = l1
concatenar(inscabeza(l1, x), l2) =
    inscabeza(concatenar(l1, l2), x)

primera(crear()) = error_posicion()
ultima(crear()) = error_posicion()
si esvacia(l1) entonces ultima(inscabeza(l1, x)) =
    primera(inscabeza(l1, x))
sino ultima(inscabeza(l1, x)) = ultima(l1)
fsi

borraultimo(crear()) = crear()
si esvacia(l1) entonces borraultimo(inscabeza(l1, x)) =
    crear()
sino borraultimo(inscabeza(l1, x)) =
    inscabeza(borraultimo(l1), x)
fsi

```

**(E.6) Dada la sintaxis y la semántica de la operación Examen:**

```

Examen(lista) → lista
VAR l: lista; x, y: item;
Examen(crear()) = crear()
Examen(IC(crear(), x)) = IC(crear(), x)
Examen(IC(IC(l, y), x)) = IC(IC(Examen(l), x), y)

```

**A) Explicar el funcionamiento de la operación *Examen*.**

**B) Aplicar la operación *Examen* a la lista:**

*IC(IC(IC(crear(), c), b), a)*

**SOLUCIÓN:**

A) La operación *Examen* recibe una lista y devuelve una lista. La lista que devuelve es de la misma longitud que la original en la que se han intercambiado los elementos de las posiciones pares por los elementos de las posiciones impares. Con más detalle:

- Si la lista que recibe está vacía, devuelve la lista vacía.

- Si la lista que recibe posee un único elemento, devuelve la misma lista.
- Si la lista posee más de un elemento, realiza llamadas recursivas a *Examen* hasta que se llega a una lista vacía o a una lista formada por un único elemento (los dos casos anteriores). Previamente a la llamada recursiva, se intercambia el elemento de la segunda posición por el de la primera posición.

Si la lista contiene un número impar de elementos, el último no cambia su posición.

B) Si se aplica la operación *Examen* sobre la lista

$IC(IC(IC(crear(), c), b), a)$   
se obtiene:

$$\begin{aligned} Examen(IC(IC(IC(crear(), c), b), a)) = \\ IC(IC(Examen(IC(crear(), c)), a), b) = \\ IC(IC(IC(crear(), c), a), b) \end{aligned}$$

#### (E.7) Dada la sintaxis y semántica de la siguiente operación

$examen(pila, pila) \rightarrow \text{bool}$

que actúa sobre una pila, aplica esta operación sobre una pila de tres elementos para posteriormente explicar razonadamente qué hace (di qué hace y por qué):

**OPERACIÓN**  
 $examen(pila, pila) \rightarrow \text{bool}$

**OPERACIONES AUXILIARES**

$examenAux3(pila, pila, pila) \rightarrow \text{bool}$   
 $examenAux2(pila, pila) \rightarrow \text{bool}$

**VAR**  $p, q, r: \text{pila}; x, y, z: \text{item};$

**ECUACIONES**

- (1)  $\text{examen(crear(), crear())} = \text{CIERTO}$
- (2)  $\text{examen(apilar}(p, x), \text{crear()}) = \text{FALSO}$
- (3)  $\text{examen(crear(), apilar}(q, y)) = \text{FALSO}$
- (4)  $\text{examen(apilar}(p, x), \text{apilar}(q, y)) =$   
 $\text{examenAux3}(p, \text{apilar}(q, y), \text{apilar}(crear(), x))$
- (5)  $\text{examenAux3}(\text{apilar}(p, x), \text{apilar}(q, y), \text{apilar}(r, z)) =$   
 $\text{examenAux3}(p, \text{apilar}(q, y), \text{apilar}(\text{apilar}(r, z), x))$

- (6)  $\text{examenAux3}(\text{crear(), apilar}(q, y), \text{apilar}(r, z)) =$   
 $\text{examenAux2}(\text{apilar}(q, y), \text{apilar}(r, z))$
- (7)  $\text{examenAux2}(\text{crear(), apilar}(r, z)) = \text{FALSO}$
- (8)  $\text{examenAux2}(\text{apilar}(q, y), \text{crear()}) = \text{FALSO}$
- (9)  $\text{examenAux2}(\text{crear(), crear()}) = \text{CIERTO}$
- (10)  $\text{si } y == z \text{ entonces}$   
 $\text{examenAux2}(\text{apilar}(q, y), \text{apilar}(r, z)) = \text{examenAux2}(q, r)$
- (11)  $\text{sino } \text{examenAux2}(\text{apilar}(q, y), \text{apilar}(r, z)) = \text{FALSO}$   
 $\text{fsi}$

**SOLUCIÓN:**

La operación *examen* actúa sobre dos pilas y devuelve un valor booleano: devuelve CIERTO si una pila es la inversa de la otra y FALSO en caso contrario. Cuando la operación *examen* actúa sobre dos pilas que son idénticas (contienen los mismos elementos), sólo puede devolver CIERTO cuando las pilas contenga los mismos elementos del tope al fondo y en sentido inverso.

La operación *examen* funciona de la siguiente forma:

- Dadas las pilas  $p$  (primer parámetro) y  $q$  (segundo parámetro), la operación desapila todos los elementos de la pila  $p$  y los apila en una pila auxiliar vacía  $r$  (operación *examenAux3()*, ecuaciones 4, 5 y 6). De esta forma, en  $r$  obtenemos una pila que contiene los elementos de forma inversa a la pila  $p$ .
- Cuando se han desapilado todos los elementos (ecuación 6, cuando  $p$  es la pila vacía *crear()*), se procede a comparar la pila  $q$  con la pila  $r$  elemento a elemento (operación *examenAux2()*, ecuaciones 7, 8, 9, 10 y 11). Para ello, se realiza una llamada recursiva a la operación *examenAux2()*, desapilando cada vez un elemento de ambas pilas (ecuación 10). Si en algún momento, alguno de los elementos no es el mismo (ecuación 11) se devuelve FALSO. Si después de desapilar y comparar todos los elementos, las dos pilas están vacías, significa que eran idénticas (ecuación 9) y se devuelve CIERTO ( $q$  y  $r$  son idénticas, por tanto  $p$  es la inversa de  $q$  y viceversa).
- Las ecuaciones 7, 8 y 9 definen los casos base de *examenAux2()*. Las ecuaciones 1, 2 y 3 definen los casos base de *examen()*.

A continuación se muestra un ejemplo de aplicación de la operación *examen* sobre una pila de tres elementos. En el examen había que hacer

un solo ejemplo. Sin embargo, aquí incluimos dos ejemplos: en uno devuelve FALSO y en el otro CIERTO. Para abbreviar,  $A()$  significa *apilar()*.

**Ejemplo 1:**  $p$  y  $q$  son la misma pila. Devuelve FALSO.

$$\begin{aligned} t &= A(A(A(crear(), a), b), c) \\ u &= A(A(A(crear(), a), b), c) \end{aligned}$$

1. Por 4:

$$\begin{aligned} examen(t, u) &= \\ examen(A(A(A(crear(), a), b), c), u) &= \\ examenAux3(A(A(crear(), a), b), u, A(crear(), c)) & \end{aligned}$$

2. Por 5:

$$\begin{aligned} examenAux3(A(A(crear(), a), b), u, A(crear(), c)) &= \\ examenAux3(A(crear(), a), u, A(A(crear(), c), b)) & \end{aligned}$$

3. Por 5:

$$\begin{aligned} examenAux3(A(crear(), a), u, A(A(crear(), c), b)) &= \\ examenAux3(crear(), u, A(A(A(crear(), c), b), a)) & \end{aligned}$$

4. Por 6:

$$\begin{aligned} examenAux3(crear(), u, A(A(A(crear(), c), b), a)) &= \\ examenAux2(u, A(A(A(crear(), c), b), a)) &= \\ examenAux2(A(A(A(crear(), a), b), c), A(A(A(crear(), c), b), a)) & \end{aligned}$$

5. Por 11:

$$\begin{aligned} examenAux2(A(A(A(crear(), a), b), c), A(A(A(crear(), c), b), a)) &= \\ FALSO & \end{aligned}$$

**Ejemplo 2:**  $p$  es la inversa de  $q$  y viceversa. Devuelve CIERTO.

$$\begin{aligned} t &= A(A(A(crear(), a), b), c) \\ u &= A(A(A(crear(), c), b), a) \end{aligned}$$

1. Por 4:

$$\begin{aligned} examen(t, u) &= \\ examen(A(A(A(crear(), a), b), c), u) &= \\ examenAux3(A(A(crear(), a), b), u, A(crear(), c)) & \end{aligned}$$

2. Por 5:

$$\begin{aligned} examenAux3(A(A(crear(), a), b), u, A(crear(), c)) &= \\ examenAux3(A(crear(), a), u, A(A(crear(), c), b)) & \end{aligned}$$

3. Por 5:

$$\begin{aligned} examenAux3(A(crear(), a), u, A(A(crear(), c), b)) &= \\ examenAux3(crear(), u, A(A(A(crear(), c), b), a)) & \end{aligned}$$

4. Por 6:

$$\begin{aligned} examenAux3(crear(), u, A(A(A(crear(), c), b), a)) &= \\ examenAux2(u, A(A(A(crear(), c), b), a)) &= \\ examenAux2(A(A(A(crear(), c), b), a), A(A(A(crear(), c), b), a)) & \end{aligned}$$

5. Por 10:

$$\begin{aligned} examenAux2(A(A(A(crear(), c), b), a), A(A(A(crear(), c), b), a)) &= \\ examenAux2(A(A(crear(), c), b), A(A(crear(), c), b)) & \end{aligned}$$

6. Por 10:

$$\begin{aligned} examenAux2(A(A(crear(), c), b), A(A(crear(), c), b)) &= \\ examenAux2(A(crear(), c), A(crear(), c)) & \end{aligned}$$

7. Por 10:

$$\begin{aligned} examenAux2(A(crear(), c), A(crear(), c)) &= \\ examenAux2(crear(), crear()) & \end{aligned}$$

8. Por 9:

$$examenAux2(crear(), crear()) = CIERTO$$

(E.8) Completa en esta misma hoja las ecuaciones que aparecen a continuación y que expresan el comportamiento de las operaciones de: *obtener* en una lista de acceso por posición, *suma* y *multiplicación* en el conjunto de los números Naturales en el que existen la operación *cero* :  $\rightarrow$  natural y la operación *suc* : natural  $\rightarrow$  natural (devuelve el sucesor de un número Natural).

```
VAR l1: lista; x: item; p: posicion; a: natural;

obtener(lista, posicion) → item
obtener(crear(), p) = ...
si p == primera (inscabeza(l1, x)) entonces
    obtener(inscabeza(l1, x), p) = ...
```

```

sino
  obtener(inscabeza(l1, x), p) = ...
fsi

suma(natural, natural) → natural
suma(      a      ,      cero    ) = ...
suma(      ,      ) = ...

multiplicacion(natural, natural) → natural
multiplicacion(   ,      ) = ...
multiplicacion(   ,      ) = ...

```

#### SOLUCIÓN:

Para la operación *obtener* se planteará una ecuación recursiva que finalizará cuando la posición apunte al primer elemento de la lista, en cuyo caso se devolverá este elemento. Si la posición no apunta a la cabeza de la lista se realiza una llamada recursiva con la lista original sin su primer elemento. Si se ha recorrido toda la lista y no se ha encontrado la posición, se realizará una llamada a *obtener* con la lista vacía (caso base) y se devolverá un error del tipo esperado por la operación (error del tipo elemento, *error\_item*).

En la operación *suma* se plantean un caso base y un caso general. En el caso base, la suma de cualquier número con cero es cero. En el caso general, se plantea una ecuación recursiva que acaba en el caso base, es decir, se va decrementando el segundo número hasta llegar a cero. Para ello, se especifica que la suma de dos números *a* y *b* es igual al sucesor de la suma de *a* y *b* decrementado en una unidad.

La operación *multiplicacion* se plantea en función de la suma. La multiplicación de dos números *a* y *b* es equivalente a realizar *b* sumas del número *a*. El número *b* se va decrementando en cada llamada recursiva hasta llegar al caso base (la multiplicación de cualquier número por cero es cero).

```

VAR l1: lista; x: item; p: posicion; a: natural;

obtener(lista, posicion) → item
obtener(crear(), p) = error_item
si p == primera (inscabeza(l1, x)) entonces
  obtener(inscabeza(l1, x), p) = x
sino
  obtener(inscabeza(l1, x), p) = obtener(l1, p)
fsi

```

```

suma(natural, natural) → natural
suma(a, cero) = a
suma(a, suc(b)) = suc(suma(a, b))

multiplicacion(natural, natural) → natural
multiplicacion(a, cero) = cero
multiplicacion(a, suc(b)) = suma(a, multiplicacion(a, b))

```

(E.9) Especificar la sintaxis y la semántica de la operación *Examen*, que actúa sobre un árbol binario y devuelve CIERTO si todos los nodos tienen dos hijos (excepto los nodos hoja, evidentemente).

#### SOLUCIÓN:

A continuación se muestran dos formas distintas de resolver este ejercicio, aunque la idea que subyace en las dos soluciones es la misma: se plantea una ecuación recursiva en la que se comprueba que cada nodo del árbol tenga 2 hijos no vacíos o tenga 2 hijos vacíos (es una hoja). Para ello, se plantean cuatro casos base:

- (1) Un árbol vacío se asume que tiene dos hijos.
- (2) Una hoja tiene dos hijos vacíos.
- (3) y (4) Un árbol que tiene un hijo vacío y otro no vacío, no tiene dos hijos.

Mediante el uso de variables booleanas se indican que los casos base (1) y (2) devuelven CIERTO (son árboles que tienen dos hijos) y que los casos base (3) y (4) devuelven FALSO.

En el caso general se plantea una ecuación recursiva que comprueba que el hijo de la izquierda y el hijo de la derecha tengan dos hijos (mediante una llamada recursiva a la operación para el hijo de la izquierda y otra llamada para el hijo de la derecha). Si en algún momento algún árbol sólo tiene un hijo se devolverá FALSO para esa llamada y, consecuentemente, el resultado final será FALSO.

a) SINTAXIS  
*Examen(arbin)* → bool

SEMÁNTICA  
 VAR i, d: arbin; x, y: item;

```

Examen(crea_arbin()) = CIERTO
Examen(enraizar(crea_arbin(), x, crea_arbin())) = CIERTO
Examen(enraizar(crea_arbin(), x, enraizar(i, y, d))) = FALSO
Examen(enraizar(enraizar(i, y, d), x, crea_arbin())) = FALSO
Examen(enraizar(i, x, d)) = Examen(i) & Examen(d)

```

## b) SINTAXIS

Examen(arbin) → bool

## SEMÁNTICA

VAR i, d: arbin; x: item;

```

Examen(crea_arbin()) = CIERTO
si ((esvacío(i) & no esvacío(d)) &
    (no esvacío(i) & esvacío(d))
    entonces Examen(enraizar(i, x, d)) = FALSO
sino
    Examen(enraizar(i, x, d)) = Examen(i) & Examen(d)
fsi

```

## 2.2. Complejidad

### 2.2.1. Preguntas

**(P.13)** La complejidad temporal de las operaciones de inserción y búsqueda en un árbol de búsqueda digital es  $O(n)$  siendo  $n$  el número de elementos del conjunto.

Falso. La complejidad de las operaciones de inserción y búsqueda dependen de la altura del árbol ya que es un árbol de búsqueda. En el peor de los casos hay que realizar tantos pasos como altura tenga el árbol, es decir,  $O(h)$  donde  $h$  es  $N + 1$  siendo  $N$  el número de bits de la clave.

**(P.14)** En la escala de complejidades, la complejidad logarítmica es menor que la lineal.

Verdadero. Por definición, en la escala de complejidades, la complejidad logarítmica es menor que la lineal. En general, de mejor a peor complejidad están: constante, logarítmica, lineal, polinómica y exponencial.

### 2.2.2. Ejercicios

**(E.10)** Calcular la complejidad temporal en su peor caso de los siguientes fragmentos de código C++:

A)

```

for(i = 0; i < n; i++) {
    for(j = 1, sum = a[0]; j <= i; j++) sum += a[j];
    cout << "La suma del subarray " << i << " es ";
    cout << sum << endl;
}

```

B)

```

for(i = 4; i < n; i++) {
    for(j = i - 3, sum = a[i-4]; j <= i; j++) sum += a[j];
    cout << "La suma del subarray " << i - 4 << " es ";
    cout << sum << endl;
}

```

## SOLUCIÓN:

A) Dado que el bucle exterior se repite  $n$  veces, y el bucle interior se repite  $i$  veces, con  $i$  desde 1 hasta  $n - 1$ , se obtendría el siguiente número de pasos:

$$\begin{aligned} \text{Nº pasos: } & \sum_{i=0 \dots n-1} (1 + \sum_{j=1 \dots i} (1)) = \\ & \sum_{i=0 \dots n-1} (1 + i) = \\ & (1 + n) * n / 2 = (n^2 + n) / 2 \end{aligned}$$

Con lo que la complejidad en su peor caso sería  $O(n^2)$ .

B) El bucle exterior se repite  $n - 4$  veces. El bucle interior, aunque parece que el número de veces que se ejecuta varía, siempre se ejecuta 4 veces ( $j = i - 3 \dots i$ ). Por tanto, el número de pasos sería  $\sum_{i=4 \dots n-1} (1) = n - 4$ . Con lo que la complejidad en su peor caso sería:  $O(n)$ .

## El lenguaje C++

El lenguaje de programación C++ es uno de los más empleados en la actualidad. Se puede decir que C++ es un lenguaje híbrido, ya que permite programar tanto en estilo procedural (como si fuese C) como en estilo orientado a objetos o como en ambos a la vez. Además, también se puede emplear mediante programación basada en eventos para crear programas que usen interfaz gráfico de usuario.

El nacimiento de C++ se sitúa en el año 1980, cuando Bjarne Stroustrup, de los laboratorios Bell, desarrolló una extensión de C llamada “C with Classes” que permitía aplicar los conceptos de la programación orientada a objetos con el lenguaje C. Stroustrup se basó en las características de orientación a objetos del lenguaje de programación Simula, aunque también tomó ideas de otros lenguajes importantes de la época como ALGOL68 o ADA.

Durante los siguientes años, Stroustrup continuó el desarrollo del nuevo lenguaje y en 1983 se acuñó el término C++.

En 1985, Bjarne Stroustrup publicó la primera versión de “The C++ Programming Language” (Addison-Wesley, 1985), que a partir de entonces se convirtió en el libro de referencia del lenguaje.

En la actualidad, este lenguaje se encuentra estandarizado a nivel internacional con el estándar ISO/IEC 14882:1998 con el título “Information Technology - Programming Languages - C++”, publicado el 1 de septiembre de 1998. En septiembre del año 2011 se publicó una versión corre-

gida del estándar (ISO/IEC 14882:2011). Finalmente, en la actualidad se está preparando una nueva versión del lenguaje, llamada “C++14”, que se espera que esté lista para el año 2014.

### 3.1. Preguntas

**(P.15) Sea el método Primera perteneciente a la clase TLista que devuelve la primera posición de la lista que lo invoca:**

```
class TLista {
public:
    ...
private:
    TNodo *lis;
}

TPosicion TLista::Primera() {
    TPosicion p;
    p.pos = lis;
    return p;
}
```

En la instrucción `return p;` del método `Primera` se invoca al constructor de copia de `TPosicion`.

Verdadero. El método `Primera` devuelve un objeto de tipo `TPosicion` por valor. Cuando un método devuelve un objeto por valor, se invoca automáticamente al constructor de copia correspondiente a la clase del objeto devuelto.

**(P.16) En layering los métodos de la clase derivada pueden acceder a la parte pública de la clase base.**

Verdadero. El layering es la declaración de una clase que posee algún miembro de dato que es un objeto de otra clase. Desde la clase “derivada” (la clase que contiene al objeto de la otra clase) se puede acceder a la parte pública de la clase “base”, como en cualquier otra situación (no afecta para nada que sea layering).

**(P.17) En herencia pública, la parte privada de la clase base es accesible desde los métodos de la clase derivada.**

Falso. La parte privada de una clase sólo es accesible desde los métodos de la propia clase. Si queremos que una parte de una clase no sea accesible desde el exterior de la clase (esté oculta), pero que sí que sea accesible desde la clase derivada, entonces se tiene que declarar como protegida (`protected`).

**(P.18) Para que se ejecute el constructor de copia de la clase base, éste debe ser invocado explícitamente en la fase de inicialización de la clase derivada.**

Verdadero. Cuando se realiza cualquier tipo de herencia, en el constructor de copia de la clase derivada se debe invocar explícitamente (en la fase de inicialización) al constructor de copia de la clase base.

**(P.19) En herencia privada los métodos de la clase derivada pueden acceder a la parte pública de la clase base.**

Verdadero. Cuando se realiza herencia privada, los miembros public de la base quedan como `private` en la derivada, por lo que todos los métodos de la clase derivada pueden acceder a ellos.

**(P.20) La sobrecarga del operador corchete tiene que definirse de la siguiente forma para que pueda aparecer a ambos lados de una asignación: `Titem& operator[] (int i);`**

Falso. Depende de la versión del compilador. Por ejemplo, en la versión 4.0.3 es necesario definir dos métodos para la sobrecarga del operador corchete:

- `Titem& operator[] (int i); // Cuando es parte izquierda de un operador (por ejemplo, el operador asignación).`
- `Titem operator[] (int i) const; // Cuando es parte derecha de un operador.`

**(P.21) Sea el método Primera perteneciente a la clase TLista que devuelve la primera posición de la lista que lo invoca:**

```
class TLista {
public:
    ...
private:
    TNodo *lis;
```

```

}

TPosicion TLista::Primera() {
    TPosicion p;
    p.pos = lis;
    return p;
}

```

**En el método Primera se invoca al constructor y destructor para el objeto TPosicion p.**

Verdadero. El objeto p es una variable local: cuando se declara se invoca a su constructor por defecto y cuando sale de ámbito (al acabar el método Primera) se invoca a su destructor.

**(P.22) Si definimos un método en la parte privada de una clase, éste será accesible desde los métodos de la propia clase y desde todas sus clases derivadas.**

Falso. La parte privada de una clase sólo es accesible desde los métodos de la propia clase. Si queremos que una parte de una clase sea accesible desde una clase derivada de ésta, entonces se tiene que declarar como protegida (protected).

**(P.23) En C++, el puntero this sólo se puede usar dentro de los métodos de la clase.**

Verdadero. Por definición, el puntero this se puede usar sólo dentro de los métodos de la clase. Este puntero apunta al propio objeto al que pertenece ese método. Por último, (\*this) representa al objeto completo.

## 3.2. Ejercicios

**(E.11) Dada la siguiente declaración de la clase TArbin que representa un árbol binario de búsqueda cuyas etiquetas son enteros y donde no se permiten etiquetas repetidas, escribe el código del constructor por defecto TArbin(), del destructor ~TArbin() y del método Insertar(int), que devuelve cierto si el número se puede insertar y falso en caso contrario. Escribe también el código del constructor y destructor de la clase TNodo. Importante: se tiene que escribir el código de todos los métodos auxiliares que se empleen.**

```

class TArbin {
public:
    TArbin();
    ...
    ~TArbin();
    ...
    bool Insertar(int);
    ...

private:
    TNodo *n;
};

class TNodo {
public:
    TNodo();
    ...
    ~TNodo();
    ...

private:
    int etiqueta;
    TArbin hijoI, hijoD;
};

```

SOLUCIÓN:

Clase TArbin:

```

TArbin::TArbin() {
    n = NULL;
}

TArbin::~TArbin() {
    if(n != NULL)
    {
        delete n;
        n = NULL;
    }
}

bool
TArbin::Insertar(int item) {
    if(n == NULL)
    {
        n = new TNodo;
        if(n == NULL)

```

```

        return false;
    n->etiqueta = item;
    return true;
}
else
{
    if(item < n->etiqueta)
        return (n->hijoI).Insertar(item);
    else if(item > n->etiqueta)
        return (n->hijoD).Insertar(item);
    else
        return false;
}
}

```

Clase TNodo:

```

TNodo::TNodo() {
    etiqueta = 0;
}

TNodo::~TNodo() {
    etiqueta = 0;
}

```

**(E.12)** Dada la siguiente declaración de la clase TLista que representa una lista ordenada (de menor a mayor) doblemente enlazada de números enteros donde no se permiten repetidos, escribe el código del constructor por defecto TLista(), del destructor ~TLista() y del método Insertar(int), que devuelve cierto si el número se puede insertar y falso en caso contrario. Escribe también el código del constructor y destructor de la clase TNodo. Importante: se tiene que escribir el código de todos los métodos auxiliares que se empleen.

```

class TLista {
public:
    TLista();
    ...
    ~TLista();
    ...
    bool Insertar(int);
    ...

private:
    TNodo *primero, *ultimo;
}

```

```

};

class TNodo {
public:
    TNodo();
    ...
    ~TNodo();
    ...

private:
    int etiqueta;
    TNodo *anterior, *siguiente;
};

```

SOLUCIÓN:

Clase TLista:

```

TLista::TLista() {
    primero = NULL;
    ultimo = NULL;
}

TLista::~TLista() {
    TNodo *aux;

    while(primero)
    {
        aux = primero;
        primero = primero->siguiente;

        delete aux;
    }

    primero = NULL;
    ultimo = NULL;
}

bool
TLista::Insertar(int item) {
    if(primero == NULL)
    {
        primero = new TNodo;
        if(primero == NULL)
            return false;

        ultimo = primero;
    }
    else
    {
        TNodo *nuevo = new TNodo;
        if(nuevo == NULL)
            return false;

        nuevo->etiqueta = item;
        nuevo->anterior = ultimo;
        nuevo->siguiente = primero;
        primero->anterior = nuevo;
        primero = nuevo;
        ultimo = nuevo;
    }
}

```

```

primero->etiqueta = item;
return true;
}
else
{
    TNodo *aux, *nuevo;

    if(item < primero->etiqueta)
    {
        nuevo = new TNodo;
        if(nuevo == NULL)
            return false;

        nuevo->etiqueta = item;
        nuevo->siguiente = primero;
        primero->anterior = nuevo;
        primero = nuevo;
        return true;
    }

    aux = primero;
    while(aux)
    {
        if(item < aux->etiqueta)
        {
            nuevo = new TNodo;
            if(nuevo == NULL)
                return false;

            nuevo->etiqueta = item;
            nuevo->anterior = aux->anterior;
            nuevo->siguiente = aux;
            aux->anterior->siguiente = nuevo;
            aux->anterior = nuevo;
            return true;
        }
        else if(item == aux->etiqueta)
            return false;
        aux = aux->siguiente;
    }

    nuevo = new TNodo;
    if(nuevo == NULL)
        return false;

    nuevo->etiqueta = item;
}

```

```

nuevo->anterior = ultimo;
ultimo->siguiente = nuevo;
ultimo = nuevo;
return true;
}
}

```

## Clase TNodo:

```

TNodo::TNodo() {
    etiqueta = 0;
    anterior = NULL;
    siguiente = NULL;
}

TNodo::~TNodo() {
    etiqueta = 0;
    anterior = NULL;
    siguiente = NULL;
}

```

# 4

## Tipos lineales

El vector es una estructura de datos presente en casi todos los lenguajes de programación y, en muchas ocasiones, casi la única que se proporciona de forma directa. Es por ello que tradicionalmente ha sido la base para la construcción de otras estructuras de datos más complejas. A la hora de definir un vector muchos programadores lo “ven” como un conjunto consecutivo de posiciones de memoria. Es esta una interpretación sesgada, influenciada por la representación que a bajo nivel (nivel de lenguaje máquina) utiliza el compilador de un lenguaje de programación para construir un vector (aunque no siempre se instrumentan así). Hay que distinguir, por tanto, entre el concepto abstracto de “tipo de datos vector” y la representación que de ella hace un compilador dado en una máquina concreta.

Las listas constituyen una estructura flexible porque pueden aumentar o disminuir su tamaño según se requiera. Además, los elementos (en inglés *items*) son accesibles y se pueden insertar y suprimir en cualquier posición de la lista. Las pilas y colas surgirán al imponer restricciones a la forma en que se puede acceder a los elementos de una lista. Pero, aparte de estas restricciones, las pilas y colas son listas de elementos.

## 4.1. Preguntas

**(P.24)** En una cola representada a partir de una lista enlazada simple con un único puntero al principio de la lista (*cabeza de la cola*), todas las operaciones de la cola (*Cabeza*, *Encolar*, *Desencolar* y *EsVacia*) se realizan en tiempo constante.

Falso. Las operaciones de *Cabeza*, *Desencolar* y *EsVacia* se realizan en tiempo constante. Sin embargo, la operación *Encolar* tiene una complejidad temporal lineal ya que hay que recorrer la lista (cola) para insertar el nuevo elemento al final de la misma (*fondo* de la cola).

**(P.25)** Una lista es una secuencia de cero o más elementos de cualquier tipo de la forma:  $e_p, e_{sig(p)}, \dots$

Falso. Por definición, una lista es una secuencia de cero o más elementos de un mismo tipo de la forma  $e_1, e_2, \dots, e_n (\forall n \geq 0)$ .

**(P.26)** Una lista de acceso por posición es una secuencia de cero o más elementos que pueden ser de distinto tipo.

Falso. Los elementos no pueden ser de distinto tipo. Por definición, una lista es una secuencia de cero o más elementos de un mismo tipo de la forma  $e_1, e_2, \dots, e_n (\forall n \geq 0)$ .

**(P.27)** La operación de insertar un elemento en una lista ordenada tiene el mismo coste que la de insertar un elemento en un vector ordenado.

Falso. La complejidad temporal de la operación *insertar* depende de la implementación de la lista y de los algoritmos utilizados. Si se asumen algoritmos similares y una representación de lista enlazada, la complejidad temporal de la operación *insertar*, en el peor de los casos, es la misma (lineal) ya que hay que recorrer todas las posiciones de ambas estructuras para insertar el elemento en la última posición. Sin embargo, en el mejor caso la complejidad de *insertar* en la lista es constante (primera posición de la lista) mientras que en el vector es lineal (hay que insertar en la primera posición y desplazar todos los elementos una posición a la derecha).

**(P.28)** La complejidad temporal de la operación apilar en una pila siempre es  $O(1)$ .

Falso. La complejidad temporal depende de la implementación de la pila y del algoritmo utilizado. Por ejemplo, si utilizamos un *array* para implementar la pila y utilizamos un algoritmo que realice las inserciones por la primera componente, la complejidad de apilar un elemento sería lineal (hay que desplazar todos los elementos cada vez que insertemos uno nuevo); por otra parte, si utilizamos un algoritmo que utilice un cursor que indique la cima de la pila la complejidad de apilar un elemento sería constante.

**(P.29)** La complejidad temporal de la operación *insertar* en una lista es independiente de su implementación.

Falso. La complejidad temporal de la operación *insertar* depende de la implementación de la lista y del algoritmo utilizado. Por ejemplo, si se utiliza una representación secuencial (con un *array*) y un algoritmo que utiliza un cursor para indicar el último elemento la complejidad es constante; por otra parte, si se utiliza una representación enlazada (con punteros a nodos) la complejidad para insertar un elemento es lineal.

**(P.30)** Un vector es un conjunto ordenado de pares <índice, valor>.

Verdadero. Por definición, un vector es un conjunto ordenado de pares <índice, valor>. Para cada índice definido dentro de un rango finito existe asociado un valor. En términos matemáticos es una correspondencia entre los elementos de un conjunto de índices y los de un conjunto de valores.

## 4.2. Ejercicios

**(E.13)** ¿Qué hace la operación *X* con la sintaxis y semántica siguientes:

```
VAR i, j: ITEM; L1: Lista;
X: LISTA, ITEM → POSICION
X(Crear, i) = ERRORPOSICION
X(InsCabeza(L1, i), j) =
    si (i == j) entonces Primera(InsCabeza (L1, i))
    sino X(L1, j)
    fsi
```

**SOLUCIÓN:**

La operación *X* devuelve la posición donde se encuentra el elemento (*item*) dado como parámetro. Si no lo encuentra devuelve un error de tipo posición.

**(E.14) Escribir la sintaxis y la semántica de un operador que a partir de una lista, devuelve el primer elemento de la lista.**

**SOLUCIÓN:**

```
SINTAXIS:  
Primer(lista) → item  
  
SEMÁNTICA:  
VAR x: item; L: lista;  
  
Primer(InsCabeza(L, x)) = x  
Primer(Crear()) = ERRORITEM
```

**(E.15) Define la sintaxis y la semántica de la operación *inverso* que actúa sobre un vector de números naturales y devuelve el vector inverso del vector de entrada. Para ello se utilizará exclusivamente las operaciones constructoras generadoras del vector. Se asume que el tamaño del vector es una constante determinada, *tam*, y que están definidas y se pueden usar todas las operaciones de números naturales (*suc*, *suma*, *resta*, *multiplicacion* y *division*).**

**SOLUCIÓN:**

```
SINTAXIS:  
inverso(vector) → vector  
  
SEMÁNTICA:  
VAR v: vector; i, x: natural;  
  
inverso(Crear_vector()) = Crear_vector()  
inverso(asig(v, i, x)) =  
    asig(inverso(v), tam + suc(cero) - i, x)
```

**(E.16) Define la sintaxis y la semántica de la operación *insertar* vista en clase que inserta un elemento en una lista con acceso por posición.**

**SOLUCIÓN:**

```
SINTAXIS:  
insertar( lista, posicion, item ) → lista  
  
SEMÁNTICA:  
VAR L1: lista; x, y: item; p: posicion;  
  
insertar(crear_lista(), p, x) = crear_lista()  
si p == primera(inscabeza(L1, x)) entonces  
    insertar(inscabeza(L1, x), p, y) = inscabeza(inscabeza(L1,  
        y), x)  
sino  
    insertar(inscabeza(L1, x), p, y) = inscabeza(insertar(L1,  
        p, y), x)  
fsi
```

**(E.17) Define la sintaxis y semántica de la operación *posicion* que actúa sobre un vector y devuelve la posición menor sobre la que se ha asignado el valor que recibe como parámetro. Si no se ha asignando el valor en el vector se debe devolver 0. Ejemplo:**

```
posicion(asig(Crear(), 3, b), a) = 0  
posicion(asig(asig(Crear(), 3, b), 1, b), b) = 1  
posicion(asig(asig(asig(Crear(), 3, b), c, 2), 1, b), b) = 1  
posicion(asig(asig(asig(Crear(), 1, b), c, 2), 3, b), b) = 1
```

**SOLUCIÓN:**

```
SINTAXIS:  
posicion(vector, item) → entero  
posicionAux(vector, item, entero) → entero  
  
SEMÁNTICA:  
VAR v:vector; i: entero; x: item;  
  
posicion(Crear(), x) = 0  
si (x <> y) entonces  
    posicion(asig(v, i, x), y) = posicion(v, y)  
sino  
    posicion(asig(v, i, x), y) = posicionAux(v, y, i)  
fsi  
posicionAux(Crear(), x, i) = i  
si (x <> y) entonces  
    posicionAux(asig(v, i, x), y, j) = posicionAux(v, y, j)  
sino si (i < j) entonces
```

```

  posicionAux(asig(v, i, x), y, j) = posicionAux(v, y, i)
  sino
  posicionAux(asig(v, i, x), y, j) = posicionAux(v, y, j)
  fsi

```

**(E.18) Dar la sintaxis y la semántica de la operación *mezcla*, que actúa sobre dos colas y devuelve una cola nueva con los elementos de las dos colas encolados de forma alternada y en el mismo orden que aparecen en cada cola.**

Por ejemplo, suponiendo que C1 y C2 son dos colas, escritas de izquierda a derecha desde la cabeza al fondo de la cola, la mezcla de C1 y C2 debería dar el siguiente resultado:

```

C1 = (a b c)
C2 = (x y)

mezcla(C1, C2) = (a x b y c)

```

SOLUCIÓN:

SINTAXIS:  
 $\text{mezcla}(\text{cola}, \text{cola}) \rightarrow \text{cola}$

Métodos auxiliares:  
 $\text{mezclaAux}(\text{cola}, \text{cola}, \text{cola}) \rightarrow \text{cola}$   
 $\text{invierte}(\text{cola}) \rightarrow \text{cola}$   
 $\text{invierteAux}(\text{cola}, \text{cola}) \rightarrow \text{cola}$

SEMÁNTICA:  
 $\text{VAR } c, d, e: \text{cola}; x, y: \text{item};$   
 $\text{mezcla}(c, d) = \text{mezclaAux}(\text{invierte}(c), \text{invierte}(d), \text{crear}())$   
 $\text{mezclaAux}(\text{crear}(), \text{crear}(), c) = c$   
 $\text{mezclaAux}(\text{crear}(), \text{encolar}(c, x), d) = \text{mezclaAux}(\text{crear}(), c, \text{encolar}(d, x))$   
 $\text{mezclaAux}(\text{encolar}(c, x), \text{crear}(), d) = \text{mezclaAux}(c, \text{crear}(), \text{encolar}(d, x))$   
 $\text{mezclaAux}(\text{encolar}(c, x), \text{encolar}(d, y), e) = \text{mezclaAux}(c, d, \text{encolar}(\text{encolar}(e, x), y))$   
 $\text{invierte}(c) = \text{invierteAux}(c, \text{crear}())$   
 $\text{invierteAux}(\text{crear}(), c) = c$   
 $\text{invierteAux}(\text{encolar}(c, x), d) = \text{invierteAux}(c, \text{encolar}(d, x))$

**(E.19) Dada la especificación algebraica del TAD lista estudiado en clase, escribe la sintaxis y semántica de la operación *ordenar* que realiza la ordenación ascendente de un lista de números enteros.**

Nota: se puede emplear y devolver una lista auxiliar para realizar la ordenación.

SOLUCIÓN:

Se emplea ( $IC = \text{inscabeza}$ ).

SINTAXIS:  
 $\text{ordenar}(\text{lista}) \rightarrow \text{lista}$   
 $\text{ordenarAux}(\text{lista}, \text{lista}) \rightarrow \text{lista}$

SEMÁNTICA:  
 $\text{VAR } l1, l2: \text{lista}; x, y: \text{entero};$   
 $\text{ordenar}(\text{crear}()) = \text{crear}()$   
 $\text{ordenar}(l1) = \text{ordenarAux}(l1, \text{crear}())$   
 $\text{ordenarAux}(\text{crear}(), l2) = l2$   
 $\text{ordenarAux}(IC(l1, x), \text{crear}()) = \text{ordenarAux}(l1, IC(\text{crear}(), x))$   
 $\text{ordenarAux}(IC(l1, x), IC(l2, y)) =$   
     si  $x \leq y$  entonces  
          $\text{ordenarAux}(l1, IC(IC(l2, y), x))$   
     sino  
          $\text{ordenarAux}(l1, IC(\text{ordenarAux}(IC(\text{crear}(), x), l2), y))$   
     fsi

**(E.20) A partir de la especificación algebraica de la cola, escribe la sintaxis y semántica de la operación *M* que recibe dos colas y devuelve una cola nueva en la que se han encolado de forma alternada los elementos de las dos colas, empezando por la primera cola. Por ejemplo:**

```

C1 = (a, b, c, d)
C2 = (1, 2, 3)

M(C1, C2) = (a, 1, b, 2, c, 3, d)

```

SOLUCIÓN:

SINTAXIS:  
 $M(\text{cola}, \text{cola}) \rightarrow \text{cola}$   
 $Aux1(\text{cola}, \text{cola}, \text{cola}) \rightarrow \text{cola}$

```

Aux2(cola, cola, cola) → cola

SEMÁNTICA:
VAR c, c1, c2: cola; x: item;

M(c1, c2) = Aux1(crear(), c1, c2)

Aux1(c, crear(), crear()) = c
Aux1(c, encolar(c1, x), crear()) =
  Aux1(encolar(c, cabeza(c1)), desencolar(encolar(c1, x)),
        crear())
Aux1(c, crear(), encolar(c1, x)) =
  Aux1(encolar(c, cabeza(c1)), crear(), desencolar(encolar(c1,
    x)))
Aux1(c, encolar(c1, x), c2) =
  Aux2(encolar(c, cabeza(c1)), desencolar(encolar(c1, x)), c2)

Aux2(c, crear(), crear()) = c
Aux2(c, encolar(c1, x), crear()) =
  Aux2(encolar(c, cabeza(c1)), desencolar(encolar(c1, x)),
        crear())
Aux2(c, crear(), encolar(c1, x)) =
  Aux2(encolar(c, cabeza(c1)), crear(), desencolar(encolar(c1,
    x)))
Aux2(c, c1, encolar(c2, x)) =
  Aux1(encolar(c, cabeza(c2)), c1, desencolar(encolar(c2, x)))

```

Otra solución:

```

SINTAXIS:
M(cola, cola) → cola
Concatena(cola, cola) → cola

SEMÁNTICA:
VAR c1, c2: cola;

M(crear(), c1) = c1
M(c1, crear()) = c1
M(c1, c2) = Concatena(encolar(encolar(crear(), cabeza(c1)),
  cabeza(c2)), M(desencolar(c1), desencolar(c2)))

Concatena(crear(), c1) = c1
Concatena(c1, crear()) = c1
Concatena(c1, c2) = concatena(encolar(c1, cabeza(c2)),
  desencolar(c2))

```

(E.21) A partir de la especificación algebraica de la lista, escribe la sintaxis y semántica de la operación *SPP* que recibe una lista y devuelve una sublista en función de la posición inicial y la posición final indicadas; si la posición inicial no pertenece a la lista, se debe devolver una lista vacía; si la posición final no pertenece a la lista o es anterior a la posición inicial, se debe devolver una sublista desde la posición inicial hasta el final de la lista. Por ejemplo:

```

L = (e, a, f, b, c)
pe = posición letra e
pa = posición letra a
pb = posición letra b

SPP(L, pe, pb) = (e, a, f, b)
SPP(L, pa, pb) = (a, f, b)
SPP(L, pa, pe) = (a, f, c)

```

SOLUCIÓN:

Se emplea (*IC* = *inscabeza*).

```

SINTAXIS:
SPP(lista, posición, posicion) → lista

SEMÁNTICA:
VAR L: lista; x: item; p1, p2: posición;

SPP(crear(), p1, p2) = crear()
si primera(IC(L, x)) == p1 y p1 != p2 entonces
  SPP(IC(L, x), p1, p2) = IC(SPP(L, primera(L), p2), x)
/* También se puede escribir como:
   SPP(IC(L, x), p1, p2) = IC(SPP(L, siguiente(IC(L, x), p1),
  p2), x)
*/
sino si primera(IC(L, x)) == p1 entonces
  SPP(IC(L, x), p1, p2) = IC(crear(), x)
sino
  SPP(IC(L, x), p1, p2) = SPP(L, p1, p2)
fsi

```

Otra solución (otra forma de escribir lo mismo):

```

SINTAXIS:
SPP(lista, posición, posicion) → lista
SPaux(lista, posición) → lista

SEMÁNTICA:

```

```

VAR L: lista; x: item; p1, p2: posición;

SPP(crear(), p1, p2) = crear()
si primera(IC(L, x) == p1) entonces
  SPP(IC(L, x), p1, p2) = IC(SPPaux(L, p2), x)
sino
  SPP(IC(L, x), p1, p2) = SPP(L, p1, p2)
fsi

SPPaux(crear(), p2) = crear()
si primera(IC(L, x) == p2) entonces
  SPPaux(IC(L, x), p2) = IC(crear(), x)
sino
  SPPaux(IC(L, x), p2) = IC(SPPaux(L, p2), x)
fsi

```

## 5

**Tipo árbol**

En este capítulo se introduce el tipo de datos árbol, constituido por elementos que mantienen una estructura jerárquica, obtenida a partir de las estructuras lineales al eliminar el requisito de que cada elemento tenga como máximo un sucesor. Esto significa que tenemos opción entre diferentes "siguentes".

Un árbol es un conjunto finito de nodos, tal que:

1. hay un nodo especial llamado raíz
2. el resto de nodos están repartidos en  $n$  conjuntos disjuntos ( $n \geq 0$ )  $A_1, A_2, \dots, A_n$ , donde cada uno de estos conjuntos es a su vez un árbol. A los conjuntos  $A_1, A_2, \dots, A_n$  se les llama subárboles.

Un árbol que no tiene ningún nodo se le llama árbol vacío o nulo. Existen muchos términos a los que nos referiremos cuando hablemos de árboles:

- a) **Nodo:** está formado por un elemento (*item*) de información y las ramas a otros nodos.
- b) **Grado de un nodo:** es el número de subárboles no vacíos de un nodo.
- c) **Grado de un árbol:** es el máximo grado de los nodos de un árbol.
- d) **Hojas:** son los nodos de grado 0, también llamados nodos terminales.

- e) **Hijos de un nodo x:** son las raíces de los subárboles de dicho nodo, al que se llama padre. Todos los nodos tienen un parente, excepto el raíz.
- f) **Hermanos:** son los nodos hijos del mismo parente.
- g) **Antecesores a un nodo x:** son todos los nodos que se encuentran en el camino desde la raíz a dicho nodo.
- h) **Camino:** es una sucesión de nodos  $n_1, n_2, \dots, n_k$  de un árbol tal que  $n_i$  es el parente de  $n_{i+1}$  para  $1 \leq i < k$ .
- i) **Longitud de un camino:** es el número de arcos que se recorren en un camino.
- j) **Nivel de un nodo:**

- El nivel de un árbol vacío es 0.
- El nivel de la raíz es 1.
- Si un nodo está en el nivel  $i$ , sus hijos están en el nivel  $i + 1$ .

- k) **Profundidad (altura) de un árbol:** es el máximo nivel de los nodos del árbol.

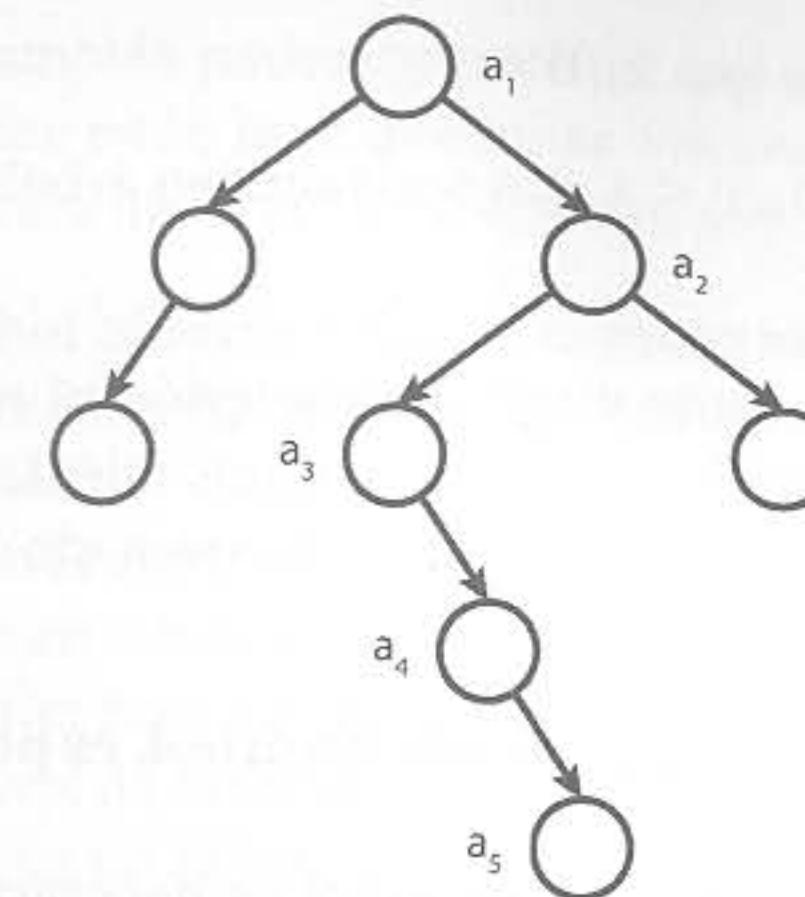
## 5.1. Árbol

### 5.1.1. Preguntas

**(P.31) Un camino en un árbol es una secuencia  $a_1, a_2, \dots, a_s$  de árboles tal que para todo  $i \in \{1, \dots, s - 1\}$ ,  $a_{i+1}$  es subárbol de  $a_i$ .**

Verdadero. Es la definición de camino. Aunque un camino se define como una secuencia de nodos, también se puede definir como una secuencia de árboles, en la que los elementos del camino son la sucesión de nodos raíz de los árboles que forman la secuencia.

Como ejemplo podemos ver la siguiente figura, en la que hay un camino de  $a_1$  a  $a_5$ :  $a_1, a_2, a_3, a_4, a_5$ .



$$\forall i \in \{1 \dots 4\} a_{i+1} \text{ es subárbol de } a_i, \\ \text{Longitud} = 5 - 1 = 4$$

**(P.32) El grado de un árbol es el máximo nivel de los nodos de un árbol.**

Falso. El grado de un árbol es el máximo número de hijos que pueden tener sus subárboles, mientras que el máximo nivel de un árbol es la altura que tiene un árbol.

**(P.33) Todo árbol general es un árbol multicamino de búsqueda.**

Falso. Un árbol multicamino de búsqueda  $T$  es un árbol  $m$ -ario vacío o es un árbol no vacío que cumple las siguientes propiedades:

1. La raíz de  $T$  es un nodo del tipo:

$n, A_0, (k_1, A_1), (k_2, A_2), \dots, (k_n, A_n)$   
donde:

$A_i, 0 \leq i \leq n$  son los subárboles del árbol  $T$

$k_i, 1 \leq i \leq n$  son las etiquetas del árbol  $T$

$1 \leq n < m$

2.  $k_i < k_{i+1}, 1 \leq i < n$

3. Todos los valores de la clave del subárbol  $A_i$ :

- son menores que  $k_{i+1}, 0 \leq i < n$

- son mayores que  $k_i$ ,  $0 < i \leq n$

4. Los subárboles  $A_i$ ,  $0 \leq i \leq n$  son también árboles multicamino de búsqueda.

Un árbol general no tiene ninguna restricción ni regla sobre los elementos (*items*) de los nodos que lo componen, mientras que el concepto de multicamino de búsqueda implica un determinado orden entre los elementos de los nodos.

**(P.34) Dado un único recorrido de un árbol, es posible reconstruir dicho árbol.**

Falso. Para poder reconstruir un árbol es necesario conocer el recorrido en inorder y cualquier otro (niveles, preorden o postorden). Solamente se podría reconstruir el árbol dado un único recorrido si además tenemos información de su estructura, por ejemplo, si el árbol es un árbol binario completo o lleno.

## 5.2. Árbol binario

### 5.2.1. Preguntas

**(P.35) En un árbol binario, el camino mínimo de la raíz es igual a la altura del árbol.**

Falso. El camino mínimo de la raíz es igual a la longitud del camino más corto desde la raíz hasta un árbol vacío.

El árbol binario no dispone de ninguna estrategia de equilibrado en sus algoritmos de inserción o borrado, por lo que el árbol puede estar desequilibrado.

En un caso extremo en el que los elementos a insertar en un árbol binario estuviesen ordenados ascendentemente o descendentemente, el árbol binario tendría la forma de una lista. En este caso, el camino mínimo es distinto a la altura del árbol. Esto sería un caso extremo en el que podemos apreciar evidentemente que el camino mínimo de la raíz no es igual a la altura del árbol, pero también podemos poner un ejemplo con un árbol equilibrado.

Supongamos que tenemos un árbol binario completo, que no es lleno. Esto implica que no existen todos los nodos del último nivel del árbol.

En este ejemplo, también podemos concluir que el camino mínimo de la raíz será a cualquier nodo hoja que no se encuentre en el último nivel, mientras que la altura del árbol será un nivel más.

**(P.36) En un árbol binario lleno, el camino mínimo de la raíz (longitud del camino más corto de ese nodo hasta un árbol vacío) es igual a la altura del árbol.**

Verdadero. Según la definición de árbol binario lleno, todos los nodos hoja del árbol están en el mismo nivel, por lo que todos los caminos desde la raíz hasta cualquier hoja son de la misma longitud. La altura se define como el máximo nivel de los nodos de un árbol. Las hojas se encuentran en el máximo nivel de un árbol.

**(P.37) A partir del recorrido por niveles de un árbol binario completo se puede obtener el árbol al que representa.**

Verdadero. Para reconstruir un árbol binario a partir de sus recorridos, necesitamos tener al menos el recorrido en inorder y cualquier otro recorrido, por ejemplo niveles. Si solo tenemos un recorrido, como en este caso, es obligatorio conocer la estructura del árbol. Si esta estructura es la de un árbol binario completo o lleno, podemos reconstruir el árbol únicamente con el recorrido por niveles.

**(P.38) El mayor elemento (*item*) almacenado en un árbol binario de búsqueda siempre se encuentra en un nodo hoja.**

Falso. Las hojas son árboles con un solo nodo, es decir, una raíz con dos subárboles vacíos. El árbol binario de búsqueda tiene que cumplir la condición de árbol multicamino de búsqueda en el que los elementos están ordenados de forma que la raíz tiene un elemento que es mayor que todos los elementos del subárbol de la izquierda y menor que todos los elementos de la derecha. Si tenemos un árbol binario de búsqueda cuya raíz solamente tiene subárbol izquierdo, estamos demostrando con un ejemplo que el mayor elemento almacenado se encuentra en la raíz y la raíz no es un nodo hoja.

**(P.39) Sólo existen tres formas de recorrido en profundidad en un árbol binario: preorden, inorder y postorden.**

Falso. Los recorridos preorden, inorder, postorden, corresponden con los recorridos de izquierda a derecha, pero existen también los recorridos que van de derecha a izquierda. Estos son RDI (raíz, derecha, izquierda),

DRI (derecha, raíz, izquierda) y DIR (derecha, izquierda, raíz).

**(P.40) El máximo número de nodos en un nivel  $i - 1$  de un árbol binario es  $2^{i-2}$ ,  $i \geq 2$ .**

Verdadero. El máximo número de nodos en el nivel  $i$  es  $2^{i-1}$ . Por tanto en el nivel  $i-1$ , tendremos que sustituir en la ecuación la  $i$ :  $2^{i-1-1} = 2^{i-2}$ .

**(P.41) Existe un único árbol binario completo que se puede construir a partir del recorrido en postorden.**

Verdadero. Para reconstruir un árbol binario a partir de sus recorridos, necesitamos tener al menos el recorrido en inorder y cualquier otro recorrido, por ejemplo postorden. Si solo tenemos un recorrido como en este caso es obligatorio conocer la estructura del árbol. Si esta estructura es la de un árbol binario completo o lleno, podemos reconstruir el árbol únicamente con el recorrido en postorden.

## 5.2.2. Ejercicios

**(E.22) Sea un árbol binario cuyas etiquetas son números naturales. Especificar la sintaxis y la semántica de la operación *arbCompleto* que comprueba si el árbol binario cumple las propiedades de árbol completo. Para ello se podrán utilizar las operaciones del tipo árbol visto en clase:**

```
MODULO ARBOLES BINARIOS
USA BOOL, NATURAL

PARAMETRO TIPO item
OPERACIONES
  <, ==, >: item, item → bool
  error_item() → item
FFPARAMETRO
TIPO arbin
OPERACIONES
  crea_arbin() → arbin
  enraizar(arbin, item, arbin) → arbin
  raiz(arbin) → item
  esvacio(arbin) → bool
  hijoiz, hijode(arbin) → arbin
  altura(arbin) → natural
  arbLleno(arbin) → bool // Se asume que ya está definido
  arbCompleto(arbin) → bool
```

### SOLUCIÓN:

```
VAR i, d: arbin; x, y: item;

arbCompleto(crea_arbin()) = TRUE
arbCompleto(enraizar(crea_arbin(), x, crea_arbin())) = TRUE
arbCompleto(enraizar(crea_arbin(), x, enraizar(crea_arbin(),
y, crea_arbin()))) = FALSE
arbCompleto(enraizar(enraizar(crea_arbin(), y, crea_arbin()),
x, crea_arbin())) = TRUE
arbCompleto(enraizar(i, x, d)) =

// Para comprobar si diferencia de alturas es 1
si (altura(i) == suc(altura(d))) entonces
  arbCompleto(d) y arbLleno(i)
sino
  si altura(i) == altura(d) entonces
    arbLleno(i) y arbCompleto(d)
  sino
    FALSE
fsi
fsi
```

**(E.23) Realizar la especificación algebraica de la función:  
Camino(Lista, ABB) → bool**

Esta función recibe dos parámetros: una lista de números enteros (Lista) y un árbol binario de búsqueda de números enteros (ABB). La función devuelve cierto si la lista (Lista) es un camino del árbol binario de búsqueda (ABB) y falso en otro caso.

Nota: La lista y el árbol binario no contienen elementos repetidos. Se pueden utilizar todas las funciones definidas en clase de las especificaciones de los números naturales, listas y árboles binarios. Se supone que también están definidas las funciones de comparación de números naturales:  $>$ ,  $<$ ,  $==$ .

### SOLUCIÓN:

```
VAR x, y: natural; i, d: ABB; L: lista;

Camino(crear(), crear()) = TRUE
Camino(crear(), enraizar(i, x, d) = TRUE
Camino(inscabeza(L, x), crear()) = FALSE
Camino(inscabeza(L, x), enraizar(i, y, d) =
si (x==y)
  si (obtener(L, primera(L)) == raiz(i))
```

```

    Camino(L, i)
sino
    si (obtener(L, primera(L)) == raiz(d))
        Camino(L, d)
    sino
        FALSE
    fsi
fsi
sino
    si (x < y)
        Camino(inscabeza(L, x), i)
    sino
        Camino(inscabeza(L, x), d)
    fsi
fsi

```

**(E.24) ¿Cuál es el máximo número de nodos en un nivel  $i$  de un árbol binario? ¿Y en un árbol de grado 4?**

SOLUCIÓN:

La solución es  $2^{i-1}$ ,  $4^{i-1}$ ,  $i \geq 1$ .

Se llega a esta solución al desarrollar la serie de elementos que se encuentran al desarrollar los nodos de un árbol completo, mediante la representación secuencial al numerar los nodos de cada nivel de izquierda a derecha:

Nivel 1: 1 (nodo raíz)

Nivel 2: 1 2 (los hijos del nodo raíz)

Nivel 3: 1 2 (los hijos del hijo izquierdo del nodo raíz) 3 4 (los hijos del hijo derecho del nodo raíz)

Nivel 4: 1 2 3 4 5 6 7 8

...

Nivel  $i$ :  $2^{i-1}$

Para el caso del árbol de grado 4 se desarrollaría de forma similar, con la variante de que cada nodo tendrá 4 hijos.

**(E.25) ¿Cuál es el máximo número de nodos en un árbol binario de altura  $k$ ? ¿Y en un árbol de grado 4 para la misma altura?**

SOLUCIÓN:

Se llega a esta solución de forma similar a la del ejercicio anterior (ya se trate de un árbol binario o de grado 4), exceptuando que la numeración no se reinicia en cada nivel del árbol:

Altura 1: 1 (nodo raíz)

Altura 2: 2 3 (los hijos del nodo raíz)

Altura 3: 4 5 (los hijos del hijo izquierdo del nodo raíz) 6 7 (los hijos del hijo derecho del nodo raíz)

Altura 4: 8 9 10 11 12 13 14 15

...

Altura  $i$ :  $2^i - 1$

### 5.3. Árbol binario de búsqueda

#### 5.3.1. Preguntas

**(P.42) El elemento (item) medio (según la relación de orden en la búsqueda) almacenado en un árbol binario de búsqueda lleno siempre se encuentra en la raíz.**

Verdadero. Para poder comprenderlo mejor, podemos crear un árbol binario de búsqueda lleno de altura 2. Según la relación de orden el elemento que se encuentra en la raíz es el elemento medio. Si hacemos crecer el árbol a altura 3, tendremos que hacer crecer el árbol dos elementos por la rama izquierda y dos por la rama derecha, de forma que el elemento medio sigue siendo la raíz.

**(P.43) Si el nodo a borrar en un árbol binario de búsqueda tiene 2 hijos, éste se puede sustituir por el hijo mayor del subárbol derecho.**

Falso. Si el nodo a borrar en un árbol binario de búsqueda tiene 2 hijos, éste se puede sustituir por el hijo mayor del subárbol izquierdo o por el hijo menor del subárbol derecho.

**(P.44) Suponiendo que tenemos un árbol binario de búsqueda lleno con  $n$  elementos, la búsqueda del elemento número  $n/2$  según la relación de orden se realiza en tiempo logarítmico.**

Falso. Si se trata de un árbol binario de búsqueda lleno con  $n$  elementos, el elemento  $n/2$  es el elemento que se encuentra en la raíz. La búsqueda se realiza en tiempo constante.

**(P.45) El elemento (item) medio (según la relación de orden en la búsqueda) almacenado en un árbol binario de búsqueda siempre se encuentra en la raíz.**

Falso. El árbol binario de búsqueda no tiene ninguna estrategia de restructuración para mantener el equilibrio del árbol durante las operaciones de inserción o de borrado, por lo que es muy posible que el árbol esté desequilibrado. Si el árbol se encuentra desequilibrado, el elemento medio (según la relación de orden en la búsqueda) estará en el lado hacia donde se produzca ese desequilibrio, por lo que no estará en la raíz.

## 5.4. Árbol enhebrado

### 5.4.1. Preguntas

**(P.46) En un árbol binario enhebrado, el primer nodo del recorrido en inorder no tiene hebra izquierda.**

Verdadero. Por definición, en un árbol enhebrado cuando un nodo no tiene hijo derecho, se sustituye el valor nulo por su sucesor en inorder (hebra derecha); cuando no tiene hijo izquierdo, se sustituye el valor nulo por su predecesor en inorder (hebra izquierda). De esta definición se deduce que el primer nodo del recorrido en inorder no tiene hebra izquierda ya que no tiene predecesor en inorder.

**(P.47) En un árbol binario enhebrado todos los nodos tienen 0 ó 2 hebras ó 2 hijos ó 1 hijo y 1 hebra.**

Falso. Por definición, en un árbol enhebrado cuando un nodo no tiene hijo derecho, se sustituye el valor nulo por su sucesor en inorder (hebra derecha); cuando no tiene hijo izquierdo, se sustituye el valor nulo por su predecesor en inorder (hebra izquierda). De esta definición se deduce que el primer nodo del recorrido en inorder no tiene hebra izquierda y que el último nodo del recorrido en inorder no tiene hebra derecha y además, como máximo, podrían tener un hijo derecho y un hijo izquierdo respectivamente. Consecuentemente, estos nodos ni tienen 2 hebras, ni tienen 2 hijos ni pueden tener un hijo y una hebra.

**(P.48) En un árbol binario enhebrado todas las hojas tienen 2 hebras.**

Falso. Por definición, en un árbol enhebrado cuando un nodo no tiene hijo derecho, se sustituye el valor nulo por su sucesor en inorder (hebra derecha); cuando no tiene hijo izquierdo, se sustituye el valor nulo por

su predecesor en inorder (hebra izquierda). De esta definición se deduce que el primer nodo del recorrido en inorder no tiene hebra izquierda y que el último nodo del recorrido en inorder no tiene hebra derecha; si estos nodos son hojas, como máximo tendrían 1 hebra cada uno.

## 5.5. Árbol AVL

### 5.5.1. Preguntas

**(P.49) La complejidad de las operaciones de equilibrado de los árboles AVL sugiere que deben utilizarse sólo si las inserciones son considerablemente más frecuentes que las búsquedas.**

Falso. La complejidad de las operaciones de equilibrado de los árboles AVL sugiere que deben utilizarse sólo si las búsquedas son más frecuentes que las inserciones o los borrados.

**(P.50) Un árbol AVL es un árbol binario de búsqueda en el que la diferencia de nodos entre el subárbol izquierdo y derecho es como máximo uno.**

Falso. Un árbol AVL es un árbol binario de búsqueda en el que se cumple que el factor de equilibrio (FE) de todos sus nodos es como máximo uno en valor absoluto. Definimos el factor de equilibrio (FE) de un nodo como la diferencia de altura entre el subárbol izquierdo y el subárbol derecho.

**(P.51) En un árbol AVL siempre que se inserte una etiqueta hay que realizar una rotación.**

Falso. Con dar un ejemplo que no cumple la condición ya podemos concluir con que la afirmación es falsa. En este caso si tenemos un árbol AVL con tres nodos de forma que el FE de cada nodo es 0. Si insertamos un nuevo nodo, el factor de equilibrio va a variar en 1, pasando a ser -1 o +1. Sigue cumpliendo la condición de árbol AVL.

**(P.52) El mínimo número de nodos que ha de tener un árbol binario de altura 4 para ser equilibrado respecto a la altura es 7.**

Verdadero. El mínimo número de nodos que ha de tener un árbol de altura  $h$  para ser equilibrado respecto a la altura se obtiene a partir de la fórmula de Fibonacci:

$$N_h = 1 + N_{h-1} + N_{h-2}$$

donde

$$N_0 = 0, N_1 = 1$$

$$\text{y } N_2 = 1 + N_1 + N_0 = 1 + 1 + 0 = 2.$$

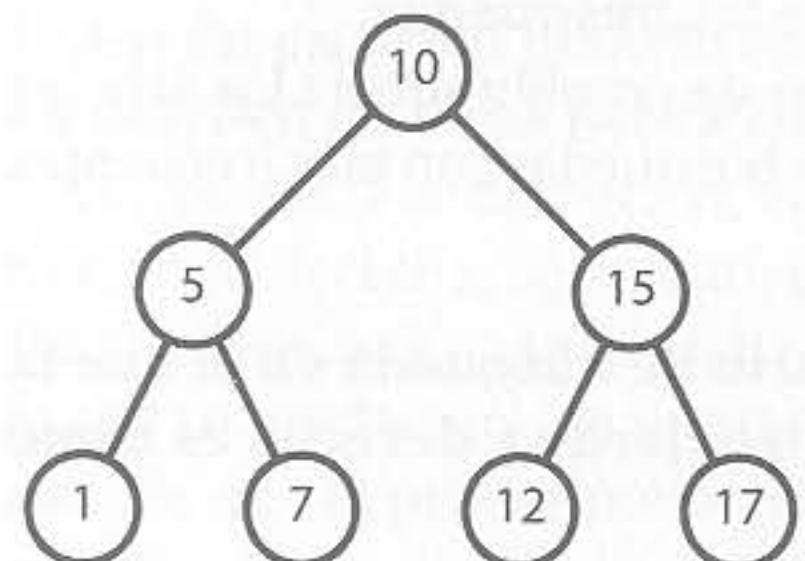
Por tanto, para  $h = 4$ :

$$N_4 = 1 + N_3 + N_2 = 1 + 1 + N_2 + N_1 + N_2 = 1 + 1 + 2 + 1 + 2 = 7$$

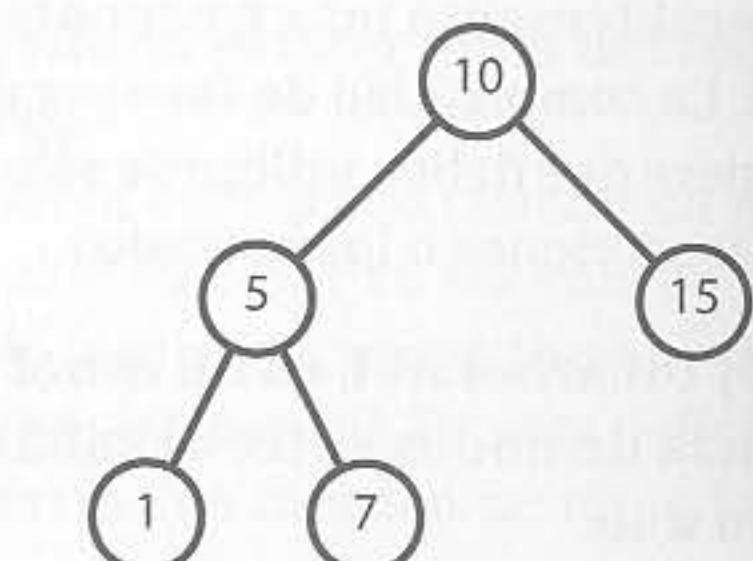
por lo que la afirmación es verdadera.

**(P.53) Un árbol completo es un árbol completamente equilibrado.**

Falso. Un árbol completo es un árbol que tiene todos los nodos numerados por niveles de izquierda a derecha y de arriba a abajo. Esta afirmación implica que no es un árbol lleno. Un árbol lleno es el único árbol que está completamente equilibrado.



ARBOL LLENO -> COMPLETAMENTE EQUILIBRADO



ARBOL COMPLETO -> NO COMPLETAMENTE EQUILIBRADO

**(P.54) Para realizar la inserción de un elemento en un árbol AVL que sea el máximo o el mínimo, se realizará como máximo una rotación doble.**

Falso. Sólo hemos de demostrar que existe un caso para el que no se cumple la afirmación. En este caso podemos plantear un ejemplo de un árbol AVL vacío al que le insertamos un elemento. Este elemento será el máximo y el mínimo y no hemos tenido que hacer ninguna rotación de ningún tipo.

**(P.55) Un árbol de Fibonacci siempre está equilibrado respecto a la altura.**

Verdadero. Por definición:

1. El árbol vacío es el árbol de Fibonacci de altura 0.
2. Un árbol con un único nodo es el árbol de Fibonacci de altura 1.

3. Si  $T_{h-1}$  y  $T_{h-2}$  son árboles de Fibonacci de alturas  $h-1$  y  $h-2$  respectivamente, entonces el árbol que resulta de enraizar un elemento con  $T_{h-1}$  (por la izquierda) y  $T_{h-2}$  (por la derecha) es el árbol de Fibonacci de altura  $h$ .
4. No hay otros árboles que sean de Fibonacci.

Según esta definición recursiva, todo árbol de Fibonacci siempre está equilibrado respecto a la altura, ya que para cada subárbol las diferencias entre las alturas del hijo de la izquierda y del hijo de la derecha siempre será 1.

**(P.56) Un árbol de Fibonacci es un árbol completamente equilibrado.**

Falso. Por definición, en un árbol de Fibonacci para cada subárbol las diferencias entre las alturas del hijo de la izquierda y del hijo de la derecha siempre será 1. Sin embargo, para que fuera un árbol completamente equilibrado las diferencias entre el número de nodos del hijo de la izquierda y del hijo de la derecha tendría que ser 1, característica que no se cumple.

**(P.57) El borrado en un árbol AVL puede requerir una rotación en todos los nodos del camino de búsqueda.**

Verdadero. El algoritmo de borrado de un árbol AVL nos dice que una vez borrado el nodo deseado, tenemos que comprobar el FE en todos los niveles para todos los nodos del camino de búsqueda y es posible que sea necesario realizar una rotación en cada nodo.

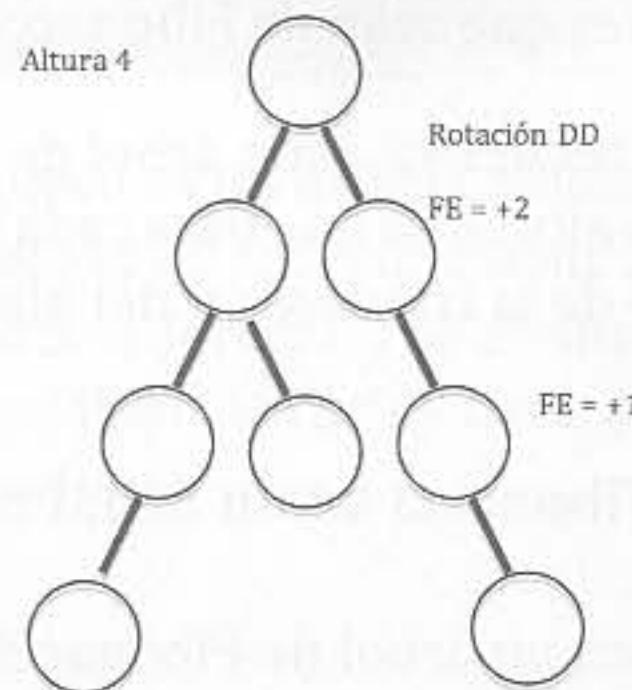
**(P.58) El número de nodos de un árbol de Fibonacci de altura  $h$  es el máximo número de nodos que puede tener un árbol binario de altura  $h$  para que sea AVL.**

Falso. Por definición, un árbol de Fibonacci de altura  $h$  tiene el mínimo número de nodos que tiene que tener un árbol AVL de altura  $h$ . Si sobre un árbol de Fibonacci de altura  $h$  se quita 1 nodo cualquiera (respetando la altura  $h$ ) el árbol resultante de altura  $h$  ya no es AVL.

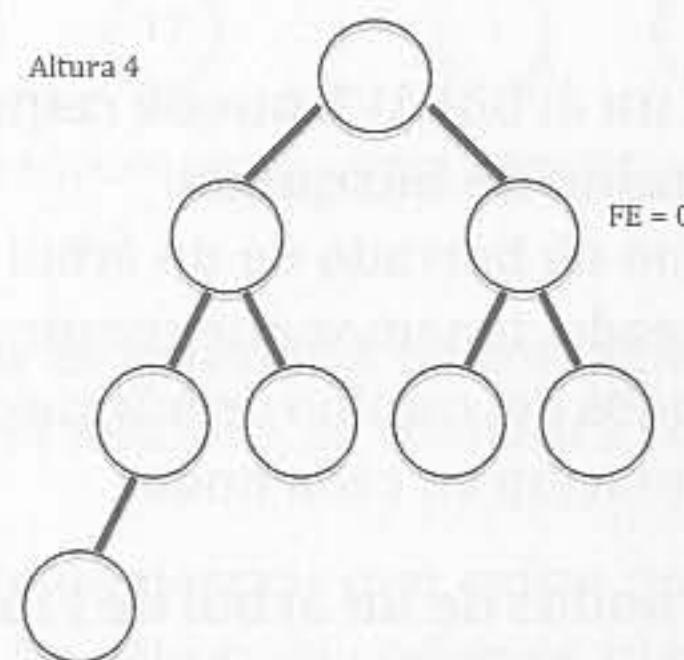
**(P.59) En el borrado del AVL, la altura del árbol decrece siempre tras realizar una rotación simple.**

Falso. Solamente decrece si la rama en la que se está realizando la rotación es la rama que determina la altura. Para ello, pongamos un ejem-

lo que no cumple la afirmación. Tenemos un árbol AVL de altura 4. Este árbol tiene un nodo que no cumple las condiciones de FE. El FE es +2 y el hijo derecha FE +1. Estas condiciones implican una rotación Derecha-Derecha.



Como podemos comprobar, después de la rotación, el factor de equilibrio se restaura, pero la altura del árbol no ha variado.



**(P.60) El número mínimo de nodos que tiene un árbol AVL de altura 5 es 12.**

Verdadero. La fórmula de Fibonacci establece:

$$N_h = 1 + N_{h-1} + N_{h-2}$$

donde

$$N_0 = 0, N_1 = 1$$

$$\text{y } N_2 = 1 + N_1 + N_0 = 1 + 1 + 0 = 2.$$

Por tanto, para  $h = 5$ :

$$N_5 = 1 + N_4 + N_3 = 1 + 1 + N_3 + N_2 + 1 + N_2 + N_1 = 1 + 1 + 1 + N_2 + N_1 + N_2 + 1 + N_2 + N_1 = 12$$

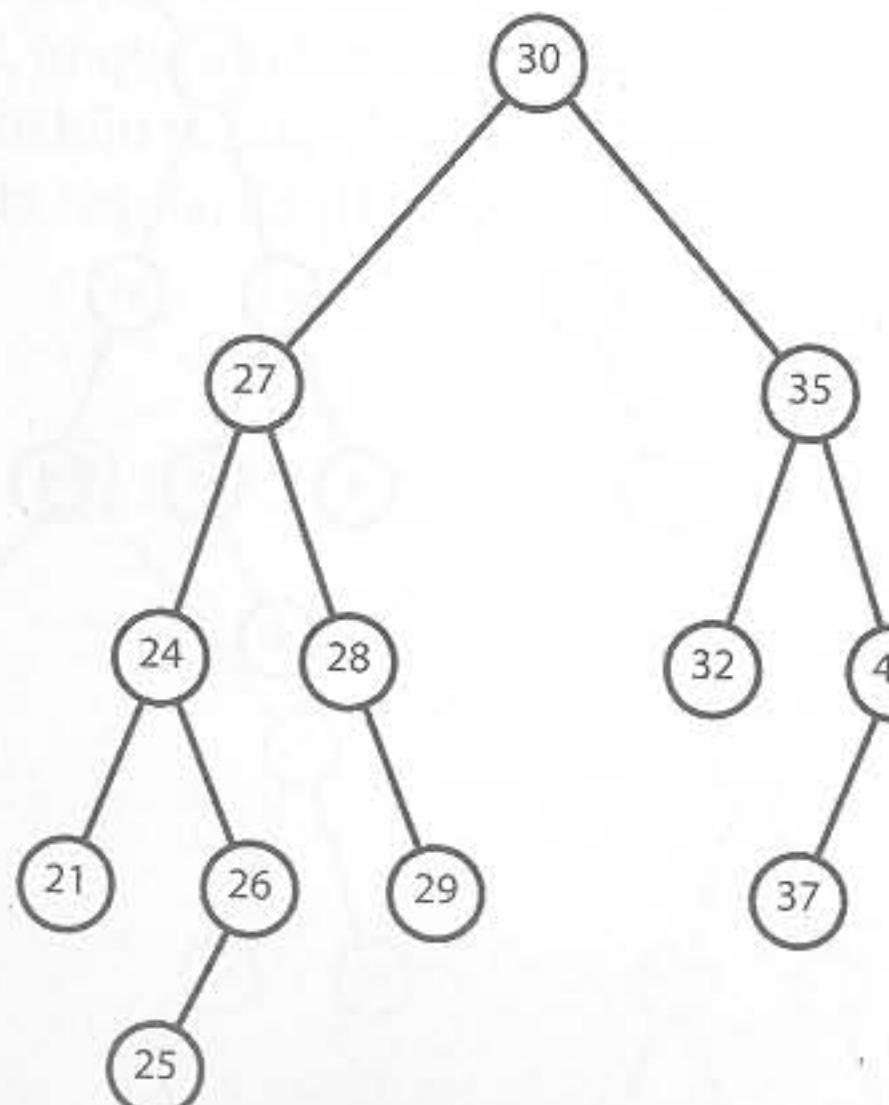
por lo que la afirmación es verdadera.

**(P.61) Los árboles AVL son aquellos en los que el número de elementos en los subárboles izquierdo y derecho difieren como mucho en 1.**

Falso. Un árbol AVL es un árbol binario de búsqueda en el que se cumple que la diferencia de altura entre el subárbol izquierdo y el subárbol de cualquier nodo del árbol es como máximo uno.

### 5.5.2. Ejercicios

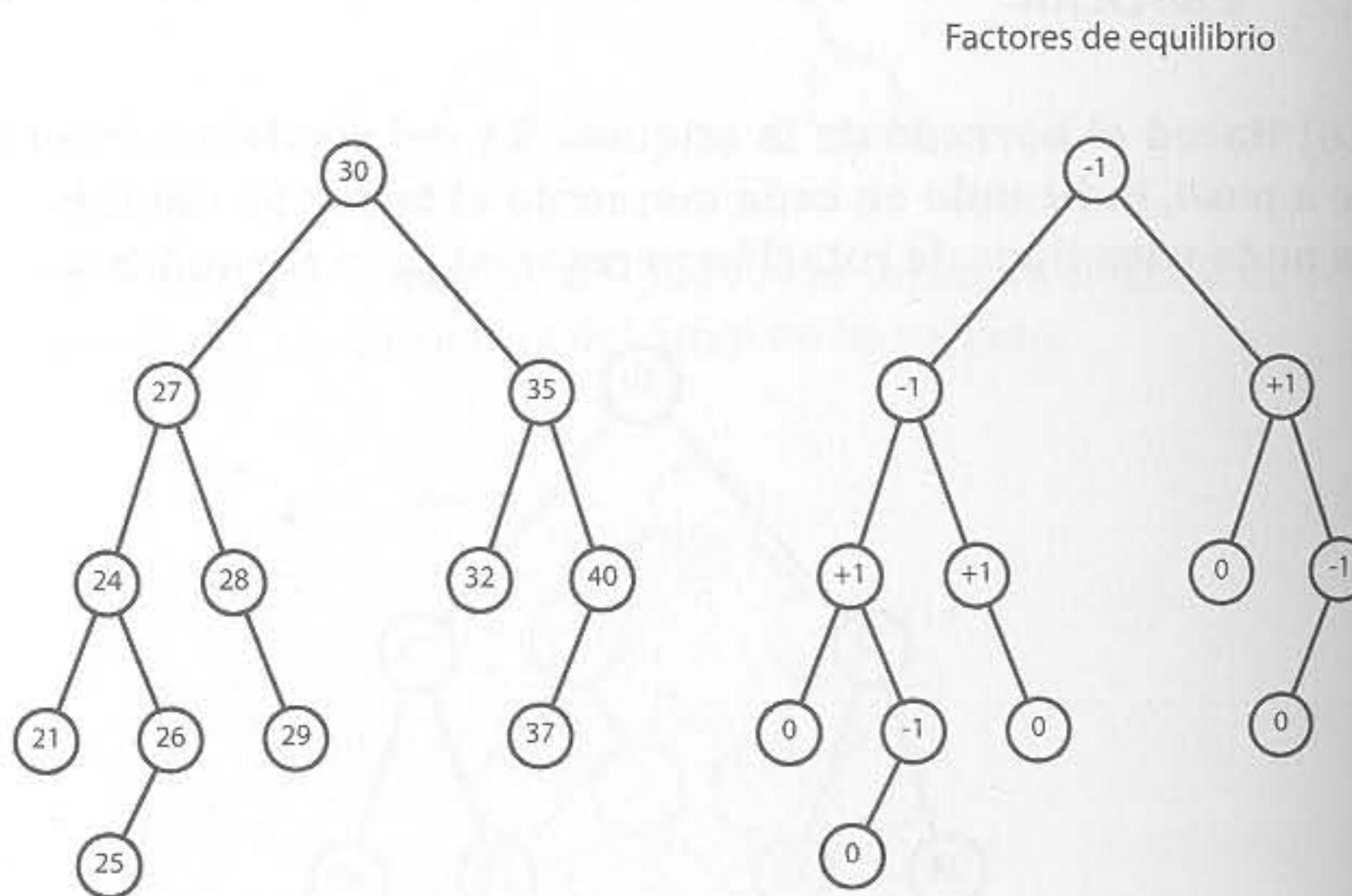
**(E.26) Haced el borrado de la etiqueta 32 del siguiente árbol AVL paso a paso, indicando en cada momento el factor de equilibrio de cada nodo y los tipos de rotación necesarios para reequilibrar.**



#### SOLUCIÓN:

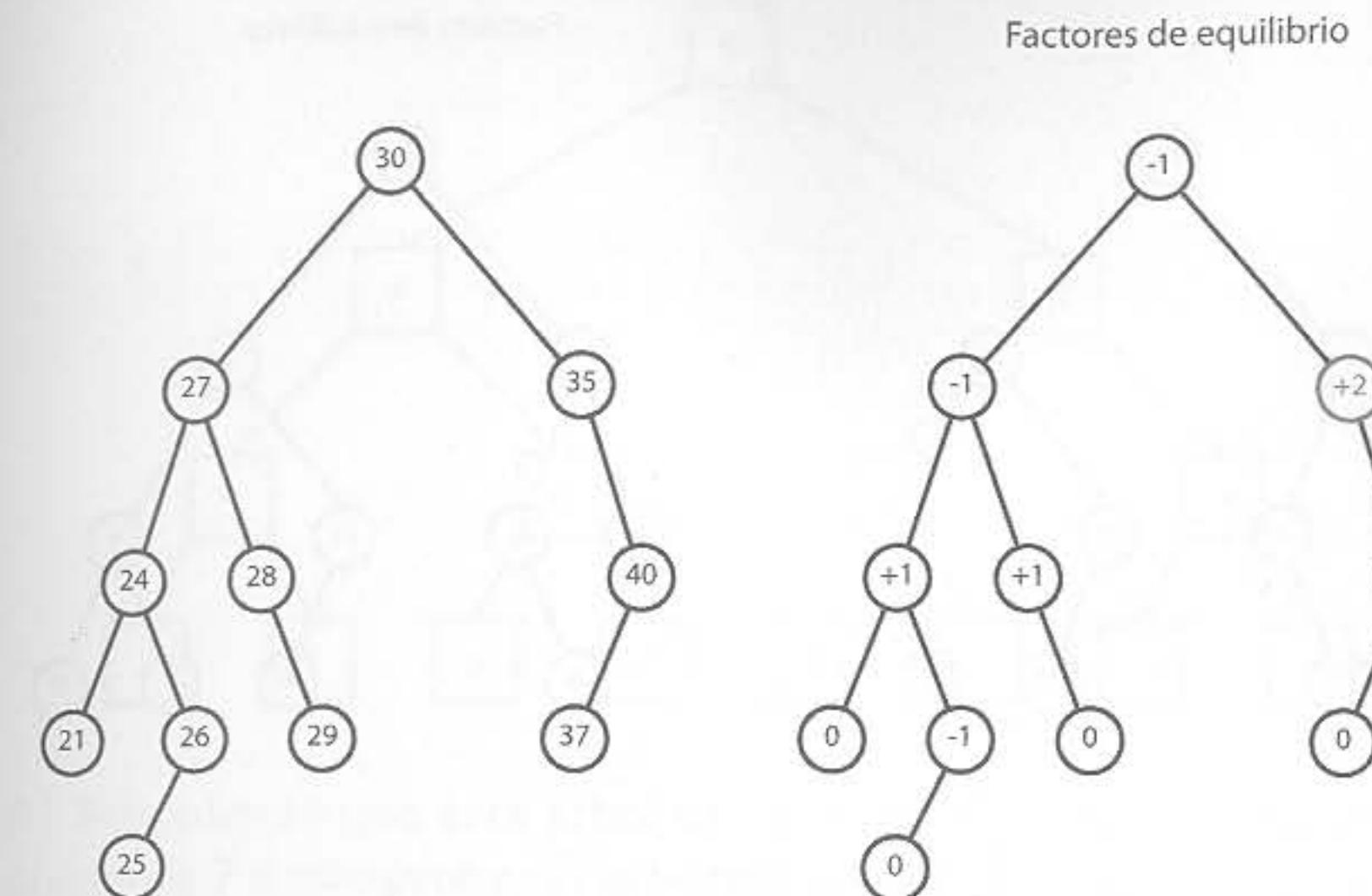
El factor de equilibrio  $FE(T)$  de un nodo  $T$  en un árbol binario se define como la altura del subárbol derecho menos la altura del subárbol izquierdo. Para cualquier nodo  $T$  en un árbol AVL, se cumple que  $FE(T) = \{-1, 0, 1\}$ .

En la siguiente figura hemos representado a la derecha del árbol AVL los factores de equilibrio de cada uno de los nodos. Como podemos observar, los factores de equilibrio de todos los nodos cumplen la condición de árbol AVL.

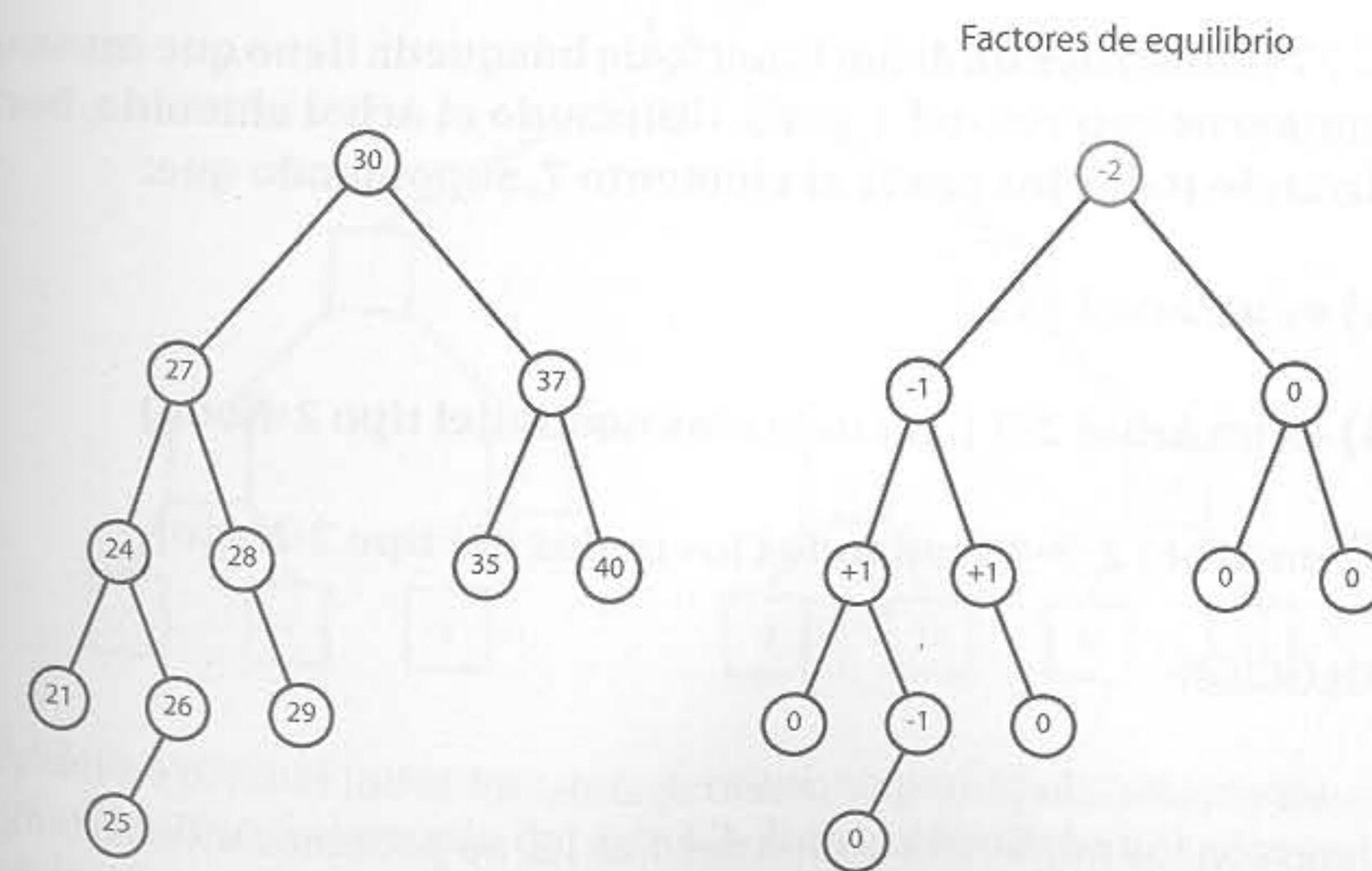


Factores de equilibrio

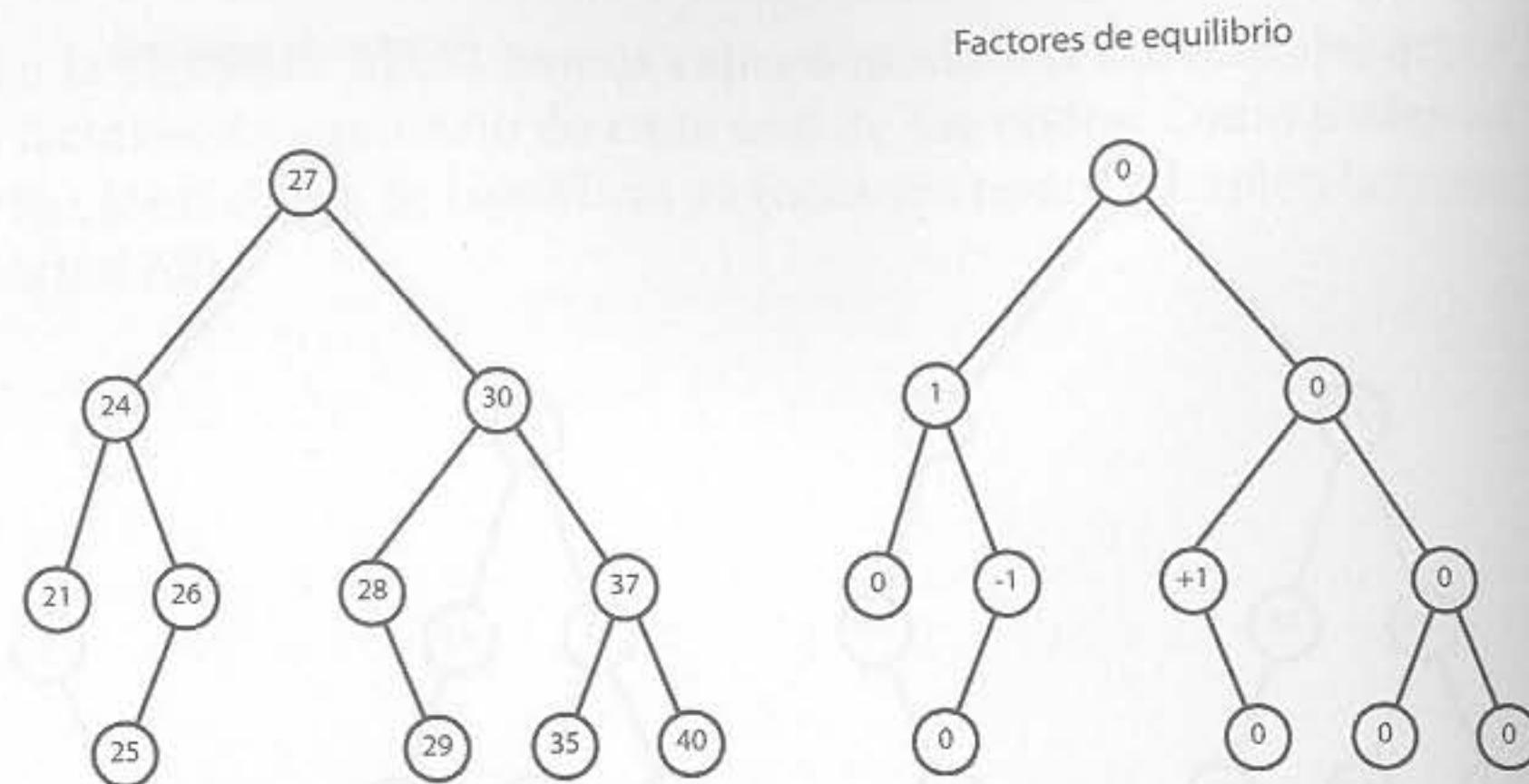
Tras borrar la etiqueta 32, podemos ver que el nodo con la etiqueta 35 queda con un factor de equilibrio +2, lo que impide que el árbol sea AVL. El factor de equilibrio de su hijo derecho es -1, por lo que tenemos que hacer una rotación derecha izquierda (DI) y la altura del subárbol decrece.



Tras la rotación DI, como la altura del subárbol ha decrecido, el factor de equilibrio del padre (30) que es el nodo raíz se ha modificado y ha pasado de -1 a -2, lo que vuelve a impedir que sea un árbol AVL. El factor de equilibrio de su hijo izquierdo es -1, por lo que tenemos que hacer una rotación izquierda izquierda (II) y la altura del árbol decrece.



Tras la rotación podemos observar que el árbol resultante es AVL. La altura del árbol ha decrecido, ya que ha pasado de 5 a 4.



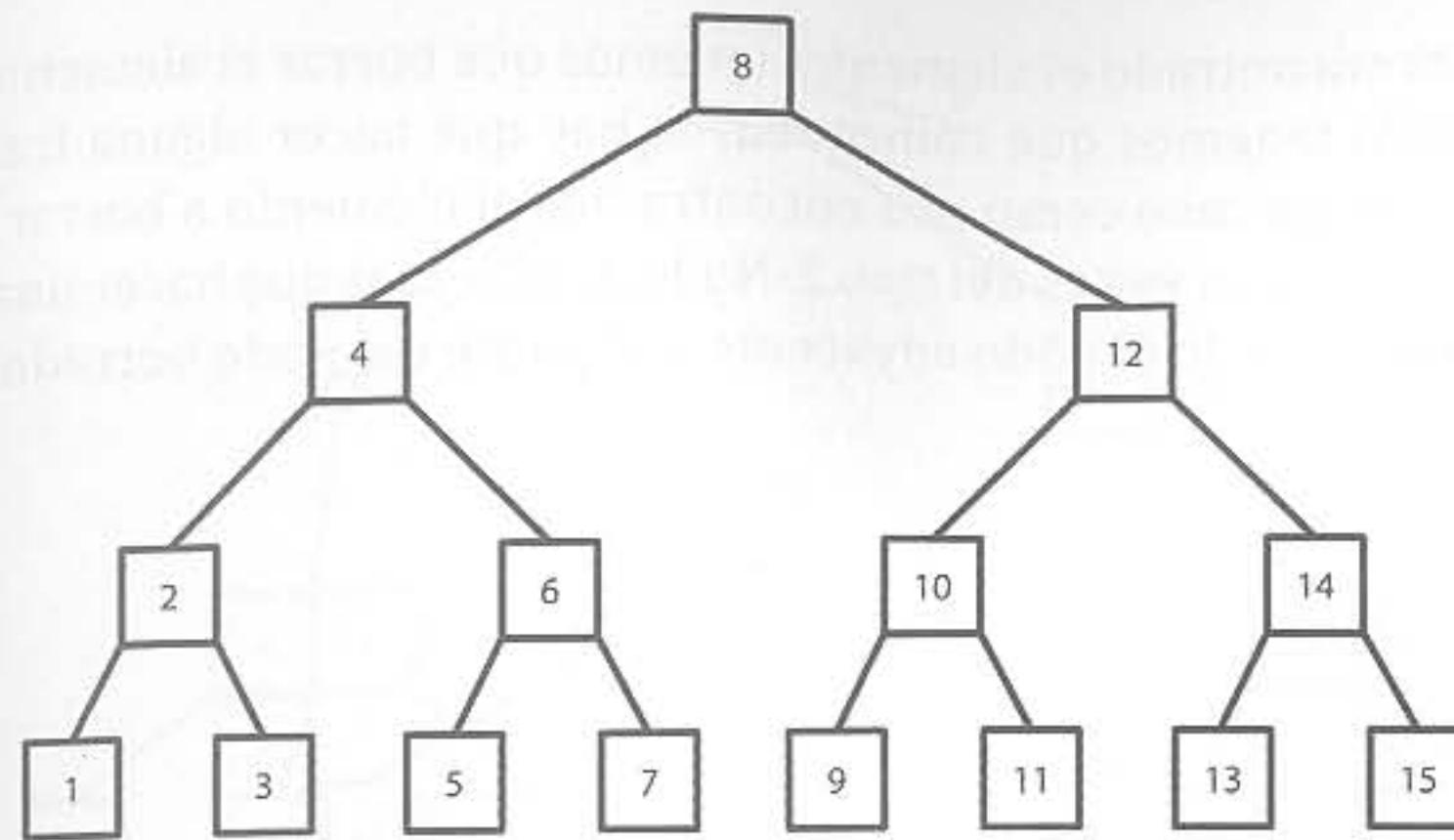
En total han sido necesarias dos rotaciones, una derecha izquierda (DI) y una izquierda izquierda (II).

**(E.27)** Construye un árbol binario de búsqueda lleno que contenga los números enteros del 1 al 15. Utilizando el árbol obtenido, borra realizando todos los pasos el elemento 7, suponiendo que:

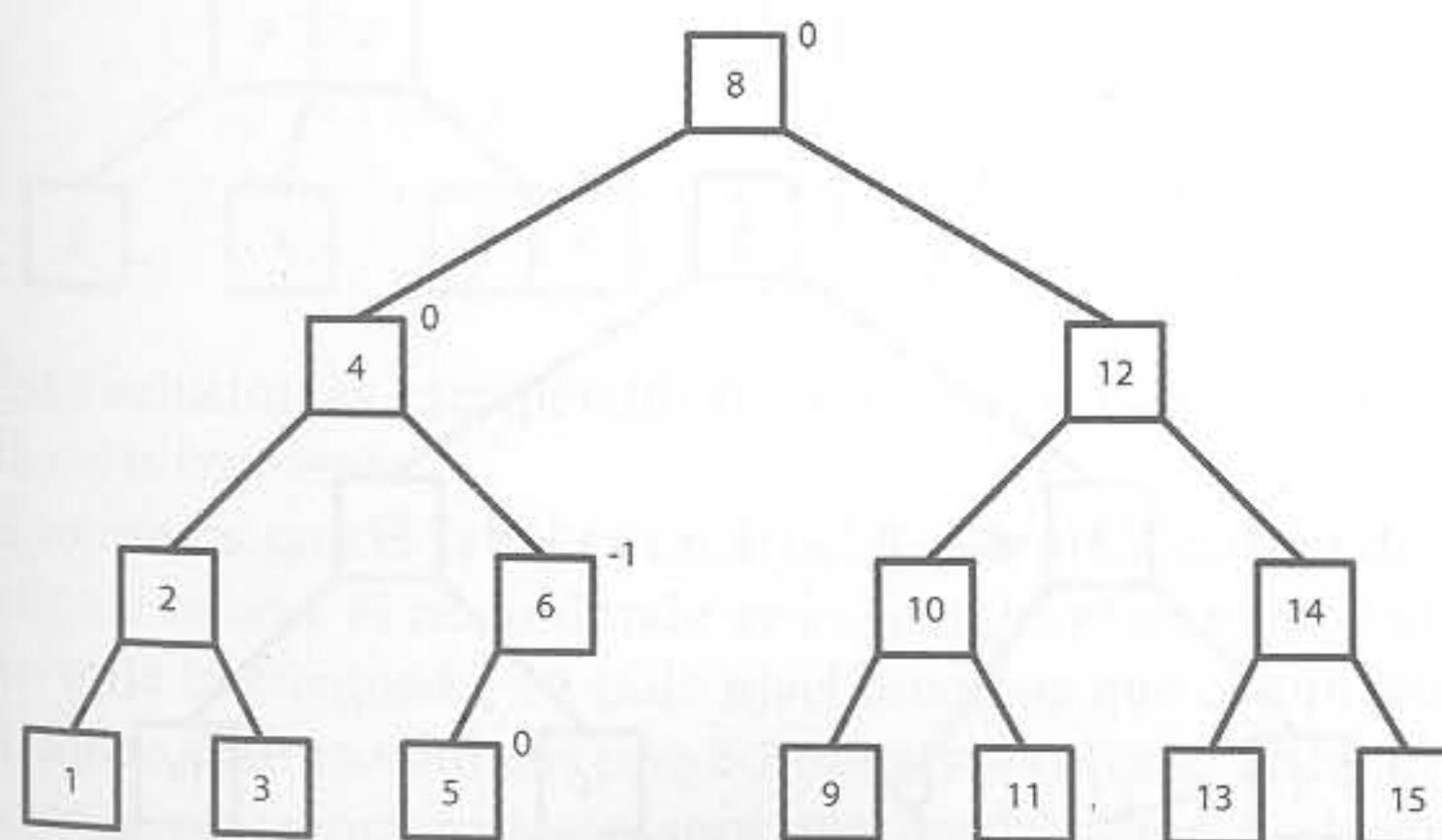
- A) es un árbol AVL
- B) es un árbol 2-3 (con todos los nodos del tipo 2-Nodo)
- C) un árbol 2-3-4 (con todos los nodos del tipo 2-Nodo)

SOLUCIÓN:

Como el ejercicio pide que construyamos un árbol binario de búsqueda lleno con los números enteros del 1 al 15, no podemos utilizar ningún algoritmo conocido, sino construir el árbol a mano de forma que los elementos que coloquemos en el árbol no hagan que el árbol incumpla la condición de árbol binario de búsqueda.

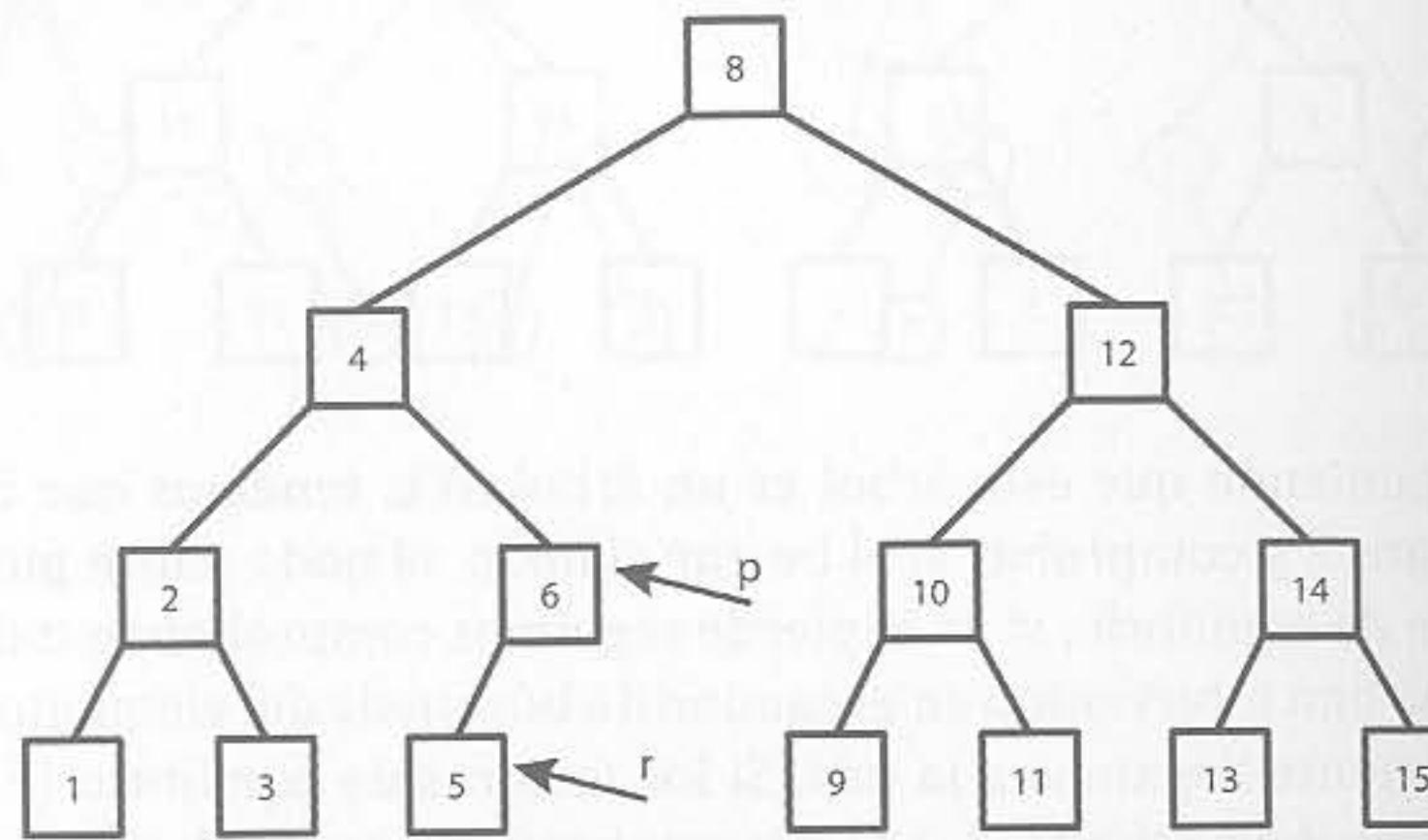


A) Suponiendo que este árbol es un árbol AVL, tenemos que buscar el elemento 7 y comprobar si al borrar el nodo, el nodo padre pierde la condición de equilibrio, si no lo pierde seguimos comprobando todos los nodos que han intervenido en el camino de búsqueda del elemento hasta que finalmente llegamos a la raíz. Si los factores de equilibrio (F.E.) siguen dentro de los límites (-1, 0, 1), entonces no hay que hacer ninguna transformación y habremos acabado el borrado.

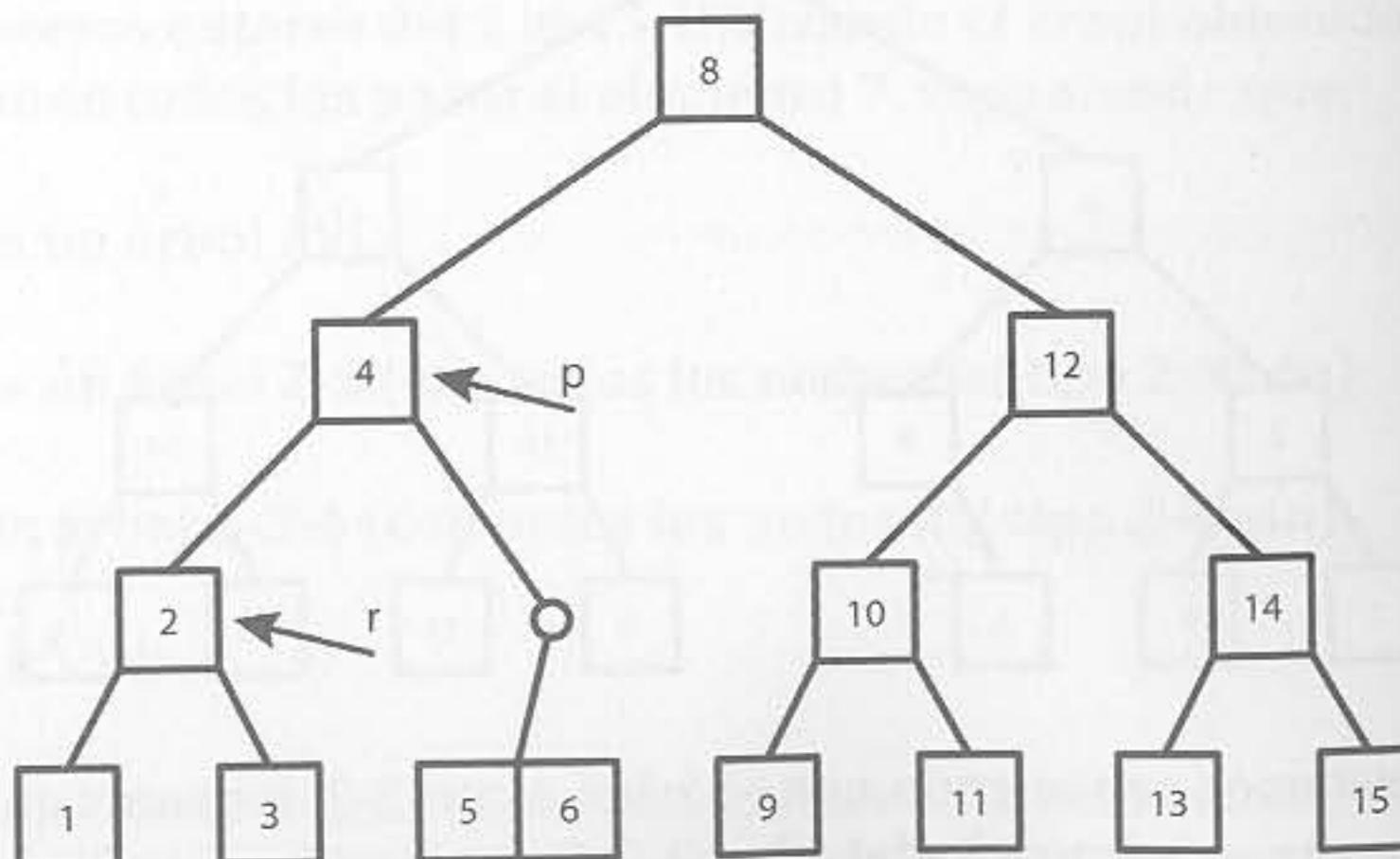


B) Si estamos suponiendo que el árbol es un 2-3, tenemos que aplicar el algoritmo de borrado del árbol 2-3. Este algoritmo nos dice que en un primer proceso tenemos que buscar el nodo donde se encuentra el elemento a borrar. Esto se hace utilizando el algoritmo de búsqueda en árboles multicamino de búsqueda.

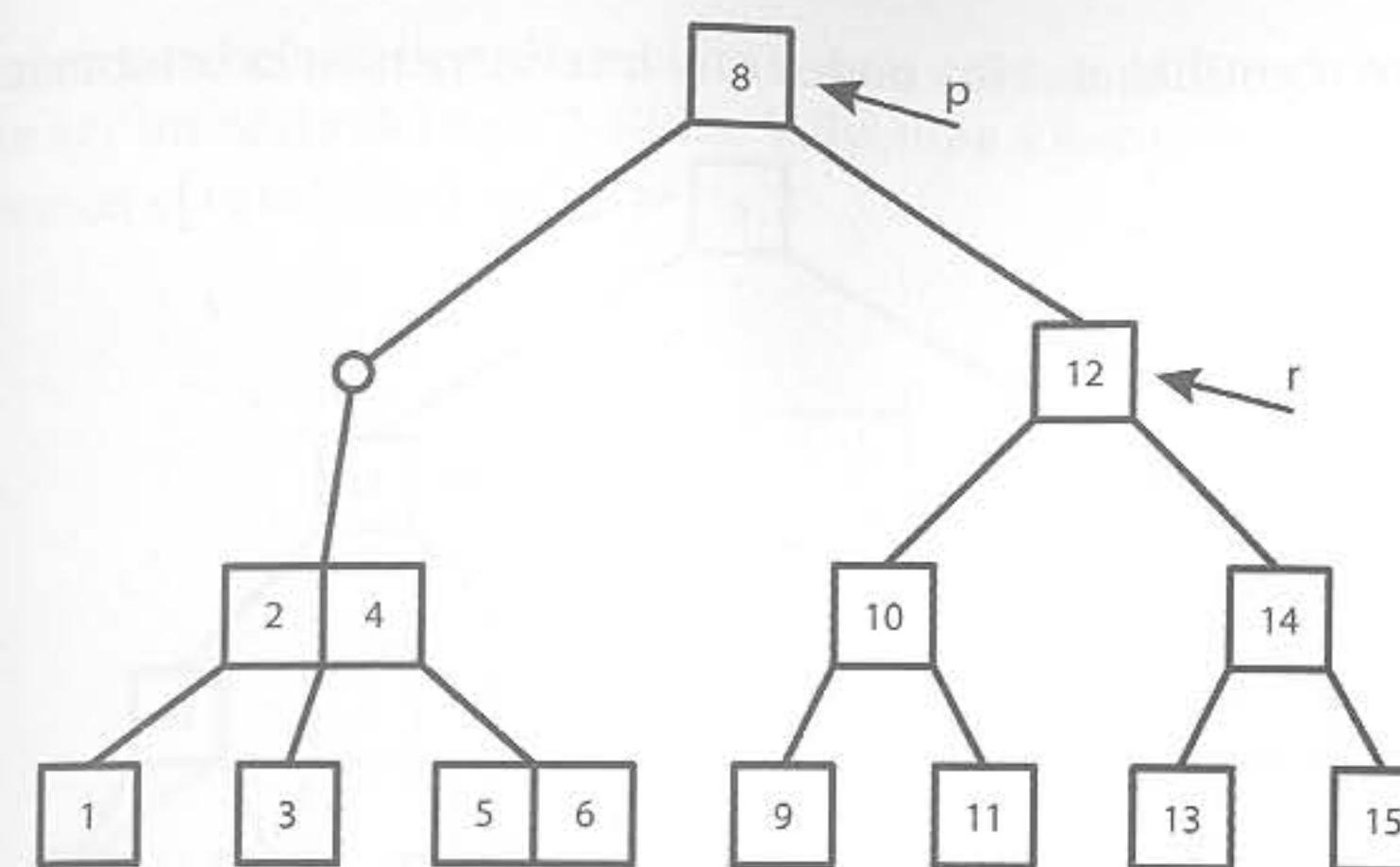
Una vez encontrado el elemento, tenemos que borrar el elemento. Una vez borrado tenemos que comprobar si hay que hacer alguna transformación. En este caso como nos encontramos el elemento a borrar en un nodo hoja que a su vez es del tipo 2-Nodo, tendremos que hacer una combinación utilizando el nodo adyacente y el padre del nodo borrado.



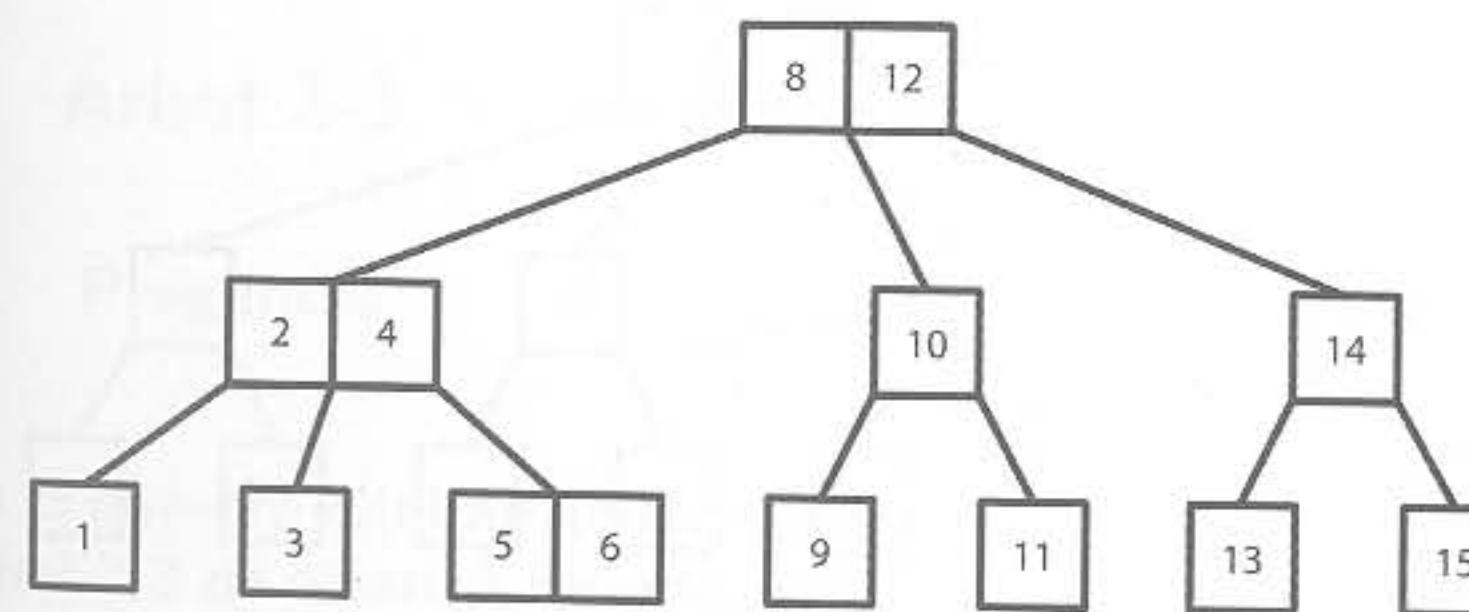
Una vez borrado el elemento, hacemos la combinación correspondiente.



Como el padre se ha quedado con un nodo vacío, volvemos a hacer otra combinación.



Volvemos a tener la misma situación anterior, pero en un nivel superior. Volvemos a hacer la transformación correspondiente y finalmente el árbol reduce en uno su altura.

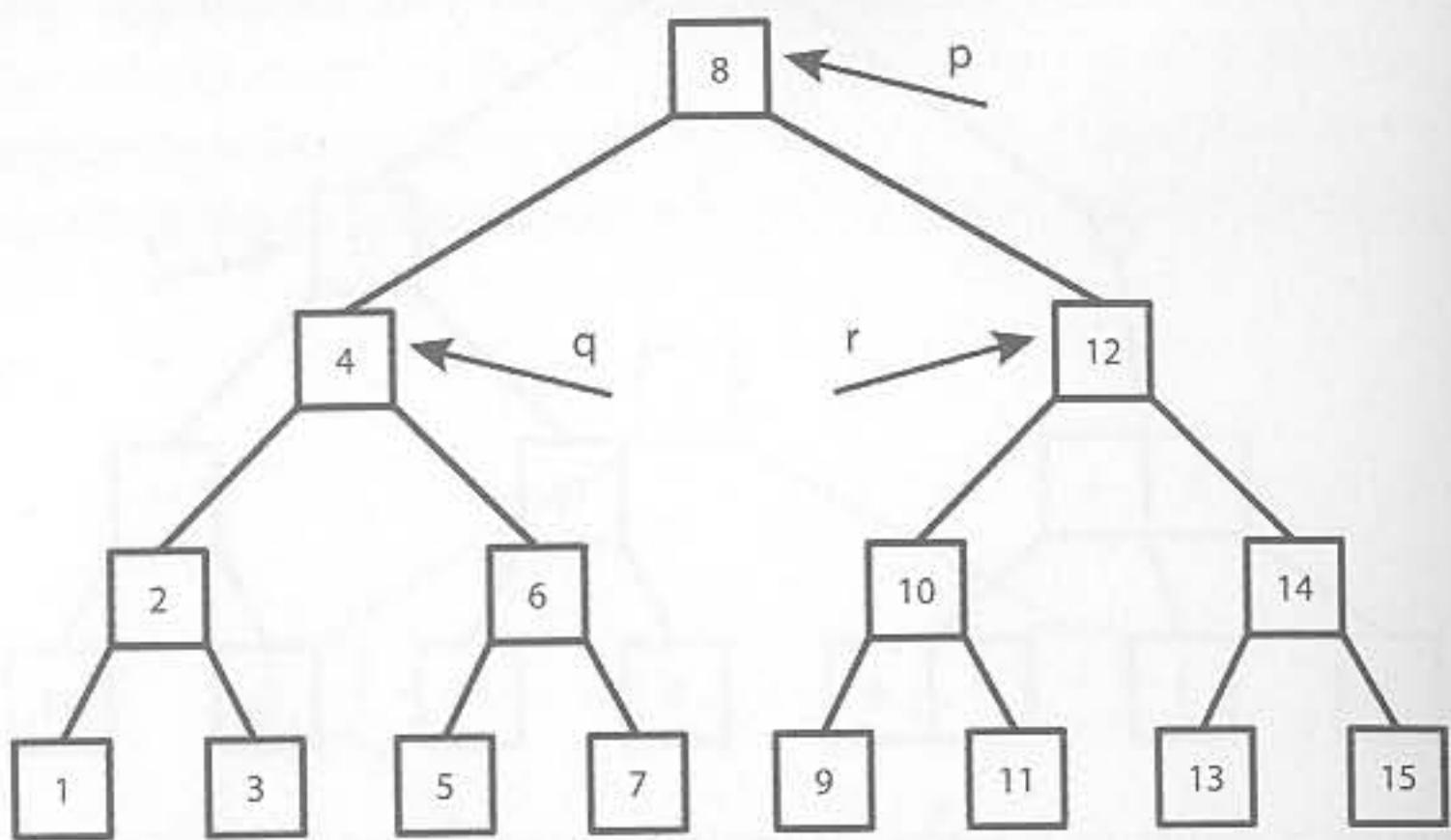


C) Esta solución es escogiendo como criterio preferencial al nodo adyacente de la izquierda.

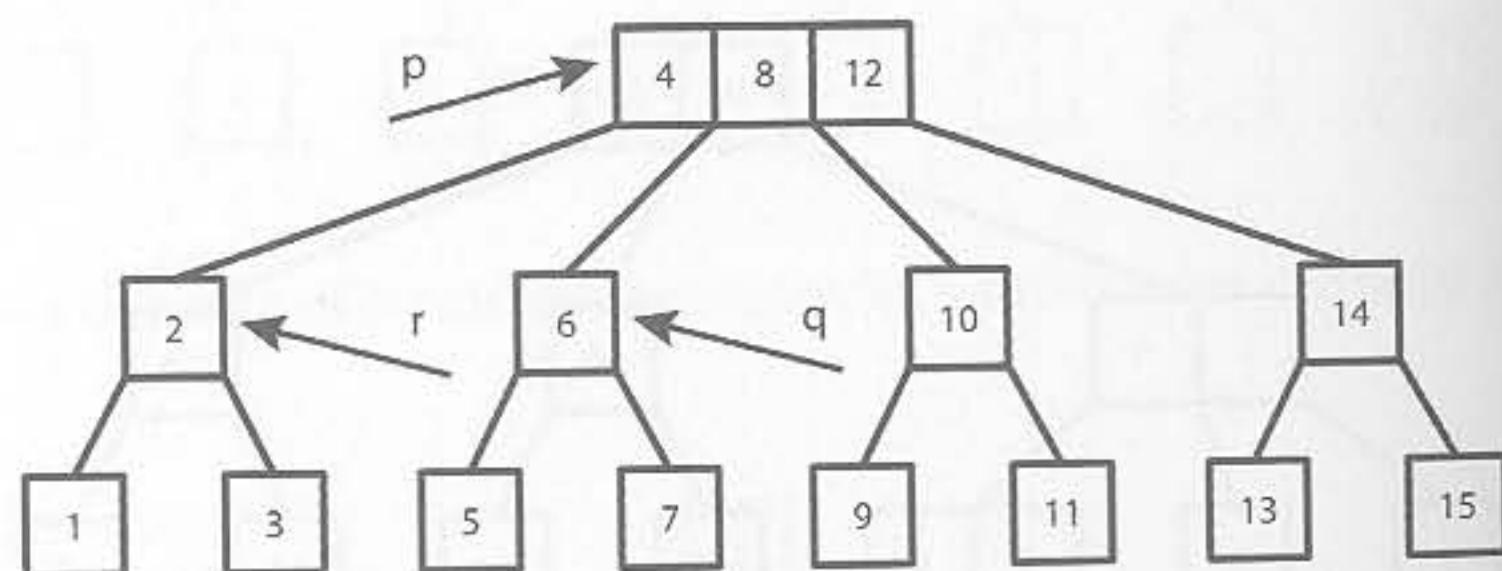
Si suponemos que el árbol es un árbol 2-3-4, el algoritmo de borrado nos obliga a buscar el nodo donde se encuentra el elemento a borrar y durante toda la búsqueda, en cada nivel tenemos que comprobar si nos encontramos ante un nodo del tipo 2-Nodo y si es el caso, hacer una transformación. Esta transformación será una combinación si el nodo adyacente es un nodo 2-Nodo y será una rotación si el nodo adyacente es un 3-Nodo o 4-Nodo.

En primer lugar nos encontramos con la raíz, que es un 2-Nodo, y comprobamos que los dos hijos el izquierdo y el derecho son también 2-Nodo. Este es un caso especial en el que hay que hacer una combinación y convertir esos 3 nodos del tipo 2-Nodo en un 4-Nodo.

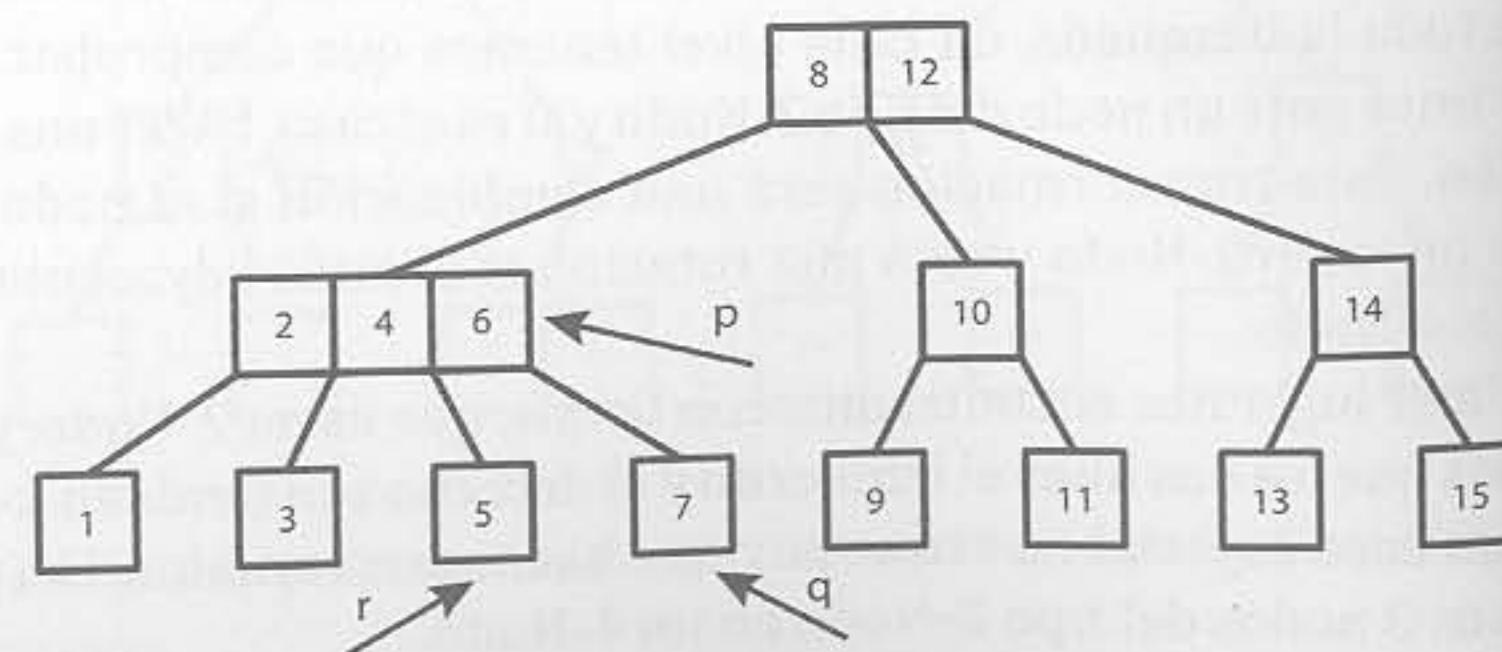
Primero identificamos los nodos que intervienen en la combinación



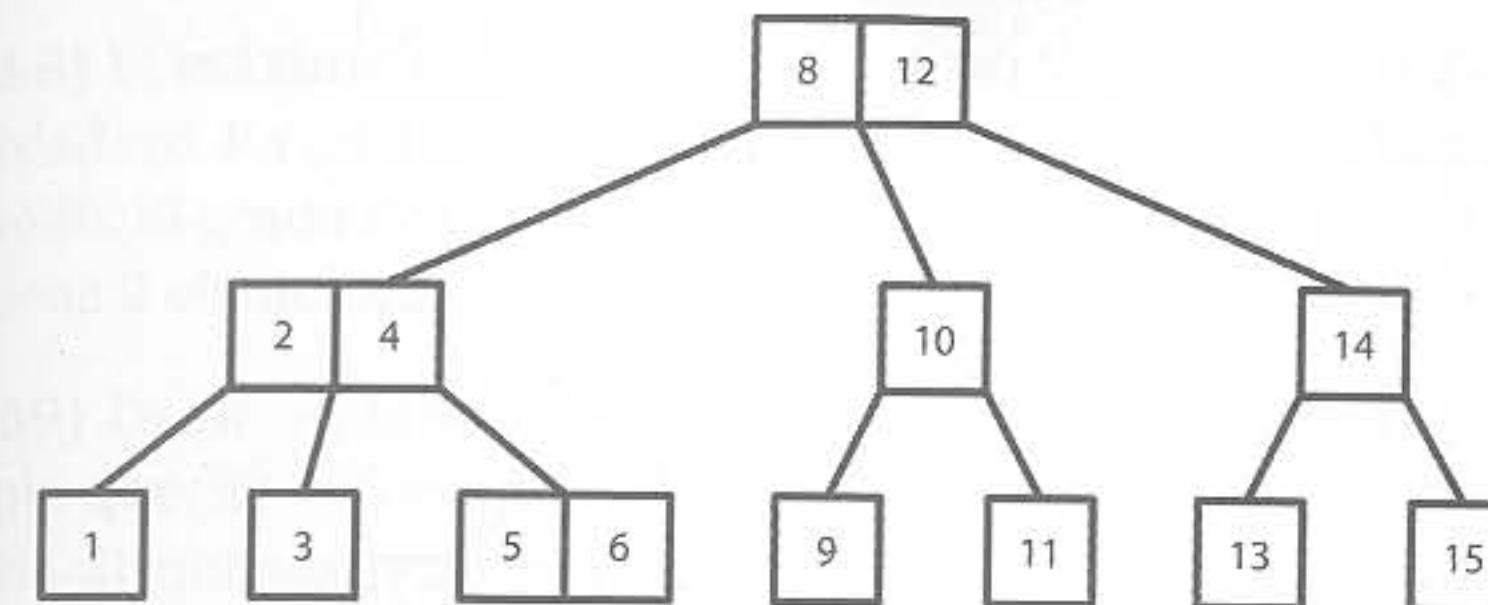
y acto seguido aplicamos la combinación.



Seguidamente buscamos el camino a seguir aplicando el algoritmo de árbol multicamino de búsqueda. El siguiente nodo también es un 2-Nodo. Como el nodo adyacente también es un 2-Nodo, tenemos que hacer una nueva combinación.



Finalmente hemos llegado a la hoja que queremos borrar y también resulta ser un nodo del tipo 2-Nodo. Volvemos a hacer la combinación y obtenemos el resultado final.



## 5.6. Árbol 2-3

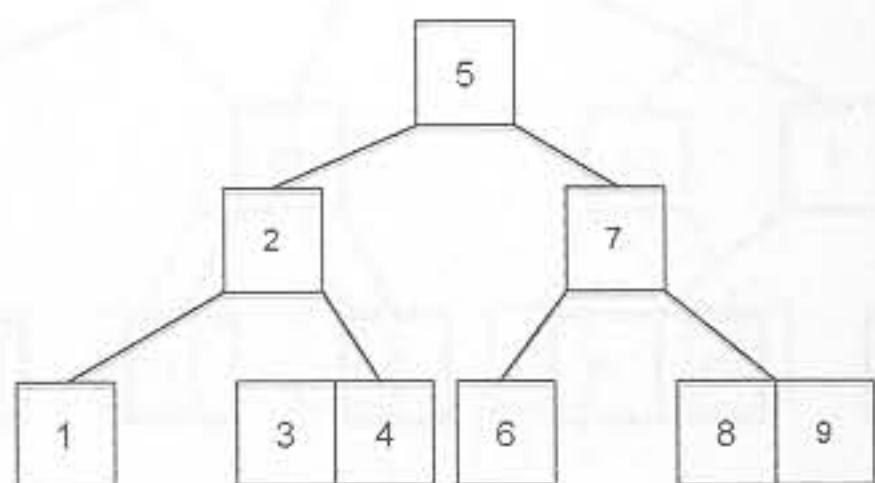
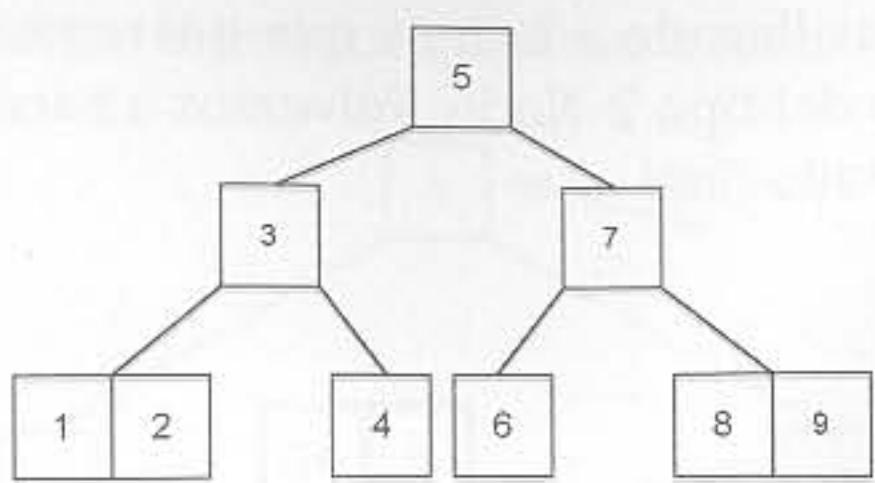
### 5.6.1. Preguntas

(P.62) El número mínimo de elementos que se puede almacenar en un árbol 2-3 de altura  $h$  coincide con el número de elementos que hay en un árbol binario completo de altura  $h$ .

Falso. En un árbol binario completo tenemos los nodos numerados por niveles de izquierda a derecha y si el árbol tiene  $n$  nodos no puede faltar ninguno. Esto no garantiza que en todos los niveles estén todos los nodos, por lo que es posible que todas las hojas no estén en el mismo nivel. Esta es una condición que tiene que cumplir el árbol 2-3

(P.63) Existe un único árbol 2-3 de altura 3 que representa a las etiquetas del 1 al 9.

Falso. Se pueden construir diferentes árboles 2-3 de altura 3 que representen las etiquetas del 1 al 9 dependiendo del orden de inserción de los elementos. Si construimos un ejemplo que no cumpla con la condición podremos demostrar que la afirmación es falsa. Por ejemplo:



**(P.64) Las operaciones de inserción y borrado de un árbol 2-3 son las mismas que las de un árbol B de grado 3.**

Verdadero. Son las mismas debido a que el árbol 2-3 es un árbol B con  $m = 3$ .

**(P.65) En un árbol 2-3 la altura del árbol sólo aumenta cuando todas las hojas del árbol son de grado tres.**

Falso. En un árbol 2-3 la altura del árbol sólo aumenta cuando realizamos una inserción y siempre que todos los nodos del camino de búsqueda del nodo hoja son de grado tres.

**(P.66) La operación de rotación en un árbol 2-3 se da cuando el hermano del nodo donde se efectúa es del tipo 3-Nodo.**

Verdadero. Esto ocurre cuando el nodo adyacente (hermano) tiene suficientes elementos (*items*) como para prestarle uno al nodo en el que se realiza el borrado. Este préstamo se realiza mediante la rotación de elementos.

**(P.67) Dado un árbol 2-3 con  $n$  elementos (*items*) con todos sus nodos del tipo 3-Nodo, la complejidad de la operación de búsqueda de un elemento es  $O(\log_3(n + 1))$ .**

Verdadero. El número de nodos es un árbol 2-3 de altura  $h$  en el que todos sus nodos son de tipo 3-Nodo, es  $n = 3^h - 1$ . La operación de bús-

queda se realiza buscando en la raíz y desplazándonos hacia las hojas. En el peor de los casos, la búsqueda nos llevará hasta las hojas, es por ello que la complejidad estará en función de la altura del árbol. Tenemos que despejar la altura ( $h$ ) de la ecuación anterior y tendremos la complejidad en el peor caso:  $O(\log_3(n + 1))$ .

**(P.68) El máximo grado de cualquier nodo de un árbol 2-3 es 3.**

Verdadero. En un árbol 2-3 cada nodo puede ser de dos tipos: 2-Nodo o 3-Nodo. El grado de un nodo de tipo 3-Nodo es 3, es decir, es un nodo que tiene 2 elementos (*items*) y 3 hijos.

**(P.69) Dado un árbol 2-3 de altura  $h$  con  $n$  elementos (*items*) se cumple que:  $3^h - 1 < n < 2^h - 1$ .**

Falso. El número de nodos en un árbol 2-3 se encuentra entre los límites  $2^h - 1$  y  $3^h - 1$ . Por tanto:

$$2^h - 1 \leq n \leq 3^h - 1$$

**(P.70) El número mínimo de elementos que se pueden almacenar en un árbol 2-3 de altura  $h$  es  $2^h - 1$ .**

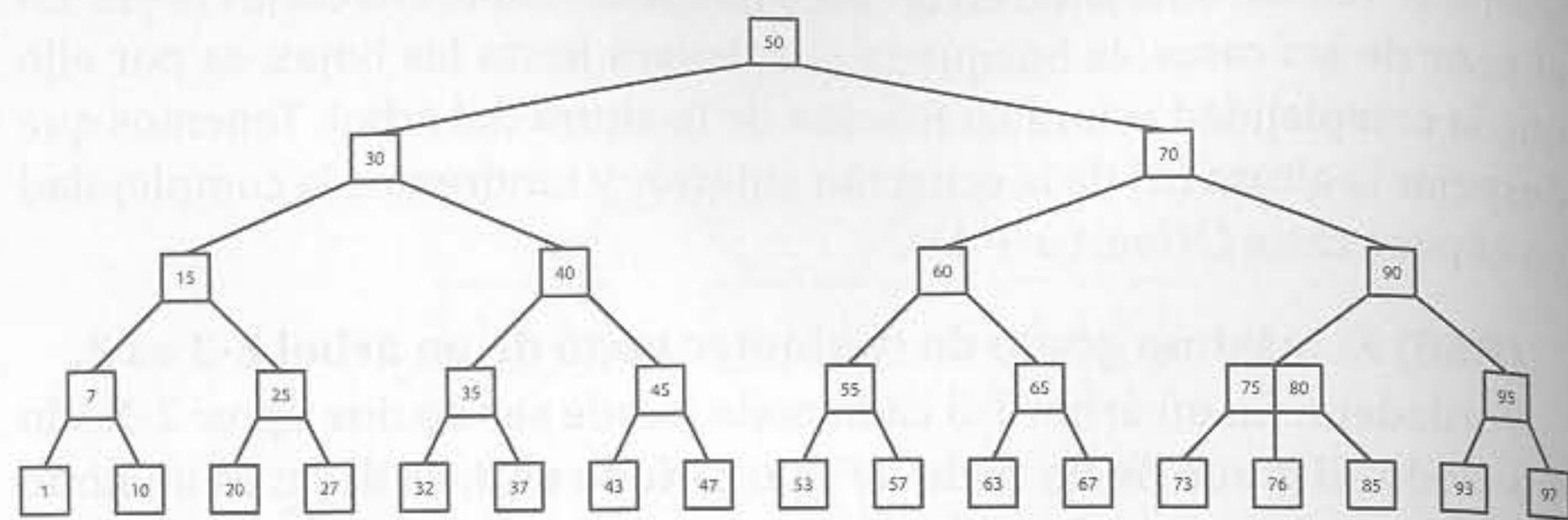
Verdadero. El numero de elementos que puede almacenar un árbol 2-3 es un valor comprendido entre  $2^h - 1$  y  $3^h - 1$ . Estos son los casos extremos en que el árbol tiene sólo 2-Nodos ó 3-Nodos. Un árbol 2-3 con algunos 2-Nodos y 3-Nodos tiene un número de elementos entre los dos límites antes indicados. Pero si un árbol 2-3 tiene todos sus nodos de tipo 2-Nodo, entonces estaríamos ante un árbol binario lleno. El numero de elementos que tiene un árbol binario lleno de altura  $h$  es  $2^h - 1$ .

## 5.6.2. Ejercicios

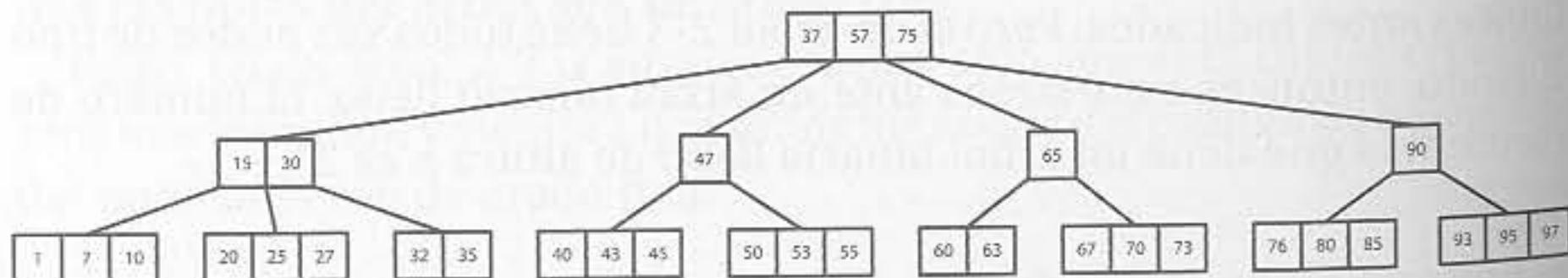
**(E.28) Dado el siguiente árbol 2-3:**

A) ¿Qué altura mínima y máxima debería tener el árbol 2-3-4 que contuviese las mismas claves? Contestar de forma razonada. Mostrar uno de los posibles árboles 2-3-4 para la altura mínima y otro para la máxima.

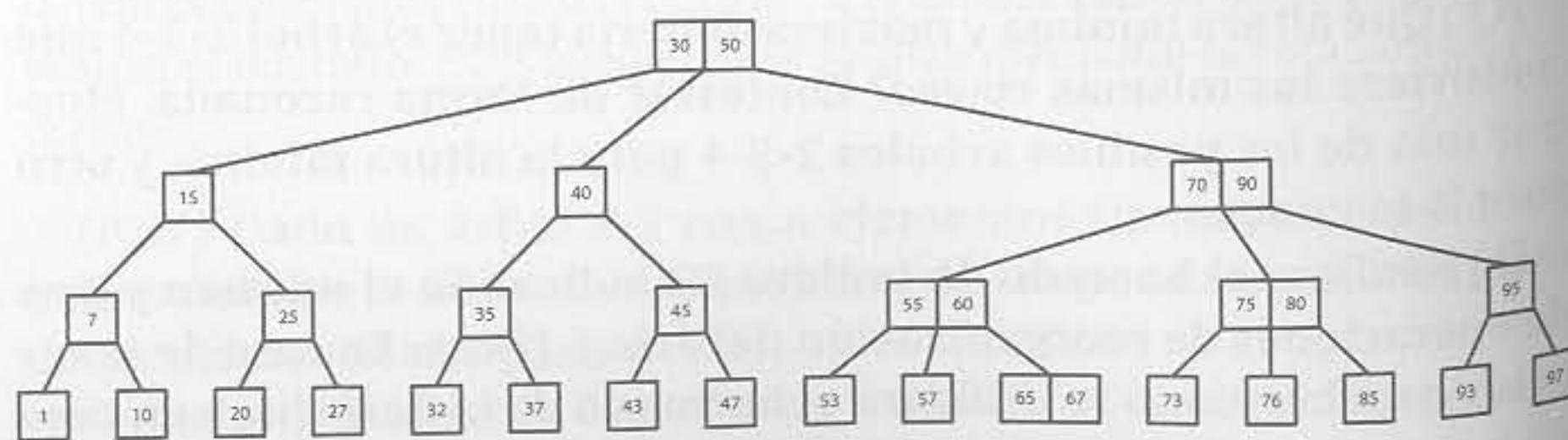
B) Realizar el borrado de la clave 63 indicando el número y tipo de operaciones de reorganización del árbol. (Nota: En caso de tener más de un hermano se utilizará el hermano de la derecha, y en caso de borrar una clave en un nodo interior se sustituirá por el menor del subárbol derecho).

**SOLUCIÓN:**

A) Dado que la altura máxima de un árbol 2-3-4 con  $n$  claves sería:  $\log_2(n + 1)$ , para  $n = 32$  tendríamos como máximo una altura 5 con 32 2-Nodos. Para  $n = 64$  tendríamos altura 6 con 64 2-Nodos. Dado que sólo hay 33 claves y el árbol ha de estar equilibrado, entonces la altura máxima debería ser 5. Un ejemplo sería el mismo árbol del enunciado, salvo que cada nodo podría almacenar 3 claves como máximo. Dado que la altura mínima de un árbol 2-3-4 con  $n$  claves sería:  $\log_4(n + 1)$ , para  $n = 15$  tendríamos como mínimo una altura 2 con 15 4-Nodos. Para  $n = 63$  tendríamos altura 3 con 21 4-Nodos. Dado que sólo hay 33 claves y el árbol ha de estar equilibrado, entonces la altura mínima debería ser 3. Un ejemplo sería el siguiente:



B) 4 combinaciones y borrado de la raíz (con lo que se reduce la altura del árbol)

**5.7. Árbol 2-3-4****5.7.1. Preguntas**

(P.71) En un árbol 2-3-4 con  $n$  elementos, la altura de dicho árbol se encuentra entre  $\log_4(n + 1)$  y  $\log_2(n + 1)$ .

Verdadero. Si suponemos que todos los nodos del árbol son 4-Nodo, para una altura  $h$  el número de nodos que posee el árbol es  $4^h - 1$ , ya que cada nivel posee el número de nodos del nivel anterior multiplicado por 4. Por otro lado, si suponemos que todos los nodos del árbol son 2-Nodo, para una altura  $h$  el número de nodos que posee el árbol es  $2^h - 1$ , ya que cada nivel posee el número de nodos del nivel anterior multiplicado por 2. A partir de las dos expresiones anteriores, podemos obtener la altura del árbol en función del número de nodos:

Suponemos que todos los nodos son 4-Nodo (cota inferior):

$$n = 4^h - 1$$

$$n + 1 = 4^h$$

$$\log_4(n + 1) = \log_4(4^h)$$

$$\log_4(n + 1) = h$$

Suponemos que todos los nodos son 2-Nodo (cota superior):

$$n = 2^h - 1$$

$$n + 1 = 2^h$$

$$\log_2(n + 1) = \log_2(2^h)$$

$$\log_2(n + 1) = h$$

(P.72) En el borrado del árbol 2-3-4 sólo se reduce la altura del árbol cuando p, q y r son 2-Nodo.

Verdadero. Cuando se cumple la condición de que p, q y r son 2-Nodo, tenemos que hacer una combinación. En el proceso de combinación eliminamos un nodo. Solamente se puede dar esta condición cuando p es la raíz, por lo tanto eliminamos la raíz y el árbol reduce su altura.

(P.73) El número de elementos que hay en un árbol 2-3-4 de altura  $h$  está comprendido entre  $2^h - 1$  y  $4^h - 1$ .

Verdadero. Si suponemos que todos los nodos del árbol son 4-Nodo, para una altura  $h$  el número de nodos que posee el árbol es  $4^h - 1$ , ya que cada nivel posee el número de nodos del nivel anterior multiplicado por 4. Por otro lado, si suponemos que todos los nodos del árbol son 2-Nodo,

para una altura  $h$  el número de nodos que posee el árbol es  $2^h - 1$ , ya que cada nivel posee el número de nodos del nivel anterior multiplicado por 2.

**(P.74) En la inserción de un árbol 2-3-4, sólo crece la altura del árbol cuando se realiza una operación de DIVIDERAIZ.**

Verdadero. En la operación DIVIDERAIZ creamos un nodo nuevo para cada uno de los elementos de la raíz. El elemento medio pasa a ocupar el puesto de la nueva raíz, el elemento izquierdo pasa a ser hijo izquierdo y el elemento derecho pasa a ser hijo derecho. En las operaciones DIVIDEHIJODE2 y DIVIDEHIJODE3 solamente se crean nodos que están al mismo nivel que el nodo p.

### 5.7.2. Ejercicios

**(E.29) Realiza los siguientes apartados:**

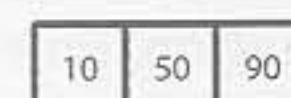
**A)** Inserta en un árbol 2-3-4 inicialmente vacío las siguientes etiquetas: 10, 50, 90, 20, 30, 80, 100, 40, 70, 85, 75, 65, 60, 55, 130, 160, 190, 200, 210, 220 y 230.

**B)** En el árbol resultado del apartado anterior, realiza los siguientes borrados: 10 y 100. (*Criterio: en caso de dos hermanos, escoge el de la izquierda*).

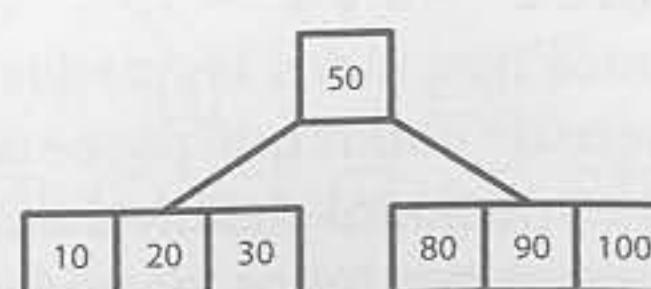
SOLUCIÓN:

A) Recordemos que en un árbol 2-3-4, cuando en el camino de búsqueda para insertar una etiqueta se encuentra un 4-Nodo, se tiene que dividir.

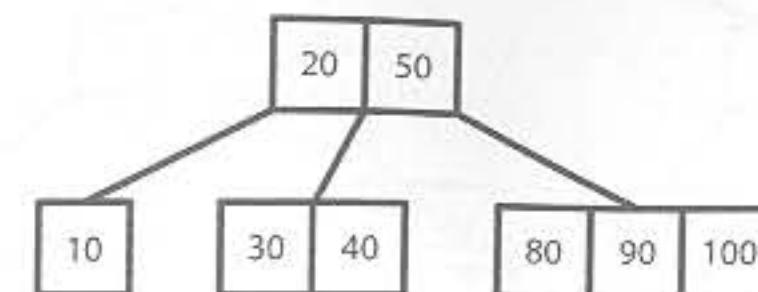
Insertamos 10, 50 y 90, se obtiene un 4-Nodo.



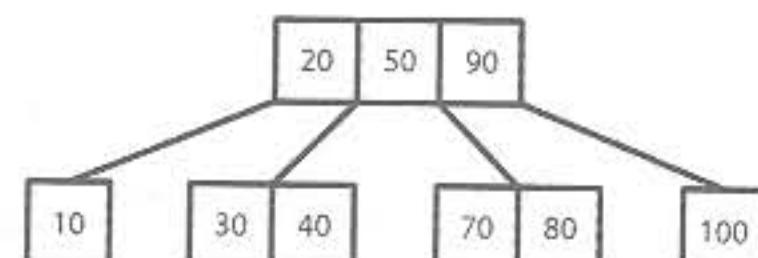
Al insertar el 20 se divide la raíz que es un 4-Nodo y queda como raíz un 2-Nodo con la etiqueta 50. A continuación insertamos 30, 80 y 100.



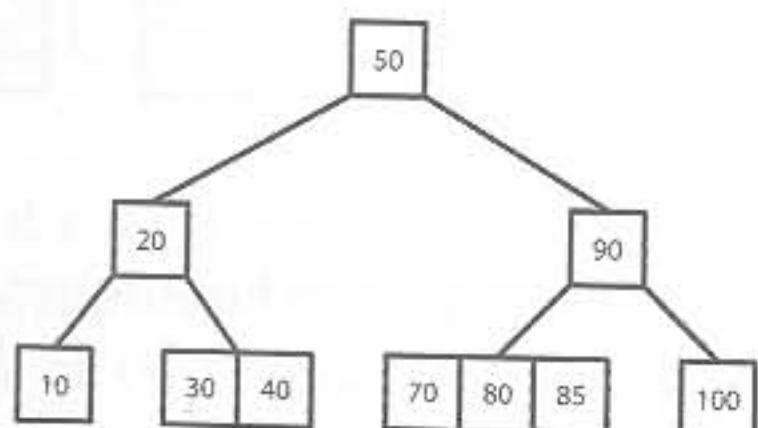
Al insertar 40, se divide el 4-Nodo con las etiquetas 10, 20 y 30. La etiqueta 20 asciende a su nodo padre, que es el nodo raíz.



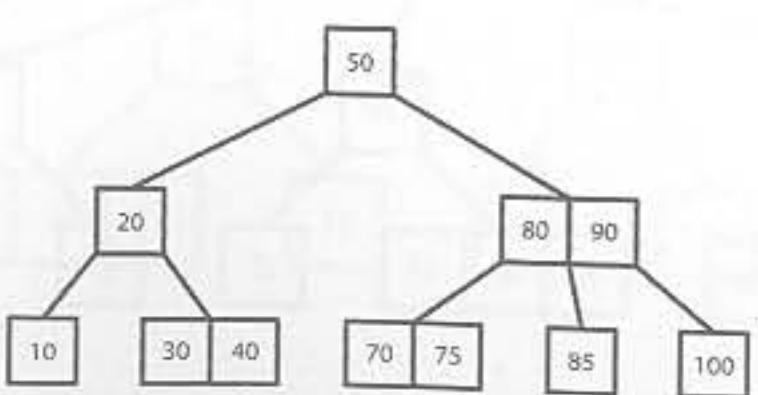
Al insertar 70, se divide el 4-Nodo con las etiquetas 80, 90 y 100. La etiqueta 90 sube al nodo padre, que es el nodo raíz. El nodo raíz pasa a ser un 4-Nodo, pero que no se tiene que dividir hasta que no se realice la siguiente inserción.



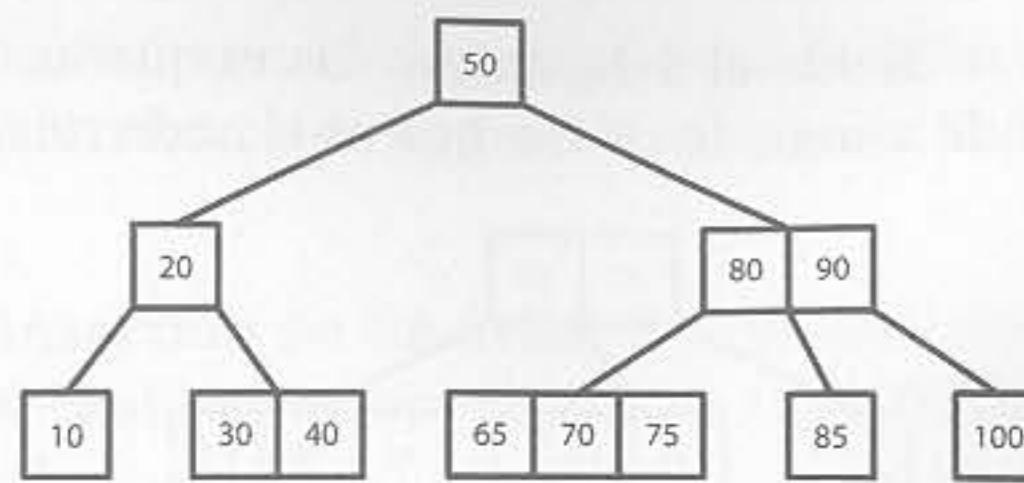
Al insertar 85, se divide la raíz que es un 4-Nodo y queda como raíz un 2-Nodo con la etiqueta 50.



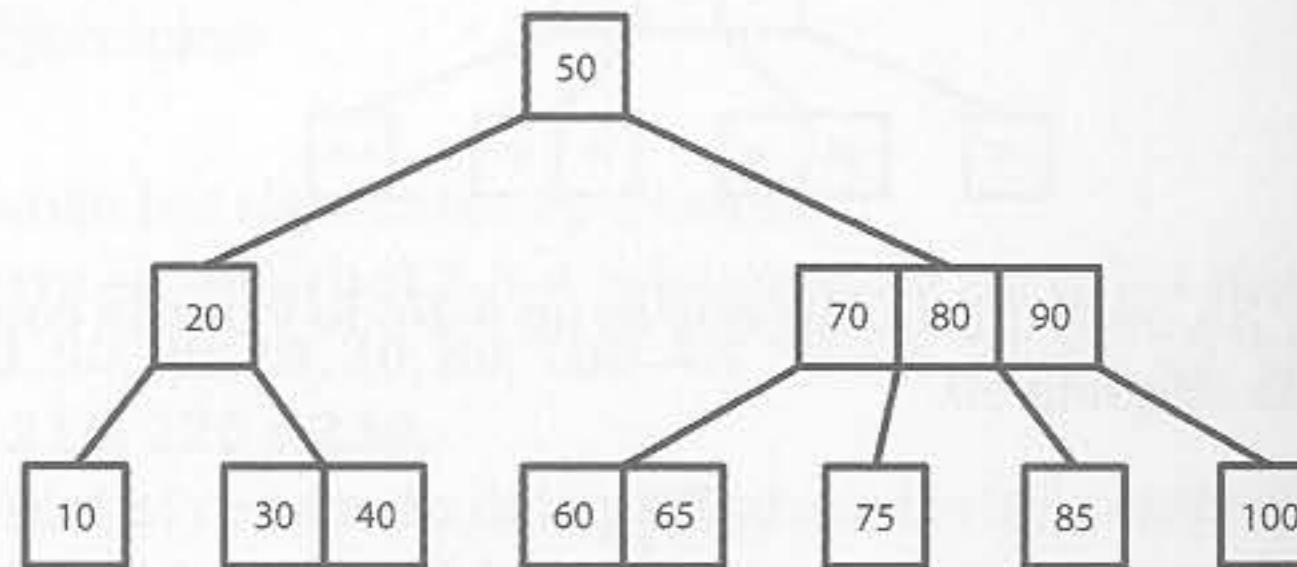
Al insertar 75, se divide el 4-Nodo con las etiquetas 70, 80 y 85. La etiqueta 80 sube al nodo padre.



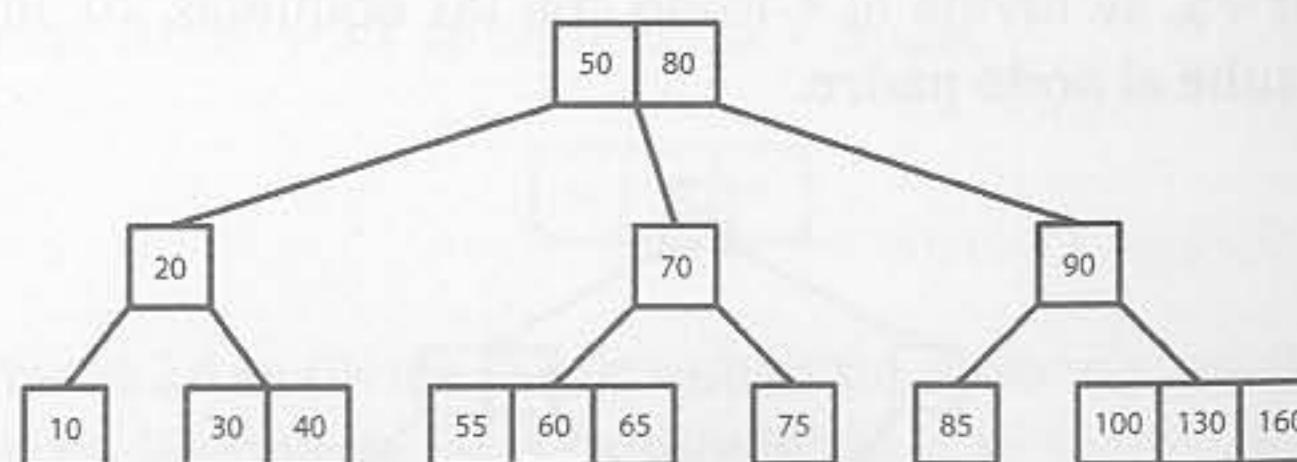
Al insertar 65 no se tiene que realizar ninguna operación de división porque por el camino de búsqueda para insertar la nueva etiqueta no se pasa por un 4-Nodo.



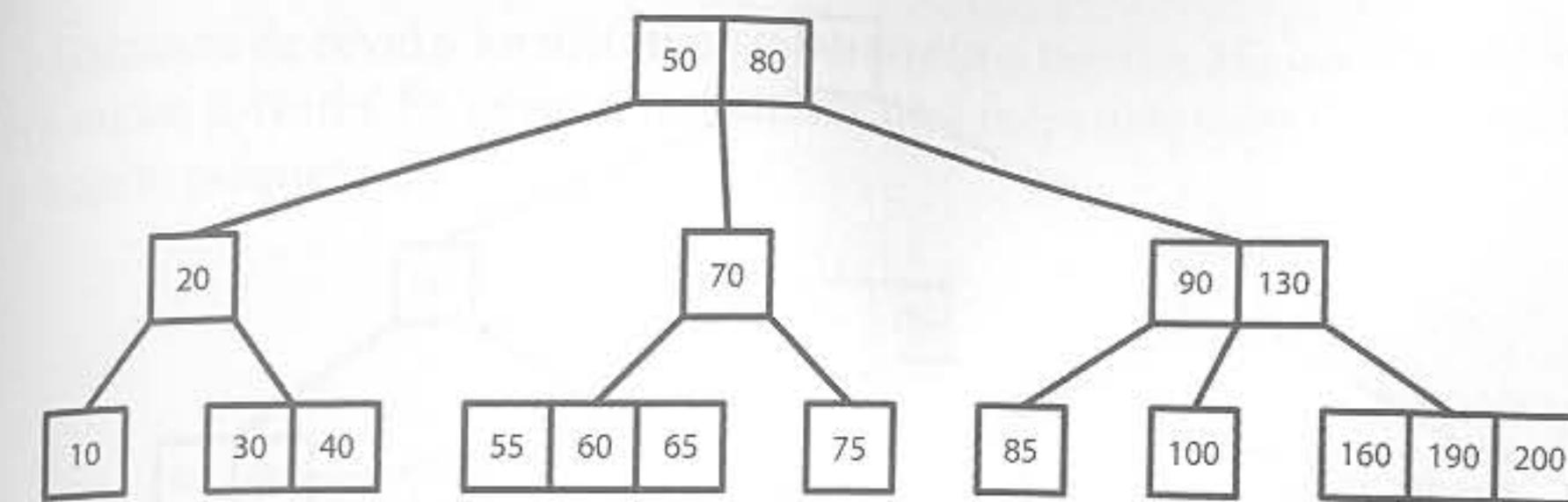
Al insertar 60, se divide el 4-Nodo con las etiquetas 65, 70 y 75. La etiqueta 70 asciende a su nodo padre, que pasa a ser un 4-Nodo, pero que no se tiene que dividir hasta que no se realice la siguiente inserción.



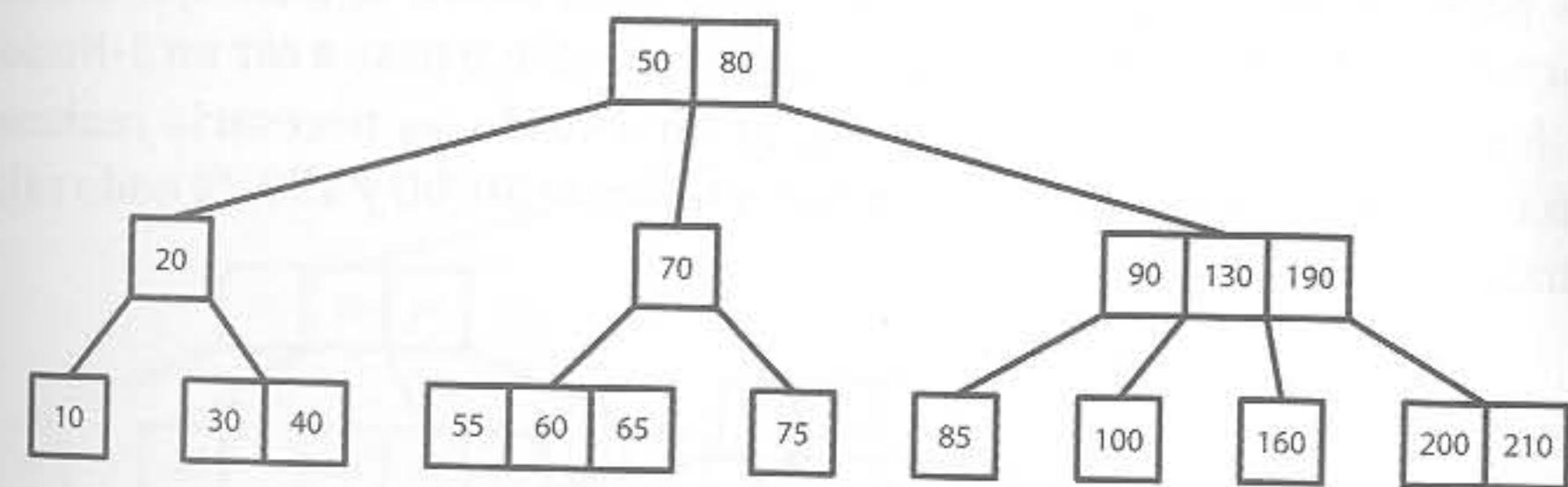
Al insertar 55, se divide el 4-Nodo con las etiquetas 70, 80 y 90. A continuación, se inserta 130 y 160, sin ningún tipo de división.



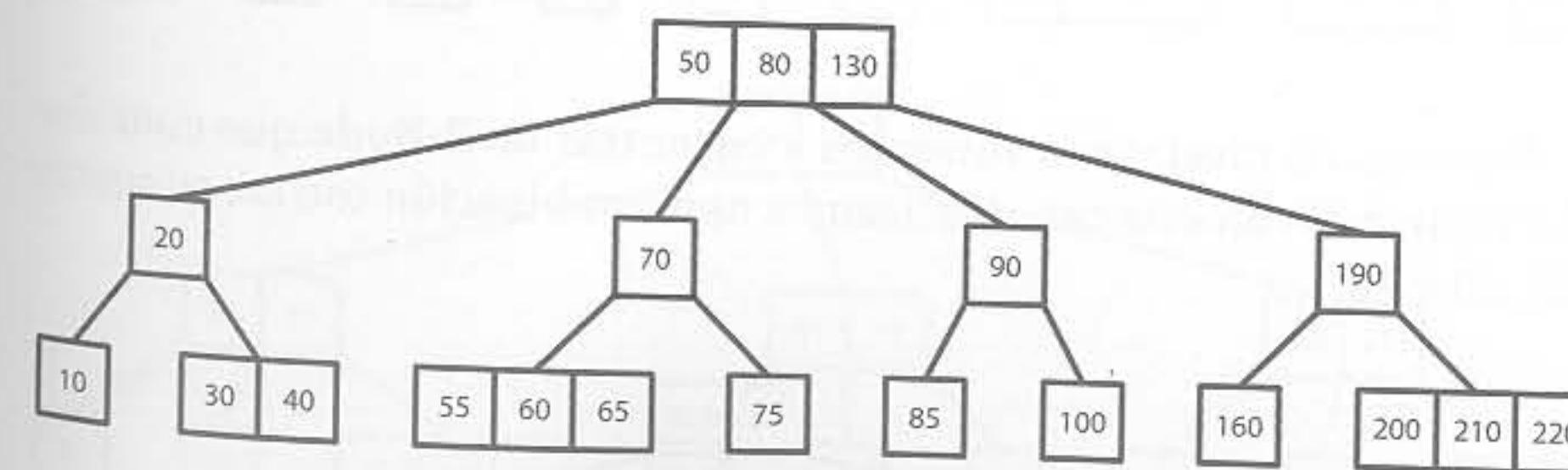
Al insertar 190, se divide el 4-Nodo con las etiquetas 100, 130 y 160. La etiqueta 130 asciende a su nodo padre, que sólo contiene la etiqueta 90. A continuación, se inserta 200, sin ningún tipo de división.



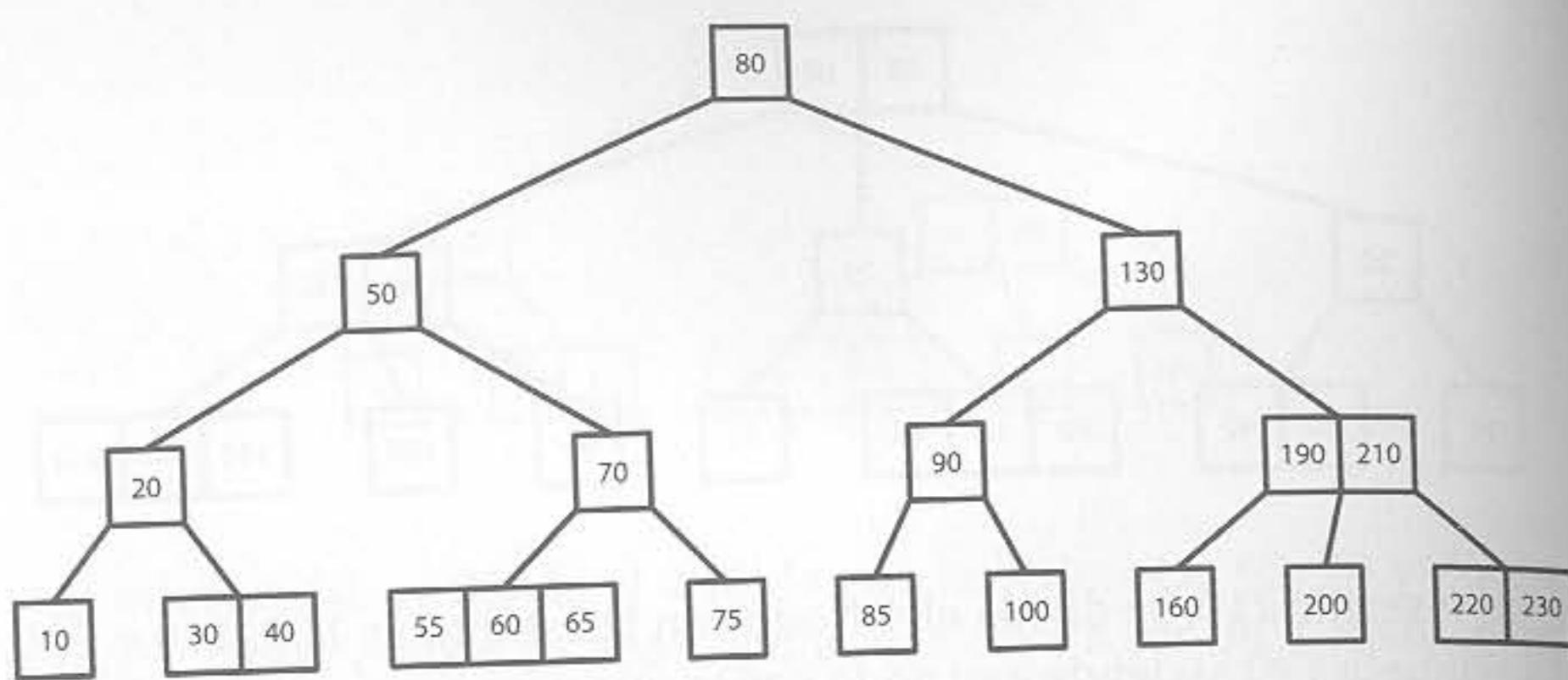
Al insertar 210, se divide el 4-Nodo con las etiquetas 160, 190 y 200. La etiqueta 190 asciende a su nodo padre, que pasa a ser un 4-Nodo, pero que no se tiene que dividir hasta que no se realice la siguiente inserción.



Al insertar 220, se divide el 4-Nodo con las etiquetas 90, 130 y 190. La etiqueta 130 asciende a su nodo padre, que es el nodo raíz del árbol.

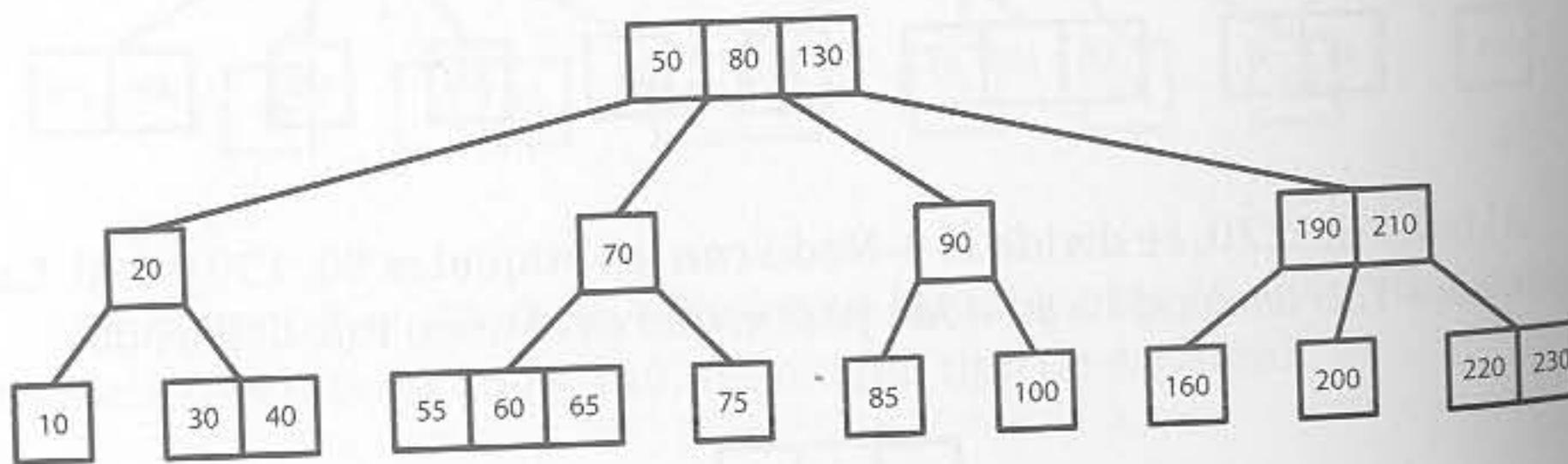


Al insertar 230, se realizan dos divisiones de nodos. Primero se divide el nodo raíz, formado por las etiquetas 50, 80 y 130, por ser un 4-Nodo. La etiqueta 80 pasa a ocupar el nodo raíz del árbol. A continuación, se divide el 4-Nodo con las etiquetas 200, 210 y 220.

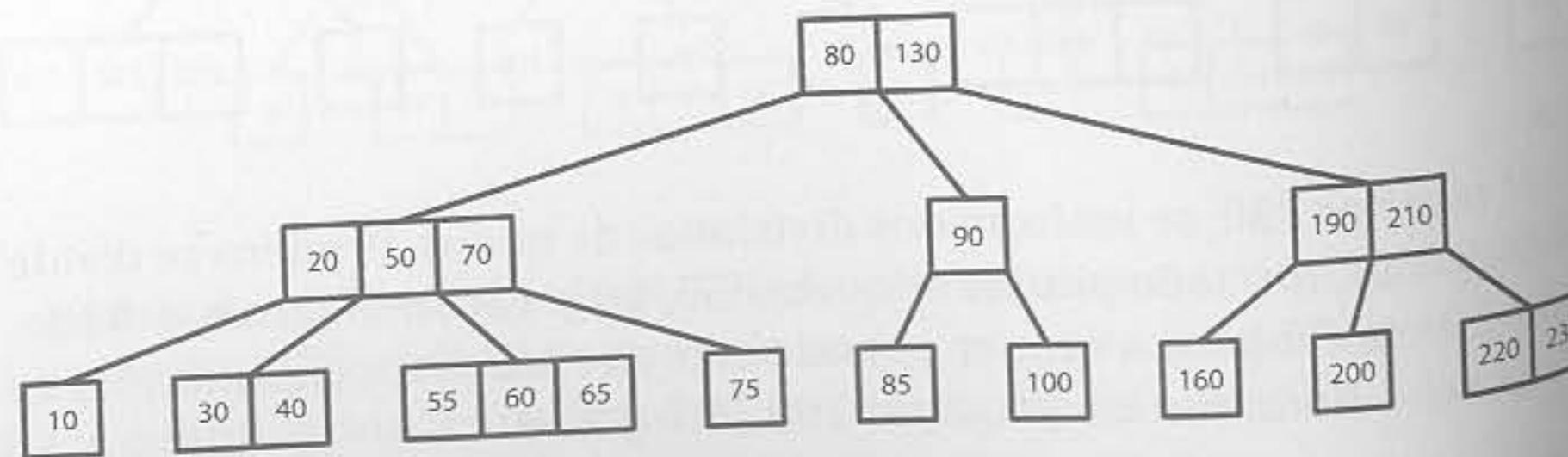


B) Recordemos que en un árbol 2-3-4, cuando en el camino de búsqueda para borrar una etiqueta se encuentra un 2-Nodo, se tiene que transformar, mediante combinación o rotación, para que pase a ser un 3-Nodo.

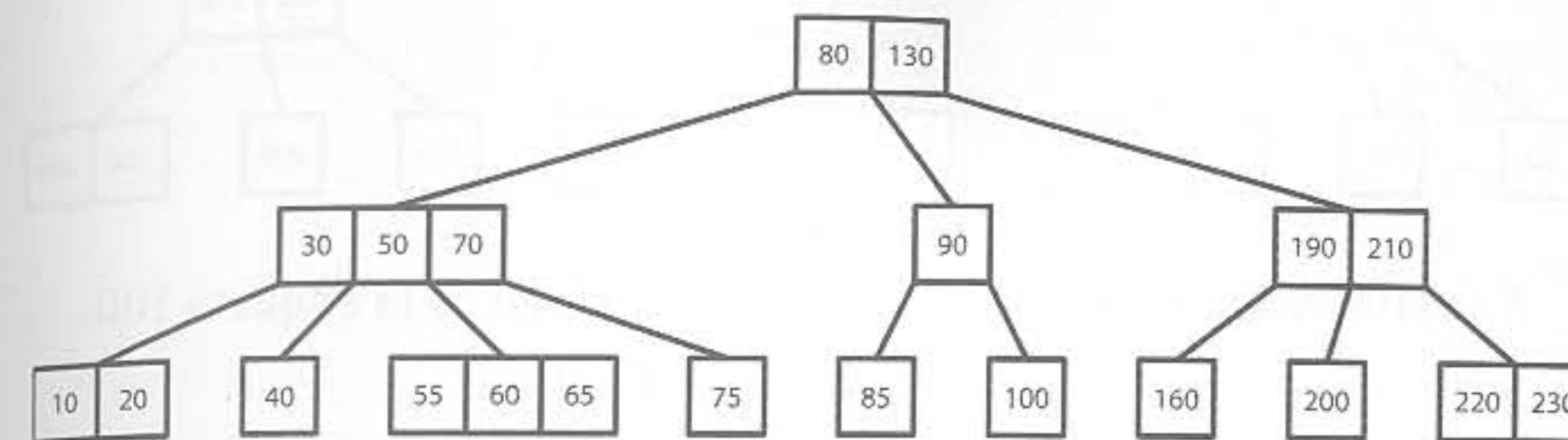
Borramos 10. Como el nodo raíz es un 2-Nodo, es necesario realizar una combinación de los nodos con las etiquetas 50, 80 y 130. El nodo raíz pasa a ser un 4-Nodo.



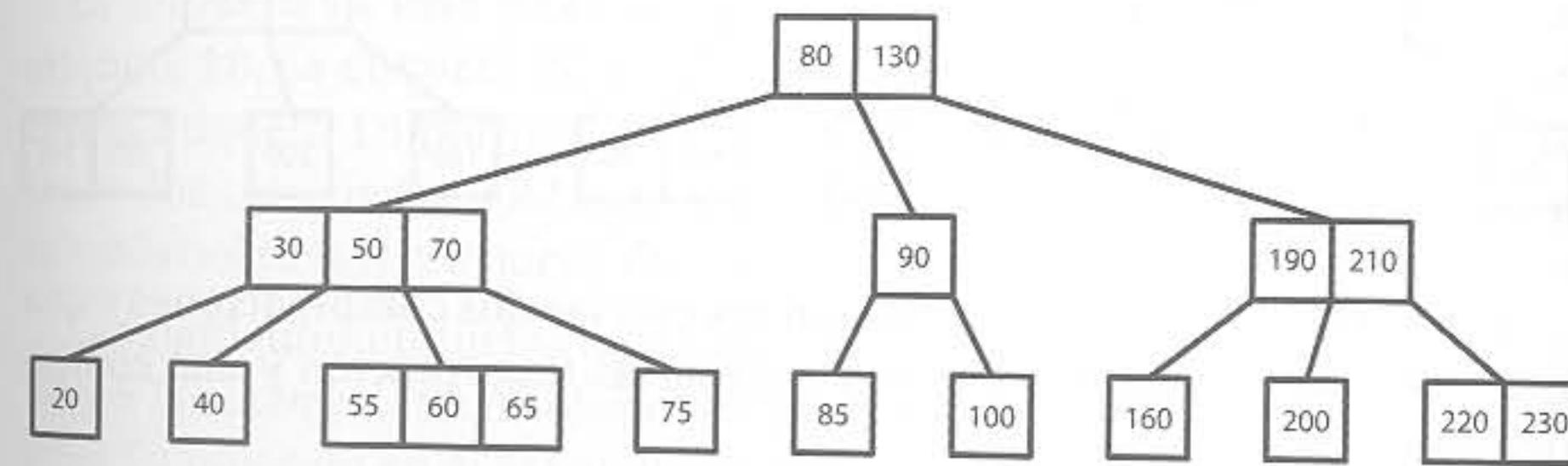
Bajamos de nivel y nos volvemos a encontrar un 2-Nodo que contiene la etiqueta 20. En este caso realizamos una combinación con las etiquetas 20, 50 y 70.



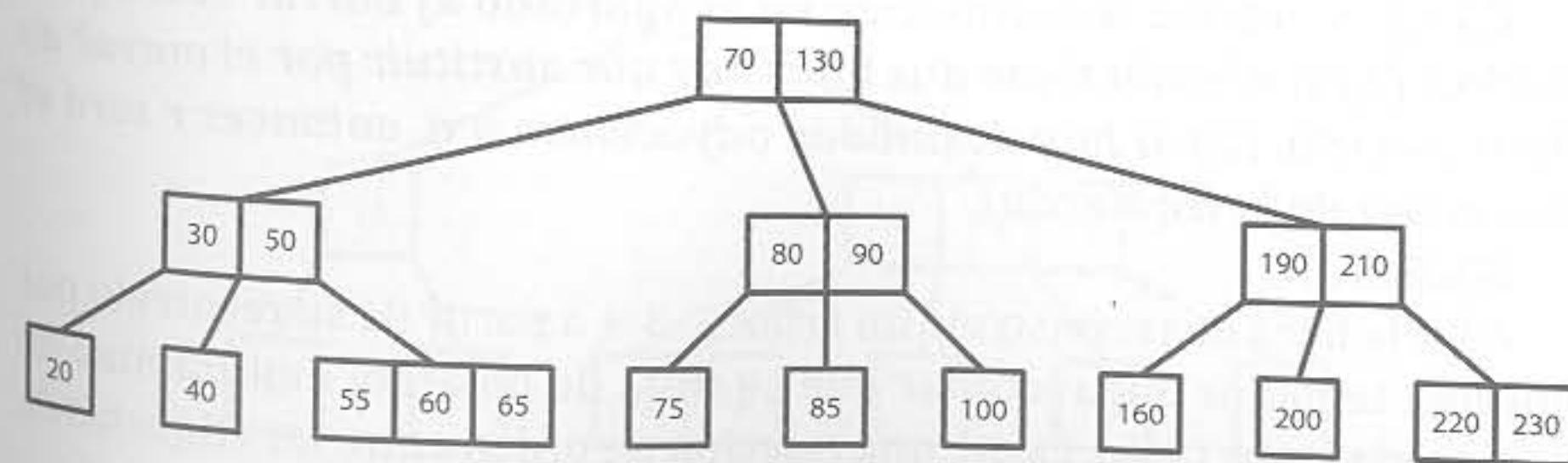
Bajamos de nivel y localizamos la etiqueta a borrar, 10, que se encuentra en un 2-Nodo. Es necesario realizar una rotación: sube la etiqueta 30 y baja la etiqueta 20.



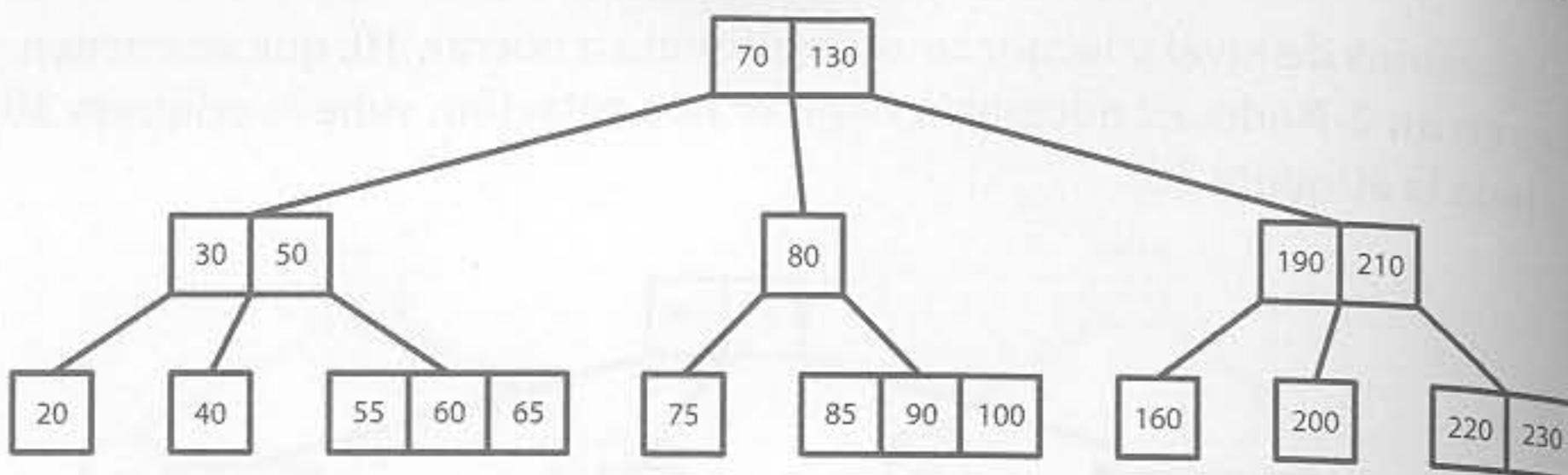
A continuación ya es posible realizar el borrado de la etiqueta 10.



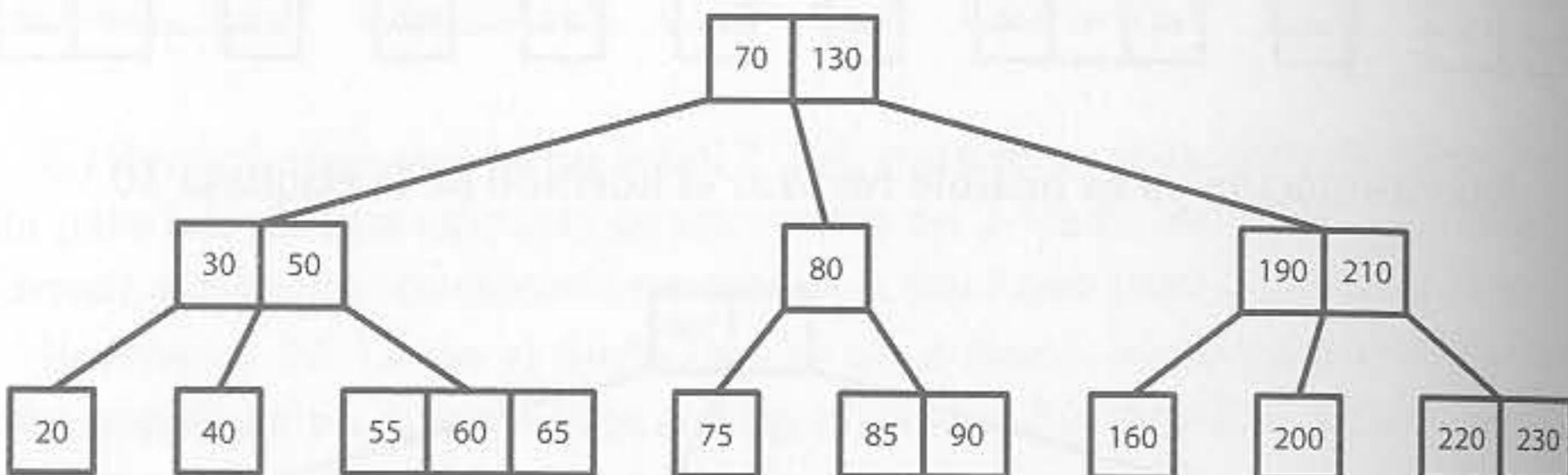
Borramos 100. Conforme realizamos la búsqueda del elemento a borrar, nos encontramos un 2-Nodo con la etiqueta 90. Realizamos una rotación: sube la etiqueta 70 y baja la etiqueta 80.



Bajamos de nivel y localizamos la etiqueta a borrar, 100, que se encuentra en un 2-Nodo. Realizamos una combinación con las etiquetas 85 y 90.



A continuación ya es posible realizar el borrado de la etiqueta 100.



En definitiva, para borrar el 100 son necesarias dos combinaciones y una rotación y para borrar el 100 son necesarias una rotación y una combinación.

#### (E.30) Realiza los siguientes apartados:

**A)** Reconstruir el árbol 2-3-4 a partir del siguiente recorrido en niveles: 25, 18, 35, 45, 55, 14, 22, 28, 32, 37, 53, 57, 10, 16, 19, 23, 26, 30, 34, 36, 38, 52, 54, 56, 58.

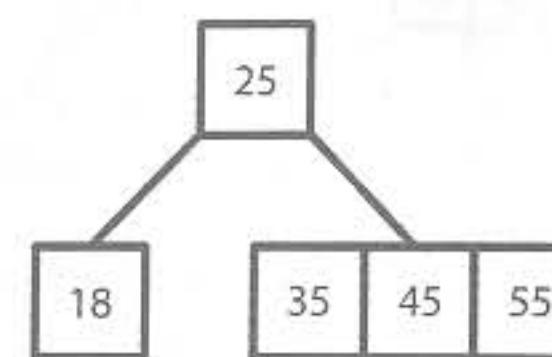
**B)** En el árbol 2-3-4 obtenido en el apartado a) borrar el 25. (*Criterios: (1) si el nodo tiene dos hijos hay que sustituir por el mayor de la izquierda, (2) Si hay dos nodos adyacentes a q, entonces r será el hermano de la izquierda.*)

SOLUCIÓN:

A) A la hora de reconstruir un árbol 2-3-4 a partir de su recorrido por niveles, tenemos que recordar que se trata de un árbol multicamino de búsqueda y, por tanto, existe una relación de orden entre las etiquetas de los nodos.

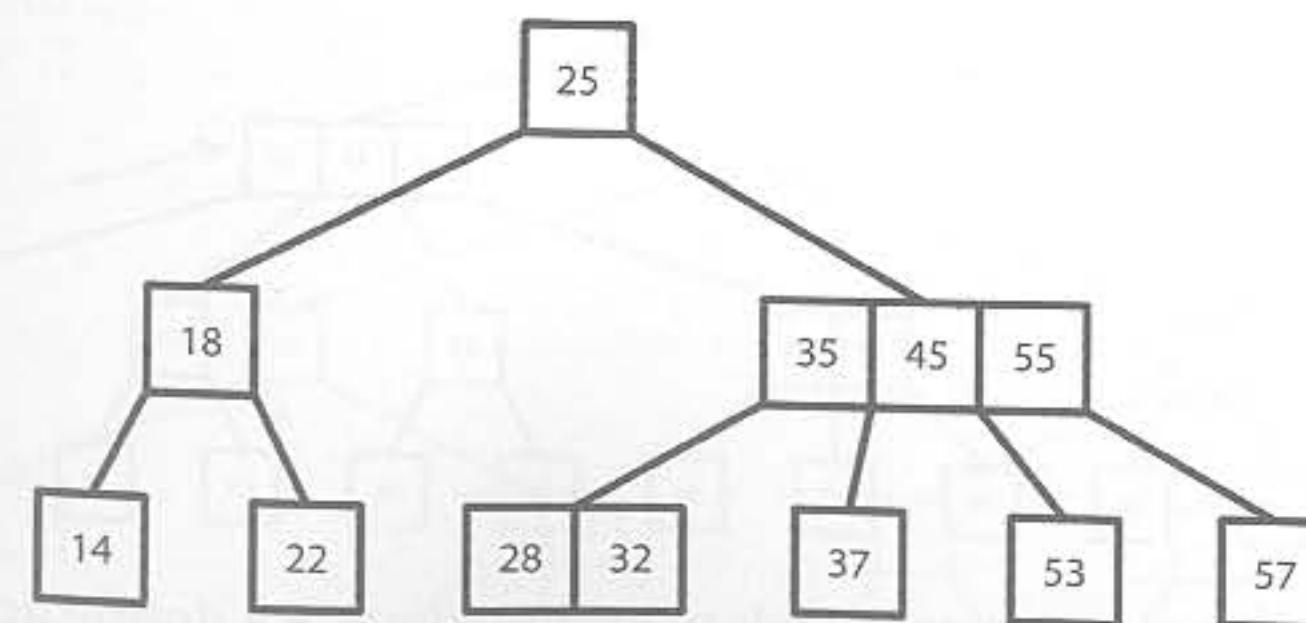
La etiqueta 25 es la única del nodo raíz del árbol. La siguiente etiqueta 18 no puede estar en el nodo raíz, ya que de estarlo tendría que ser mayor que 25. Por tanto, 18 indica el comienzo del recorrido del nivel 2. La

etiqueta 18 está en el subárbol izquierdo del nodo raíz, mientras que las etiquetas 35, 45 y 55 están en el subárbol derecho, ya que son mayores que 25. La etiqueta 14, como es menor que la etiqueta 55, está en el nivel 3.

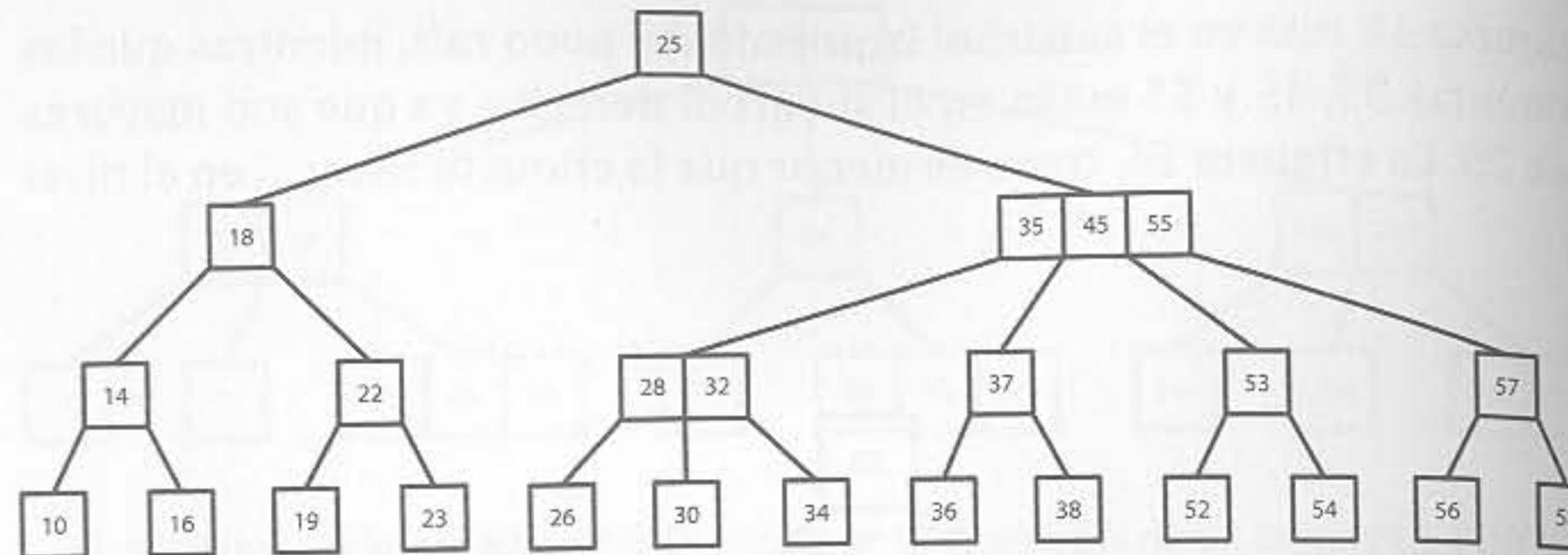


La etiqueta 14 está en el subárbol izquierdo a partir del nodo con la etiqueta 18. La etiqueta 22 está en el subárbol derecho a partir del nodo con la etiqueta 18 por dos razones: el nodo con la etiqueta 18 debe de tener dos subárboles y 22 es menor que 25, por lo que no puede estar en el subárbol derecho a partir de 25.

Las siguientes etiquetas hasta que se encuentre una menor (28, 32, 37, 53 y 57) están en subárboles del nodo formado por las etiquetas 35, 45, y 55. Su posición en el árbol viene dada por el orden que debe de existir entre las etiquetas de un nodo y de sus subárboles.



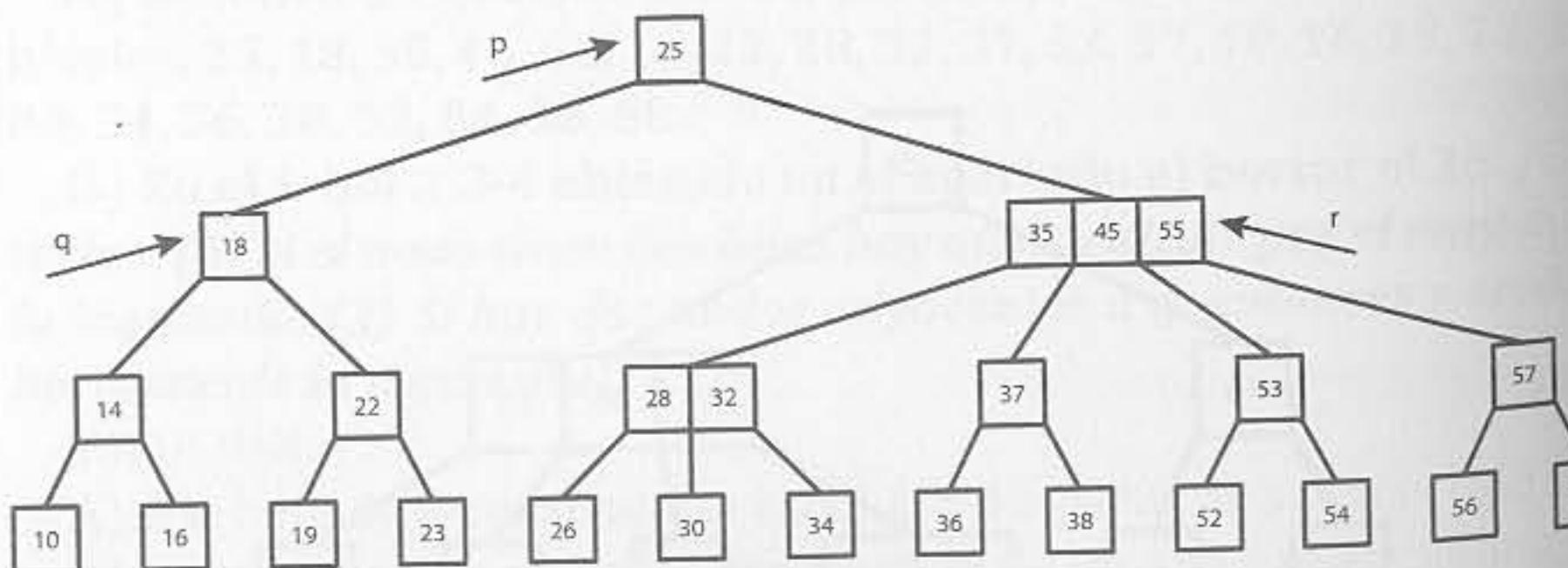
Por último, a partir de los mismos razonamientos empleados anteriormente, podemos reconstruir el nivel 4 y último del árbol.



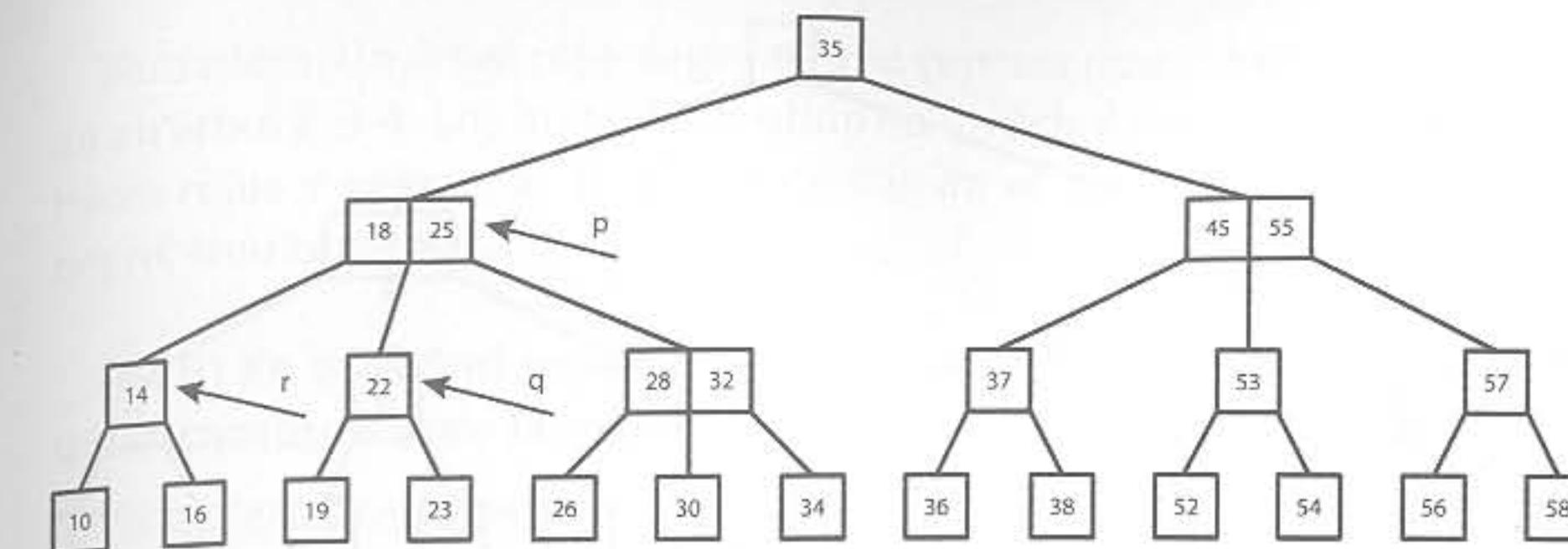
B) Recordemos que en un árbol 2-3-4, el elemento que debemos borrar debe encontrarse siempre en un 3-Nodo o 4-Nodo y tiene que ser un nodo hoja. En caso contrario, será necesario llevar a cabo algunas reestructuraciones en el árbol para lograr dicha condición. La estrategia de reestructuración requiere que en el movimiento de búsqueda, cuando pasemos a un nodo en el siguiente nivel, este nodo debe ser 3 o 4-Nodo.

En este ejercicio, el elemento 25 a borrar está situado en la raíz del árbol. Como el borrado se tiene que llevar a cabo en un nodo hoja, aplicamos el criterio de sustituir por el mayor elemento de la izquierda, en este caso el 23. Por tanto, la búsqueda (q) se tiene que realizar por el subárbol de la izquierda.

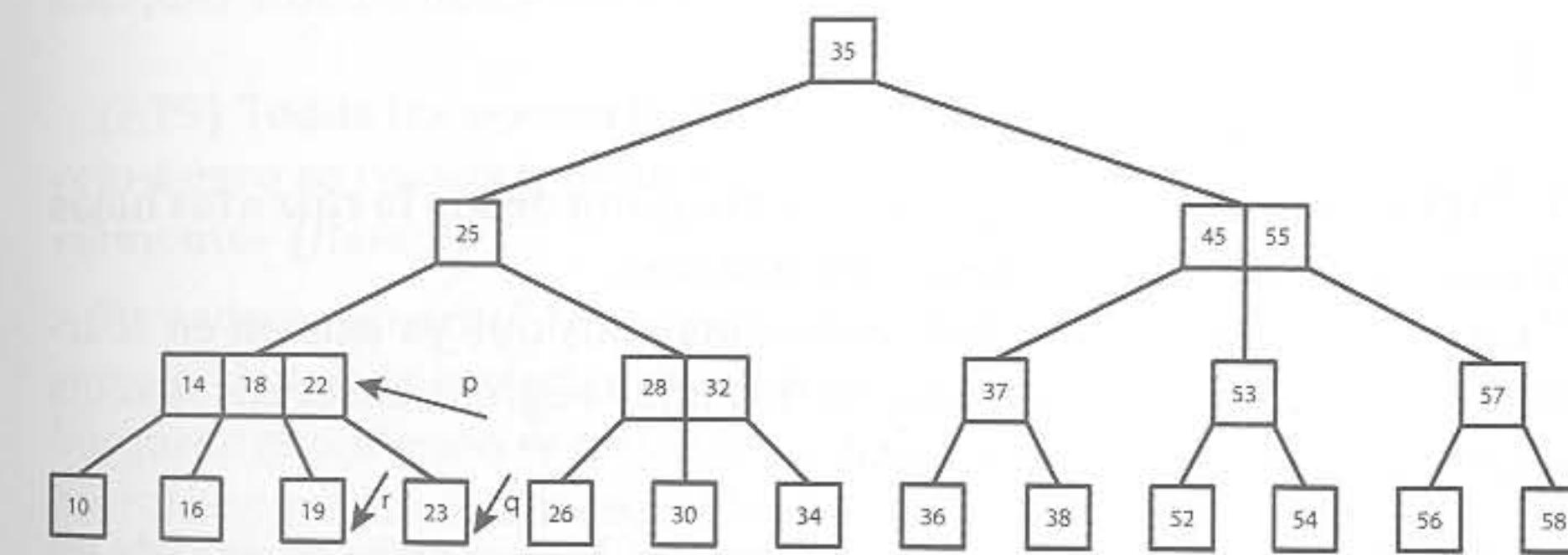
Como q es un 2-Nodo y r es un 4-Nodo, se tiene que realizar una rotación: sube el elemento 35 a su padre (el nodo raíz) y baja el 25 al nodo apuntado por q.



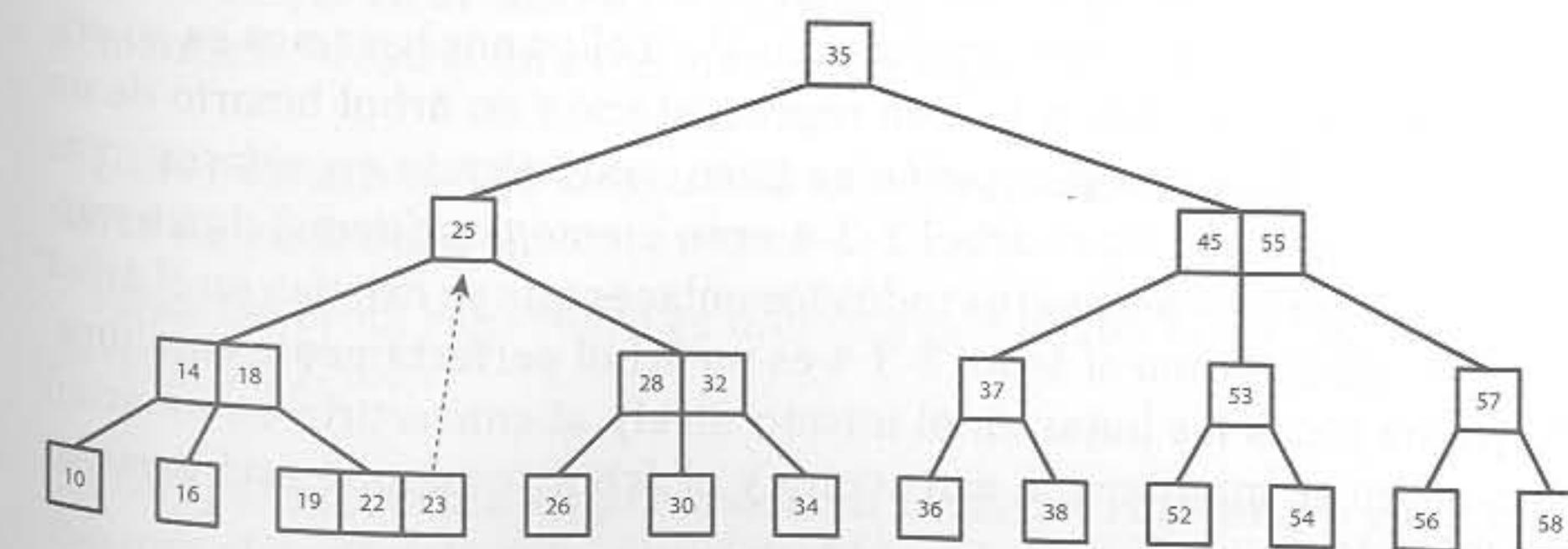
Descendemos al siguiente nivel: p apunta al nodo a donde apuntaba q y q pasa a apuntar al siguiente nodo en el camino de búsqueda del nodo a borrar. Como q es 2-Nodo es necesaria otra reestructuración: r es 2-Nodo y p es 3-Nodo, por lo que realizamos una combinación de los elementos 14, 18 y 22.



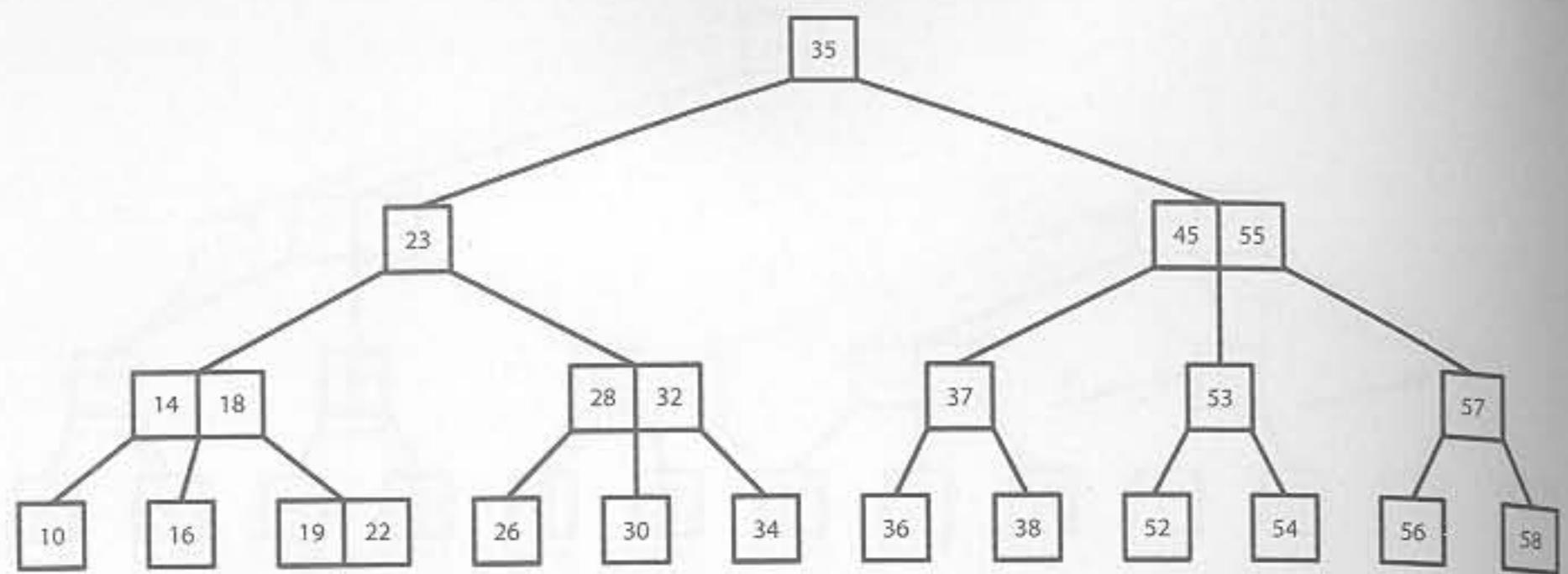
Volvemos a descender al siguiente nivel: q es 2-Nodo, r es 2-Nodo y p es 4-Nodo; realizamos una combinación de los elementos 19, 22 y 23.



Por fin p apunta al nodo que contiene el elemento 23 que deseamos borrar. Pero recordemos que en realidad queremos borrar el elemento 25, por lo que una vez borrado el elemento 23 lo emplearemos para sustituir el elemento 25.



Una vez realizada la sustitución del 25 por el 23, este es el árbol que se obtiene.



## 5.8. Árbol rojo-negro

### 5.8.1. Preguntas

**(P.75)** En un árbol rojo-negro ningún camino desde la raíz a las hojas tiene dos o más hijos negros consecutivos.

Falso. Los hijos negros representan los hijos que ya existen en el árbol 2-3-4 original. Pueden haber tantos hijos negros consecutivos como altura tenga el árbol 2-3-4 original.

**(P.76)** Todas las operaciones (inserción, borrado y búsqueda) de un árbol rojo-negro se realizan en un tiempo  $O(\log_2 n)$ , siendo  $n$  el número de nodos.

Verdadero. Un árbol rojo-negro es un árbol binario de búsqueda. La complejidad de la inserción, borrado y búsqueda en un árbol binario de búsqueda equilibrado es  $O(\log_2 n)$ . Sólo nos queda demostrar que un árbol rojo-negro es un árbol equilibrado. Para ello nos basamos en su origen. Un árbol rojo-negro es una representación en árbol binario de un árbol 2-3-4. Esta representación se hace convirtiendo en enlaces rojos todos los hijos que en el árbol 2-3-4 eran elementos (*items*) de un mismo nodo y en enlaces negros todos los enlaces que ya existían en el árbol 2-3-4 original. Como el árbol 2-3-4 es un árbol perfectamente equilibrado (tiene todas las hojas en el mismo nivel), al convertirlo en un árbol rojo-negro se mantiene la estructura y el árbol resultante está pseudo-equilibrado.

**(P.77)** Un árbol rojo-negro es un árbol binario de búsqueda que representa a un árbol 2-3-4.

Verdadero. Un árbol rojo-negro es una representación en árbol binario de un árbol 2-3-4. Los hijos de un nodo en un árbol rojo-negro son de dos tipos: rojos y negros. Si el hijo ya existía en el árbol 2-3-4 original será negro, sino será rojo.

**(P.78)** En un árbol rojo-negro, el número de hijos negros en cualquier camino desde la raíz a las hojas es siempre el mismo.

Verdadero. Los hijos negros en un árbol rojo-negro representan los enlaces que ya existían en el árbol 2-3-4. Como en un árbol 2-3-4 todas las hojas están al mismo nivel, todos los caminos que van desde la raíz hasta las hojas tienen la misma longitud. Por tanto, todos los hijos negros en cualquier camino desde la raíz a las hojas es siempre el mismo.

**(P.79)** Todas las operaciones (inserción y búsqueda) de un árbol rojo-negro se realizan en un tiempo  $O(\log_2 n)$ , siendo  $n$  el número de elementos (*items*).

Verdadero. Un árbol rojo-negro es un árbol binario de búsqueda. La complejidad de la inserción, borrado y búsqueda en un árbol binario de búsqueda equilibrado es  $O(\log_2 n)$ . Solo nos queda demostrar que un árbol rojo-negro es un árbol equilibrado. Para ello nos basamos en su origen. Un árbol rojo-negro es una representación en árbol binario de un árbol 2-3-4. Esta representación se hace convirtiendo en enlaces rojos todos los hijos que en el árbol 2-3-4 eran elementos de un mismo nodo y en enlaces negros todos los enlaces que ya existían en el árbol 2-3-4 original. Como el árbol 2-3-4 es un árbol perfectamente equilibrado (tiene todas las hojas en el mismo nivel), al convertirlo en un árbol rojo-negro se mantiene la estructura y el árbol resultante está pseudo-equilibrado.

**(P.80)** Un árbol rojo-negro es una representación sobre árbol binario de un árbol 2-3.

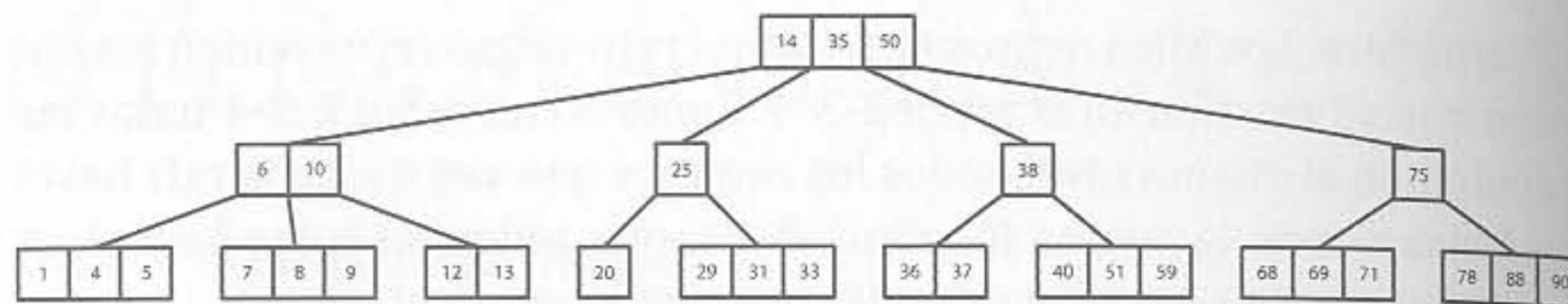
Falso. Un árbol rojo-negro es una representación sobre árbol binario de un árbol 2-3-4.

**(P.81)** El recorrido en preorden en un árbol rojo-negro es el mismo que el realizado sobre un árbol binario.

Verdadero. Un árbol rojo-negro es un árbol binario. El recorrido en preorden es igual para todos los árboles binarios.

## 5.8.2. Ejercicios

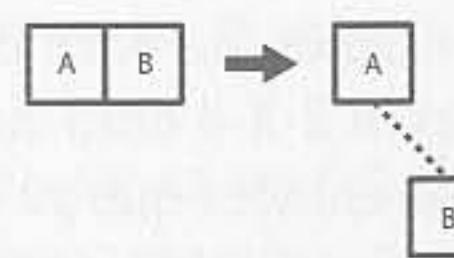
(E.31) Transformar el siguiente árbol 2-3-4 en su correspondiente árbol rojo-negro. Generar los dos árboles posibles que salen al aplicar los distintos criterios de transformación.



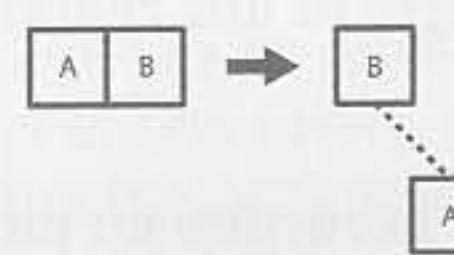
SOLUCIÓN:

Los posibles criterios de transformación son los siguientes:

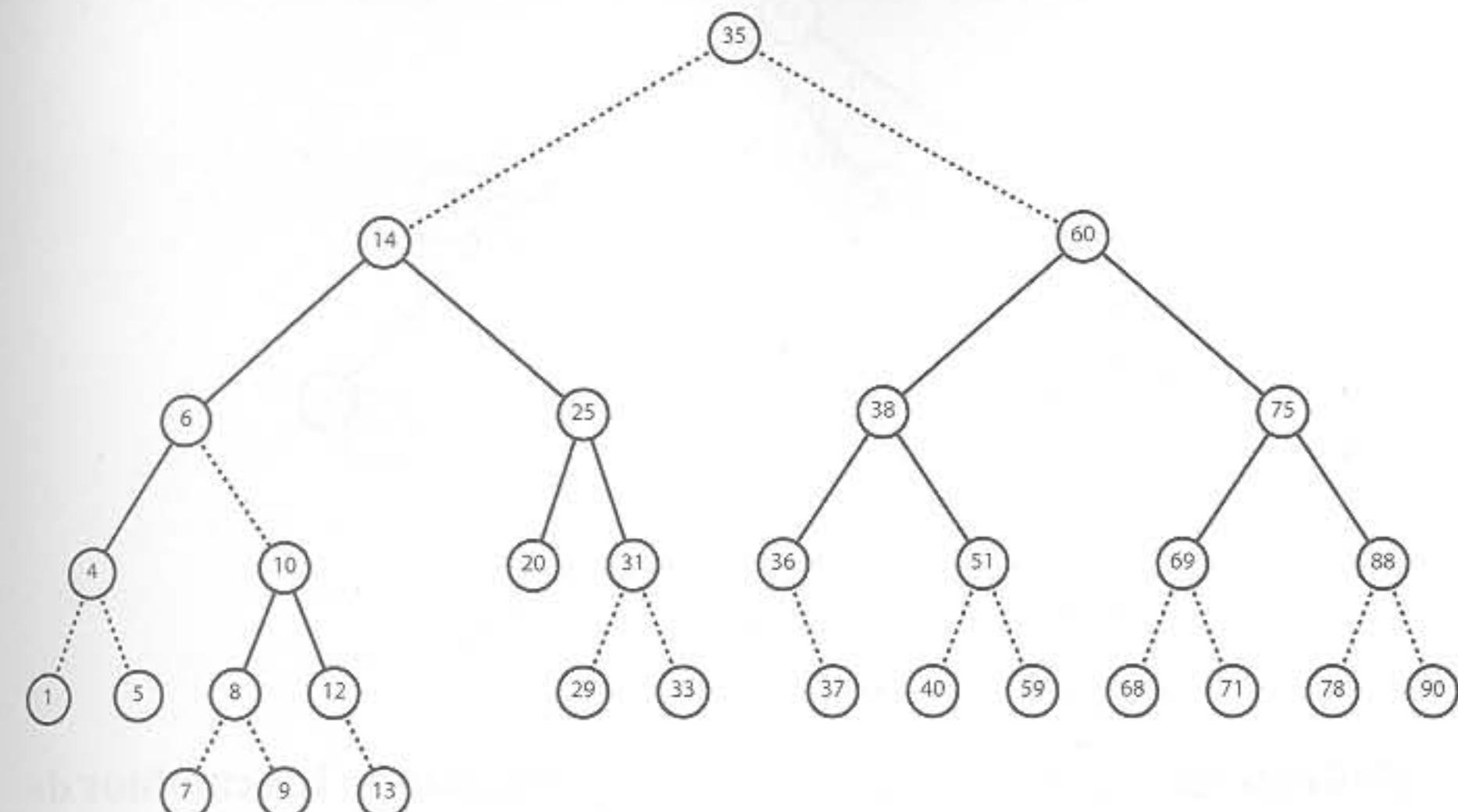
- Decidir que en el nodo del tipo 3-Nodo, el padre sea el elemento de la izquierda y que el hijo rojo sea el elemento medio.



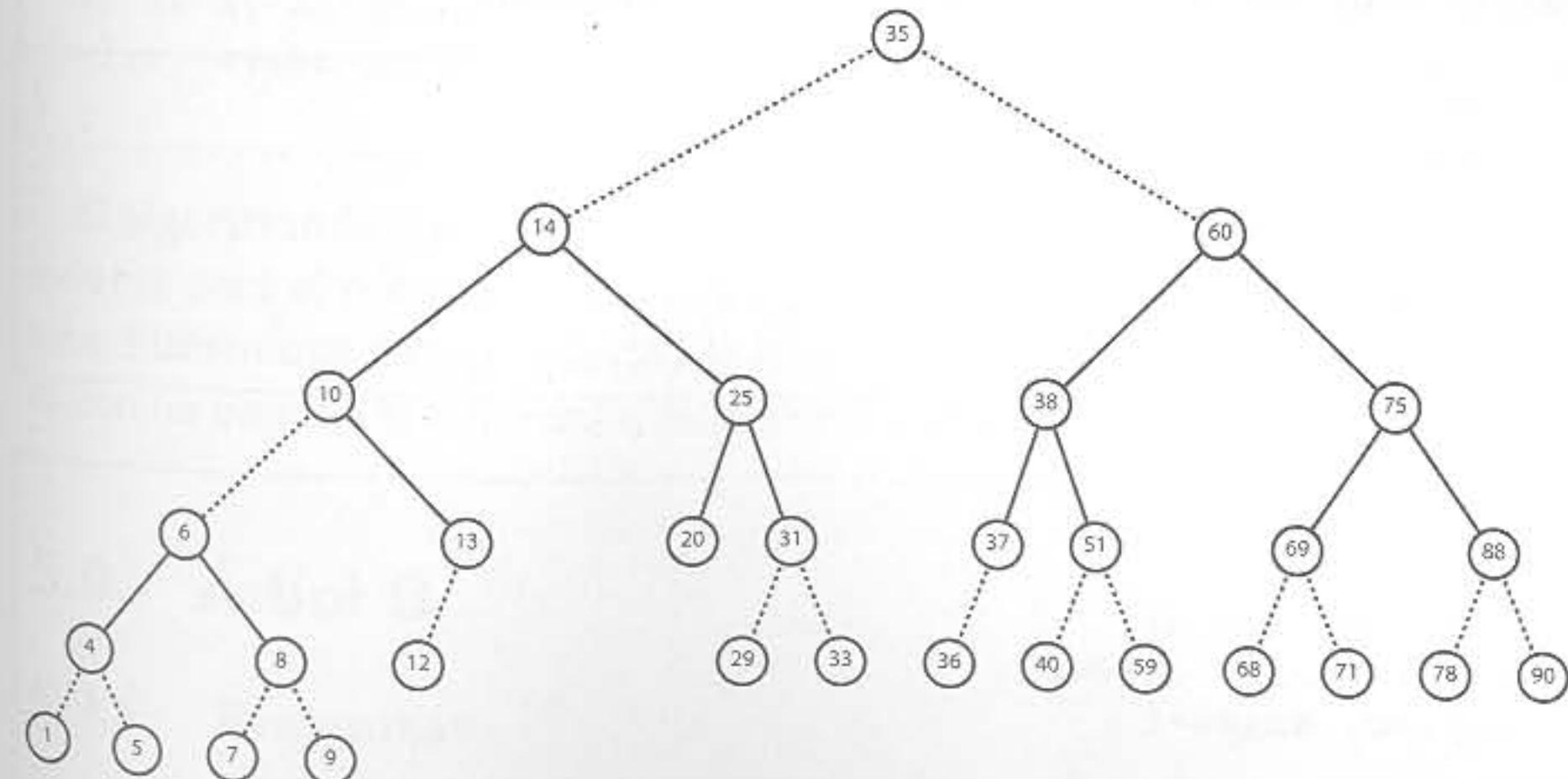
- Decidir que en el nodo del tipo 3-Nodo, el padre sea el elemento medio y que el hijo rojo sea el elemento de la izquierda.



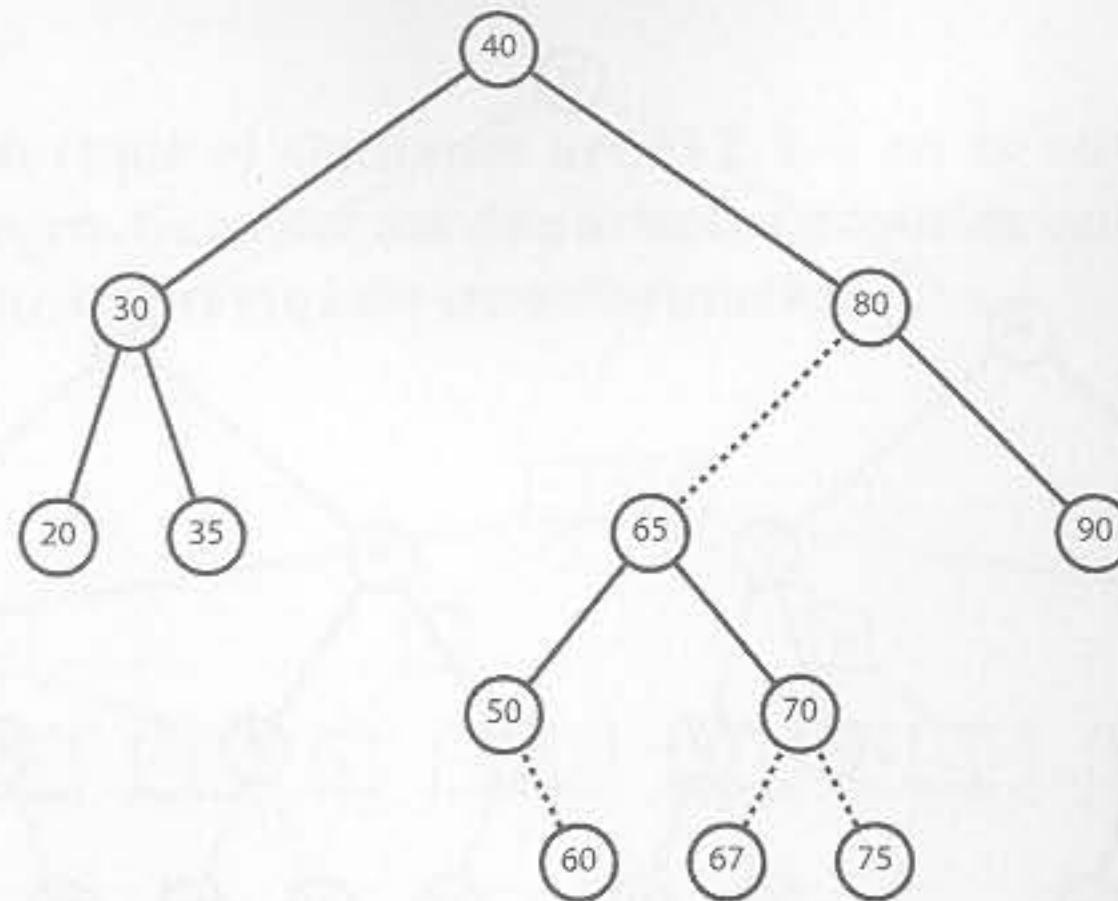
## SOLUCIÓN CON EL CRITERIO 1:



## SOLUCIÓN CON EL CRITERIO 2:



(E.32) Dado el siguiente árbol rojo-negro:



A) Realizar la inserción de la clave 72, detallando los cambios de color y rotaciones empleadas.

B) Escribir en C++ el código de la función

```
int TarbolRN::Altura234(void)
```

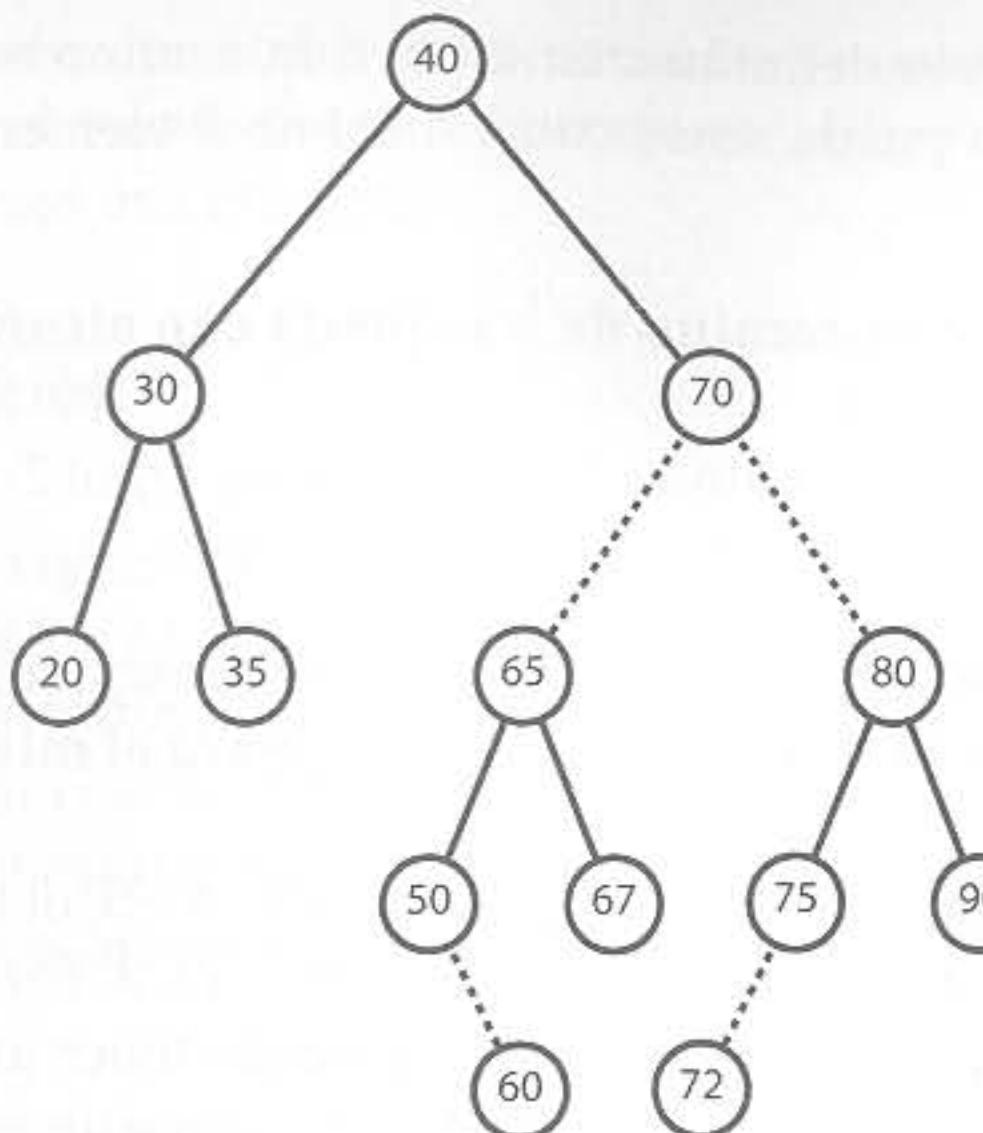
que devuelve la altura del árbol 2-3-4 equivalente al rojo-negro sobre el que trabaja la función. Añadir una breve explicación de 5 líneas como máximo de su funcionamiento. La representación del árbol rojo-negro y de su nodo será la siguiente:

```
class TarbolRN {
    Altura234(void)
    ...
private:
    Tnodo *p;
};
```

```
class Tnodo {
    int clave;
    TarbolRN iz, de;
    int colorIZ, colorDE;
    //Rojo=0, Negro=1
};
```

SOLUCIÓN:

A) 1 cambio de color y 1 rotación ID



B) Una versión recursiva del algoritmo podría ser la siguiente (la versión equivalente iterativa también sería válida):

```
int TarbolRN::Altura234 (void) {
    if(p == NULL) return 0;
    // Si el enlace con hijo es N
    else if(p->colorIZ == 1) return 1 + p->iz.Altura234();
    // Si el enlace con hijo es R
    else return p->iz.Altura234();
}
```

El algoritmo funciona del siguiente modo: la altura del árbol 2-3-4 equivalente será el número de hijos negros de cualquiera de sus ramas más uno. Puesto que esta propiedad la ha de cumplir cualquier rama, el algoritmo ha elegido la izquierda, pero sería válido también para la derecha.

## 5.9. Árbol B

### 5.9.1. Preguntas

(P.82) El nodo de un árbol B m-camino de búsqueda con  $m = 4$  puede tener como máximo 4 hijos no vacíos.

Verdadero. Cuando definimos un árbol B m-camino de búsqueda con  $m = 4$  cada nodo puede tener como máximo 3 elementos (items) y 4 hijos no vacíos.

**(P.83) El árbol B m-camino de búsqueda con altura  $k$  tiene como máximo  $m^k - 1$  claves.**

Verdadero. En un árbol binario:  $2^k - 1$ , en un árbol 2-3:  $3^k - 1$ , en un árbol 2-3-4:  $4^k - 1$ . En un árbol  $m$  de altura  $k$ :  $m^k - 1$ .

**(P.84) La altura del árbol B m-camino de búsqueda con  $m > 5$  es menor o igual que la del árbol 2-3, o 2-3-4 para el mismo número de etiquetas.**

Verdadero. Cuando tenemos un árbol B con  $m > 5$ , el número mínimo de etiquetas que puede tener un nodo es  $(m - 1)/2$ . Por lo tanto para  $m = 6$ , el numero mínimo de elementos que puede tener un nodo es 2. Por otro lado el número mínimo de etiquetas que tiene un nodo en un árbol 2-3 o en un árbol 2-3-4 es 1. Como todos los tipos de árboles que estamos planteando (árbol 2-3, 2-3-4, y árbol B) tienen que cumplir la propiedad de tener todas las hojas en el mismo nivel, el árbol que tendrá menor altura será el árbol que pueda contener el mayor número de etiquetas como mínimo en un nodo. Éste es el árbol B con  $m > 5$ .

**(P.85) El grado de un árbol B m-camino de búsqueda es  $m$ .**

Verdadero. El grado de un árbol es la cantidad de nodos hijos que puede tener cada nodo del árbol como máximo. Un árbol B de grado  $m$  puede tener como máximo  $m$  hijos.

**(P.86) El nodo de un árbol B m-camino de búsqueda con  $m = 100$  puede tener como máximo 99 claves.**

Verdadero. Por definición la cantidad de claves o elementos (*items*) que puede tener un nodo de un árbol B de orden  $m$  es  $m - 1$ .

**(P.87) El árbol B m-camino de búsqueda con altura  $k$  tiene como máximo  $m^{k-1}$  claves.**

Falso. Un árbol B m-camino de búsqueda con altura  $k$  tiene como máximo  $m^{k-1}$  claves en cada nivel o altura, pero el número total de claves máximo es  $m^k - 1$ .

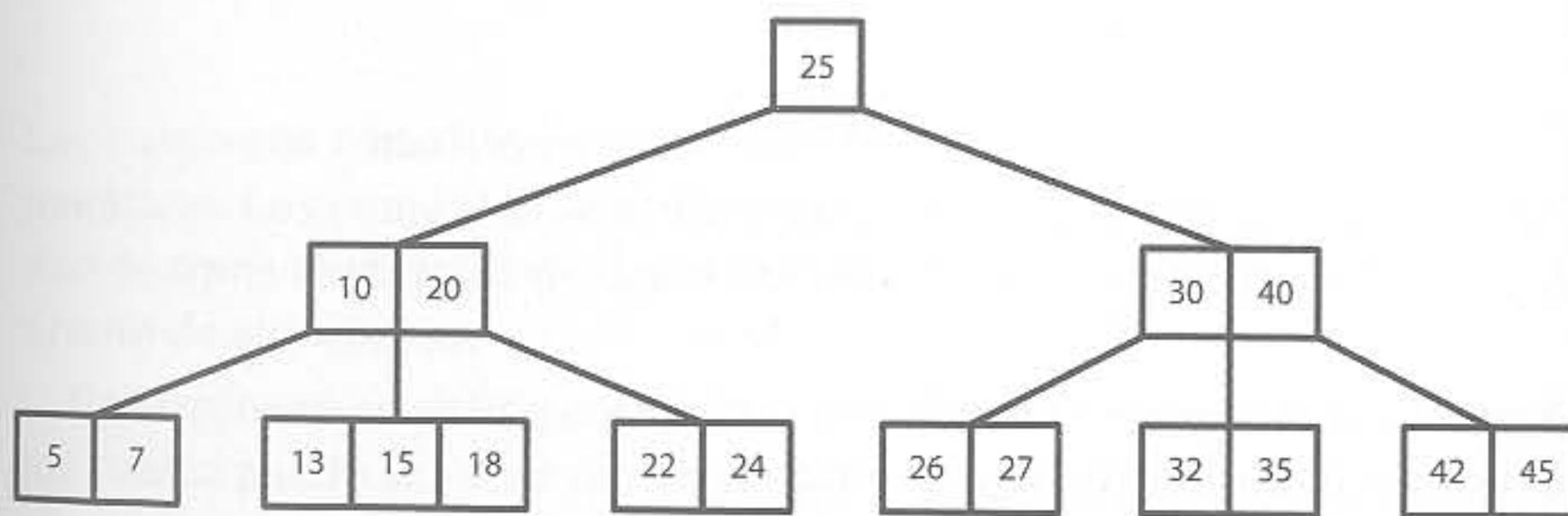
**(P.88) La raíz del árbol B m-camino de búsqueda siempre tiene al menos una etiqueta.**

Verdadero. Todos los nodos de un árbol B m-camino de búsqueda de orden  $m$  tienen al menos  $(m - 1)/2$  etiquetas, excepto la raíz que tiene que tener al menos una etiqueta.

## 5.9.2. Ejercicios

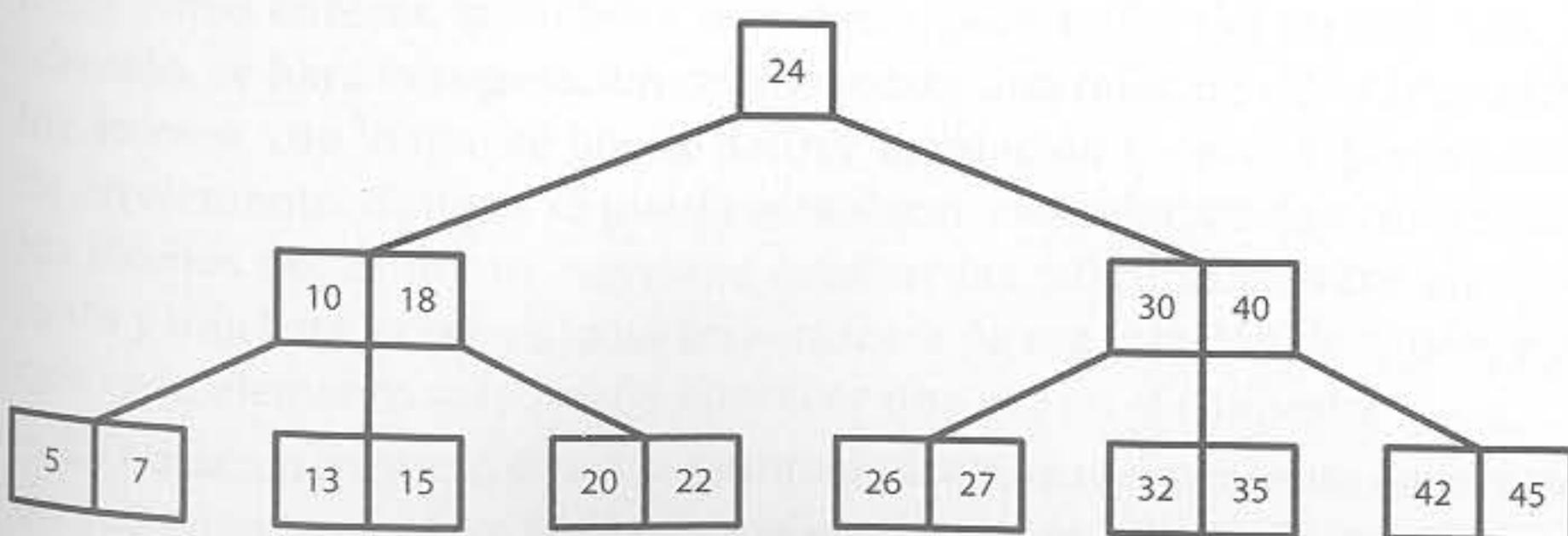
**(E.33) Dado el siguiente árbol B con  $m = 5$ , realizar el borrado de los elementos 25 y 24 (en este orden) indicando el número de rotaciones y combinaciones que han sido necesarias.**

NOTA: Para borrar un elemento que esté en un nodo interior, se sustituirá por el mayor de su rama izquierda. En el caso de disponer de dos hermanos, se consultará el hermano de la derecha.

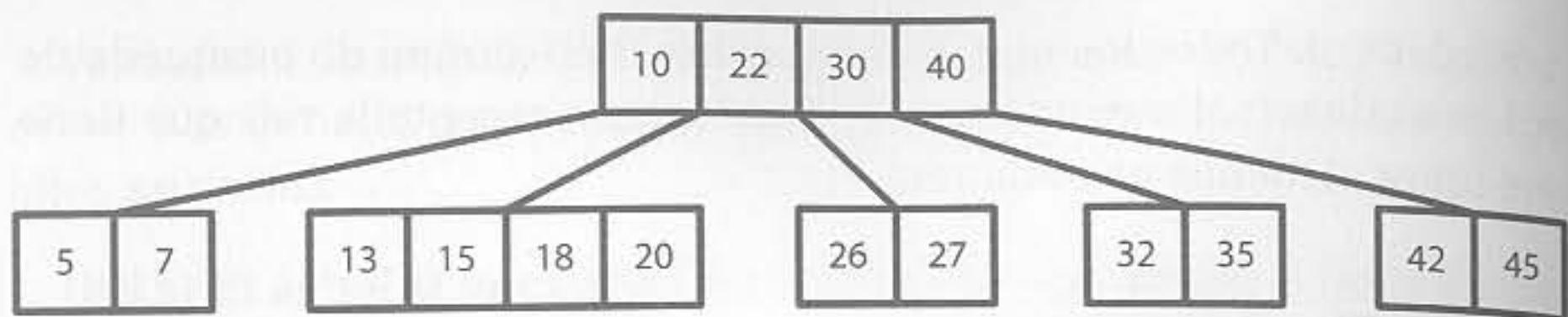


SOLUCIÓN:

A) Borrado del elemento 25: 1 rotación.



B) Borrado del elemento 24: 2 combinaciones y eliminar la raíz del árbol.



# 6

## Tipo conjunto

Los conjuntos constituyen la estructura básica que fundamenta las matemáticas. Los conjuntos se utilizan como base para construir gran cantidad de tipos abstractos de datos que se emplean en diversas técnicas de diseño de algoritmos.

Un **conjunto** se define como una colección de elementos, cada uno de los cuales puede ser a su vez un conjunto o un dato primitivo que recibe el nombre de *átomo*. Todos los elementos del conjunto son distintos y el orden de los mismos no es importante.

Por lo general, cuando los átomos se usan como herramientas para el diseño de algoritmos y estructuras de datos suelen ser elementos sencillos como enteros, caracteres o cadenas, pero todos del mismo tipo. A menudo, se hará la suposición de que existe una relación de orden entre los átomos, con lo que se puede definir la relación sucesor y predecesor de un elemento. Aunque se pueda establecer una relación de orden entre los átomos del conjunto, conviene resaltar las diferencias entre un conjunto y una lista, como es la no importancia de esa relación de orden, y el que cada elemento sólo puede aparecer una vez en el conjunto.

La notación habitual de un conjunto de átomos se representa encerrando sus miembros entre llaves, como en el caso de  $\{1, 3, 5, 7\}$  que denota el conjunto cuyos miembros son los átomos 1, 3, 5, 7. Algunas veces se representan los conjuntos mediante una expresión de la forma  $\{x / \text{proposición sobre } x\}$ , donde la proposición sobre  $x$  es un predicado que di-

ce con exactitud qué se necesita para que un objeto arbitrario  $x$  pertenezca al conjunto. Por ejemplo, la siguiente representación de conjunto  $\{x/x \text{ es un número natural y } x < 2000\}$  es equivalente al conjunto  $\{1, 2, 3, \dots, 1999\}$ .

## 6.1. Representación de conjuntos

### 6.1.1. Preguntas

**(P.89) La representación de conjuntos mediante vectores de bits tiene una complejidad espacial proporcional al tamaño del conjunto universal.**

Verdadero. La complejidad espacial sería de  $O(n)$  siendo  $n$  el tamaño del vector de bits, ya que se necesitaría una casilla del vector para cada posible elemento del conjunto.

**(P.90) La complejidad temporal en su peor caso de la búsqueda de un elemento en un conjunto de cardinalidad  $n$ , representado como una lista, es  $O(n)$ .**

Verdadero. La complejidad temporal sería de  $O(n)$  ya que el peor caso sería el buscar el elemento que está en el último nodo de la lista (o que no apareciese en la lista el elemento a buscar), y para acceder al mismo habría que pasar por todos los anteriores, por lo que se necesitarían  $n$  accesos (uno por cada nodo de la lista).

**(P.91) Cuando utilizamos una tabla de dispersión cerrada de tamaño  $B$ , el número de elementos del conjunto está limitado a  $B - 1$  elementos.**

Falso. En la dispersión cerrada, los elementos se almacenan en una tabla de tamaño fijo  $B$ , y la distribución se realiza mediante una función de dispersión y de redispersión. Por ello, el número máximo de elementos que se podrían almacenar estaría limitado al valor de  $B$ .

**(P.92) La dispersión cerrada con estrategia de redispersión aleatoria tiene la siguiente función de redispersión:  $h_i(x) = (h_{i-1}(x) + C)MODB$ .**

Verdadero. La función de redispersión que se presenta, el intento actual,  $h_i(x)$ , está en función del intento anterior,  $h_{i-1}(x)$ , añadiéndole una

constante,  $C$ . Este tipo de dispersión aunque es mejor que la redispersión lineal, sigue produciendo "amontonamiento", ya que el siguiente intento sólo depende del anterior.

**(P.93) En la dispersión abierta sólo se producen colisiones entre claves sinónimas.**

Verdadero. Esto es así ya que las claves no sinónimas están almacenadas en listas independientes, por lo que para buscar una clave sólo se consultarán (es decir, sólo se producirán colisiones) sus sinónimas ya almacenadas en la tabla (en la misma lista).

**(P.94) En los conjuntos representados como listas ordenadas, la complejidad temporal en su peor caso de la operación 'diferencia de conjuntos' puede ser de  $O(n)$ .**

Verdadero. Suponiendo que las dos listas tienen el mismo número de elementos,  $n$ , el algoritmo que consigue la complejidad lineal  $O(n)$  recorrería ambas listas simultáneamente ya que están ordenadas, accediendo una sola vez a cada uno de los nodos de cada lista ( $2n$  comparaciones), por lo que la cota superior de complejidad sería ésta.

**(P.95) El factor de carga de la dispersión abierta siempre está entre 0 y 1.**

Falso. El factor de carga nos mide el grado de ocupación de la tabla, representado como el cociente entre el número de elementos almacenados y el tamaño de la tabla. En el caso de la dispersión abierta se pueden almacenar un número de elementos mayor al tamaño de la tabla (ya que se trata de un vector de listas), por lo que este cociente puede ser mayor que 1.

**(P.96) Sea la dispersión cerrada con la siguiente función de redispersión:  $h_i(x) = (h_{i-1}(x) + C)MODB$ . Dos claves sinónimas ( $x, y$ ) tendrán la misma secuencia de intentos, es decir,  $h_i(x) = h_i(y)$  para cualquier valor de  $C$  y  $B$ .**

Verdadero. El ser claves sinónimas significa que:

$$H(x) = H(y) = h_0(x) = h_0(y)$$

es decir el primer intento. Con ello, los siguientes intentos tendrán el mismo valor para la fórmula:

$$h_i(x) = (h_{i-1}(x) + C)MODB$$

**(P.97)** En una tabla de dispersión cerrada con la siguiente función de redispersión para la clave 14:  $h_i(14) = (28 + 2 * i) MOD 2000$ , se recorrerán todas las posiciones de la tabla buscando una posición libre.

Falso. Según la función de redispersión, tendremos que  $C = 2$  y que  $B = 2000$ , y dado que  $C$  y  $B$  tienen números primos comunes mayores que 1, se darán casos en que no se recorran todas las posiciones de la tabla (en el caso de redispersión aleatoria). Si hablamos de redispersión con una segunda función hash,  $B$  debe ser primo y tampoco lo cumple.

**(P.98)** En la búsqueda de elementos en una tabla de dispersión cerrada, hay que distinguir durante la búsqueda las casillas vacías de las suprimidas.

Verdadero. Una de las condiciones de finalización del proceso de búsqueda de un elemento es la de encontrar una casilla vacía (en la que no se ha realizado ninguna inserción). El que se encuentre una casilla suprimida (en la que se ha producido el borrado del elemento que contenían) podría identificar el siguiente caso. Supongamos que una clave  $x$  haya tenido que ser insertada en un lugar de la tabla diferente al que le correspondía a causa de una colisión con un elemento  $y$ . Esto implicaría que al realizar la búsqueda de  $x$ , habría que pasar por la clave que colisiona con ella ( $y$ ). Pero imaginemos el caso que  $y$  se borrase, ésto supondría que la posición  $H(x)$  estaría vacía, con lo que al realizar la búsqueda de  $x$  concluiríamos que no se encuentra en el conjunto, cosa no necesariamente verdadera. Durante la inserción, las casillas suprimidas se tratarán del mismo modo que las vacías, es decir, como espacio disponible.

**(P.99)** Sea una tabla de dispersión cerrada con función de dispersión  $H(x) = x MOD B$ , con  $B = 100$  y  $x$  un número natural entre 1 y 2000. Sólo hay un valor de  $x$  que haga  $H(x) = 4$ .

Falso. Dado que  $x$  puede estar entre 1 y 2000, esto significa que todos los elementos se dividirán en 100 clases de equivalencia (el tamaño  $B$  de la tabla) determinados por la función de dispersión  $H(x) = x MOD B$ . Cada una de estas clases de equivalencia tendrá más de un elemento, por lo que habrá más de un valor de  $x$  que haga  $H(x) = 4$ , como por ejemplo  $x = 4, 104, 204$ .

**(P.100)** Cuando implementamos un TAD tabla de dispersión cerrada se usa una función de dispersión  $H$  tal que  $H(x)$  devolverá un valor desde 0 hasta  $B$ , siendo  $B$  el número finito de clases en las que dividimos el conjunto.

Falso.  $H(x)$  devolverá un valor desde 0 y  $B - 1$ , o bien, entre 1 y  $B$ , es decir, habrán  $B$  clases o posiciones en la tabla.

**(P.101)** En la dispersión cerrada sólo se producen colisiones entre claves sinónimas.

Falso. Puede darse la situación en la que  $x$  e  $y$  no sean claves sinónimas, pero que  $x$  haya colisionado con otra clave  $y$  y pase a ocupar una posición en la tabla que no le corresponde según la función de dispersión,  $H(x)$ , la cual podría ser la posición  $H(y)$ , por lo que al almacenar  $y$  se produciría una colisión entre ambas.

**(P.102)** La complejidad temporal en su peor caso de las operaciones de inserción y búsqueda en un árbol de búsqueda digital están en función del número de bits de la clave.

Verdadero. En las operaciones de inserción y búsqueda se realiza un recorrido descendente desde la raíz a las hojas. En cada una de las iteraciones se realizará una comparación con el elemento almacenado en cada nodo de este recorrido. Por ello la complejidad quedará en función de la longitud de este recorrido (o dicho de otro modo, la altura del árbol), el cual está limitado por el número de bits de la clave a buscar.

**(P.103)** El proceso de búsqueda en un árbol de búsqueda digital es igual que el del árbol binario de búsqueda.

Falso. El proceso de búsqueda de una clave en un árbol de búsqueda digital es básicamente igual que en los árboles de búsqueda binarios, excepto en que el subárbol al que hay que moverse durante el proceso de búsqueda viene determinado por un bit en la clave. En el caso del árbol de búsqueda, la elección del subárbol por el que se continúa la búsqueda viene determinado por la relación de orden existente entre los miembros del conjunto.

**(P.104)** La representación del nodo de un trie utilizando un vector de punteros es más eficiente temporalmente que utilizar una lista de punteros.

Verdadero. Utilizando un vector habría una función que dada una letra de la cadena devolvería la posición en la que se almacena el puntero en

un solo paso; con la lista habría que realizar un recorrido secuencial para encontrar la letra. Por ello, utilizando un vector sería la representación más eficiente desde el punto de vista temporal.

**(P.105) La complejidad temporal en su caso mejor de la operación de búsqueda en un trie con nodos terminales es de  $\Omega(1)$ .**

Verdadero. Se daría en el caso de que esté almacenada una cadena que no comparta ningún prefijo con ninguna otra cadena del trie, por lo que del nodo raíz colgaría directamente un nodo terminal que la almacene. Con ello en un solo paso se realizaría la búsqueda.

**(P.106) La altura de un trie con nodos terminales será como mínimo la longitud de la cadena más larga almacenada.**

Falso. Esa sería la altura máxima. La altura mínima sería cuando la cadena no comparte ningún prefijo con otra cadena del trie, por lo que del nodo raíz colgaría directamente un nodo terminal que la almacene.

### 6.1.2. Ejercicios

**(E.34) Sea una tabla de dispersión cerrada de tamaño  $B = 11$  inicialmente vacía.**

**A) Insertar los siguientes elementos (en este orden estricto):**

23, 14, 10, 15, 3, 5, 7, 8, 36, 47, 4

La estrategia de redispersión a utilizar es la que usa una segunda función de dispersión:  $K(x) = (x \text{MOD}(B - 1)) + 1$ .

**B) ¿Qué valor debe tomar  $B$  para que se asegure que la estrategia de redispersión permita explorar todas las posiciones de la tabla?**

SOLUCIÓN:

A) La función de dispersión es:  $H(x) = x \text{MOD} B$

La función de redispersión es:  $hi(x) = (H(x) + i * K(x)) \text{MOD} B$  siendo la segunda función  $K(x) = (x \text{MOD}(B - 1)) + 1$

0	47
1	23
2	4
3	14
4	15
5	5
6	36
7	3
8	8
9	7
10	10

B)  $B$  debe ser un número primo.

## 6.2. Unión-búsqueda

### 6.2.1. Preguntas

**(P.107) Sea el TAD Unión-Búsqueda implementado por medio de vectores de bits. La operación**

$\text{busqueda}(\text{Elemento}) \rightarrow \text{Conjunto}$

que nos devuelve todos los elementos del conjunto al que pertenece  $\text{Elemento}$ , tiene una complejidad  $O(k)$ , siendo  $k$  el número de conjuntos en el TAD.

Falso. Si el objetivo de la operación es devolver cada uno de los elementos del conjunto, entonces la complejidad quedaría en función del tamaño del vector (el del conjunto universal). Si el objetivo es devolver sólo el identificador del conjunto, entonces sí sería una complejidad en función de  $k$ .

**(P.108) Sea el TAD Unión-Búsqueda implementado por medio de vectores de elementos. La operación**

$\text{union}(\text{Identif}, \text{Identif}) \rightarrow \text{Identif}$

tiene una complejidad  $O(n)$ , siendo  $n$  el tamaño del vector.

Verdadero. Se necesitaría recorrer cada posición del vector para incluir el identificador de la unión en cada casilla en que apareciese el identificador de uno de los dos conjuntos que se unen.

**(P.109) Sea el TAD Unión-Búsqueda implementado por medio de árboles representados como vectores. La operación**

*union(Conjunto, Conjunto) → Conjunto*

**tiene una complejidad  $O(1)$ .**

Falso. Esta complejidad constante se daría en el caso de la operación

*union(Identif, Identif) → Identif*

ya que se devolvería solo el identificador del conjunto resultado.

**(P.110) Sea el TAD Unión-Búsqueda implementado por medio de árboles representados como vectores aplicando la regla compensada para la unión. En: *union(i, j)*, el identificador *i* quedará como raíz del nuevo árbol, si su árbol tiene menos elementos que el árbol formado por el identificador *j*.**

Falso. Es al revés, quedará *j* como raíz del nuevo árbol.

**(P.111) Sea el TAD Unión-Búsqueda implementado por medio de vectores de elementos. La operación**

*union(Identif, Identif) → Conjunto*

**que nos devuelve todos los elementos del conjunto unión, tiene una complejidad  $O(n)$ , siendo  $n$  el tamaño del vector.**

Verdadero. Se necesitaría recorrer cada posición del vector, para incluir el identificador de la unión en cada casilla en que apareciese el identificador de uno de los dos conjuntos que se unen, y mientras tanto crear el conjunto resultado.

**(P.112) Sea el TAD Unión-Búsqueda implementado por medio de árboles representados como vectores. Estos árboles son m-arios con  $m$  la cardinalidad del conjunto universal menos uno.**

Verdadero. El número máximo de hijos de un nodo se daría cuando sólo hay un conjunto que almacena a todos los elementos del conjunto universal, de modo que la raíz del único árbol tendría como hijos a todos los demás elementos del TAD.

**(P.113) En un TAD Unión-Búsqueda, sus elementos son conjuntos disjuntos entre sí.**

Verdadero. Según la propia definición del TAD Unión-Búsqueda, en este tipo de datos no se van a hacer inserciones o borrados de elementos, excepto las inserciones necesarias para crear el conjunto la primera vez, ya que las operaciones habituales son:

- **UNIÓN:** devuelve conjunto resultado de la unión de dos conjuntos disjuntos.
- **BÚSQUEDA:** devuelve el identificador del conjunto al que pertenece el elemento que se busca.

Por ejemplo, el siguiente conjunto *A* es del tipo UNIÓN-BÚSQUEDA:

$A = 1, 2, 3, 4, 5, 6, 7, 8, 9$

**(P.114) Sea el TAD Unión-Búsqueda implementado por medio de árboles representados como vectores. Estos árboles han de ser binarios.**

Falso. Serían árboles m-arios con  $m$  la cardinalidad del conjunto universal menos uno.

## 6.3. Cola de prioridad

### 6.3.1. Preguntas

**(P.115) En el proceso de inserción en un montículo máximo insertaremos en la siguiente posición libre para que siga siendo un árbol binario lleno.**

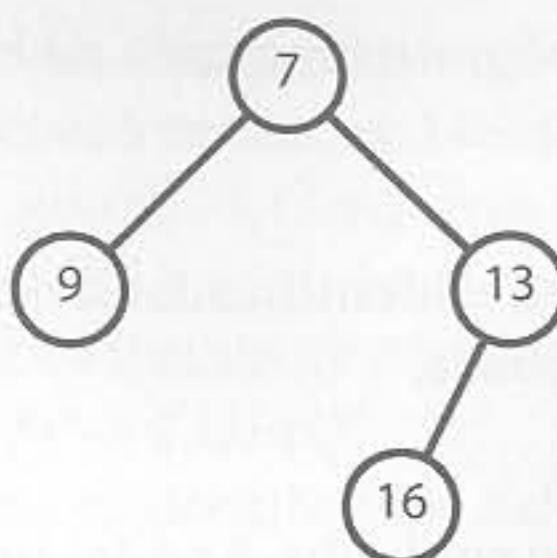
Falso. Se inserta en la siguiente posición libre para que siga siendo un árbol binario completo.

**(P.116) Todo montículo mínimo es un árbol binario lleno que además es árbol mínimo.**

Falso. Se trata de un árbol binario completo que además es árbol mínimo.

**(P.117) Para todo nodo de un árbol leftist, se cumple que la altura de su hijo izquierdo es mayor o igual que la de su hijo derecho.**

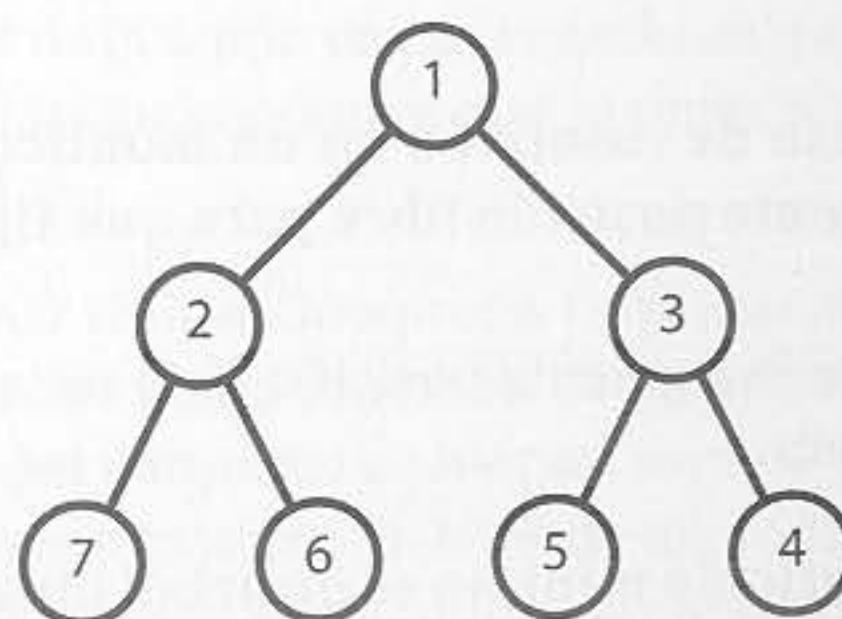
Falso. Lo que se exige es referente al camino mínimo y no a la altura de cada hijo. Se pueden dar casos en los que no se cumpla respecto a la altura. El siguiente contraejemplo contradice esta afirmación:



**(P.118)** En un heap máximo los elementos de las hojas son los que tienen mayores valores en sus claves.

Falso. Son los que tienen menor valor. Esta condición se cumpliría para el montículo mínimo.

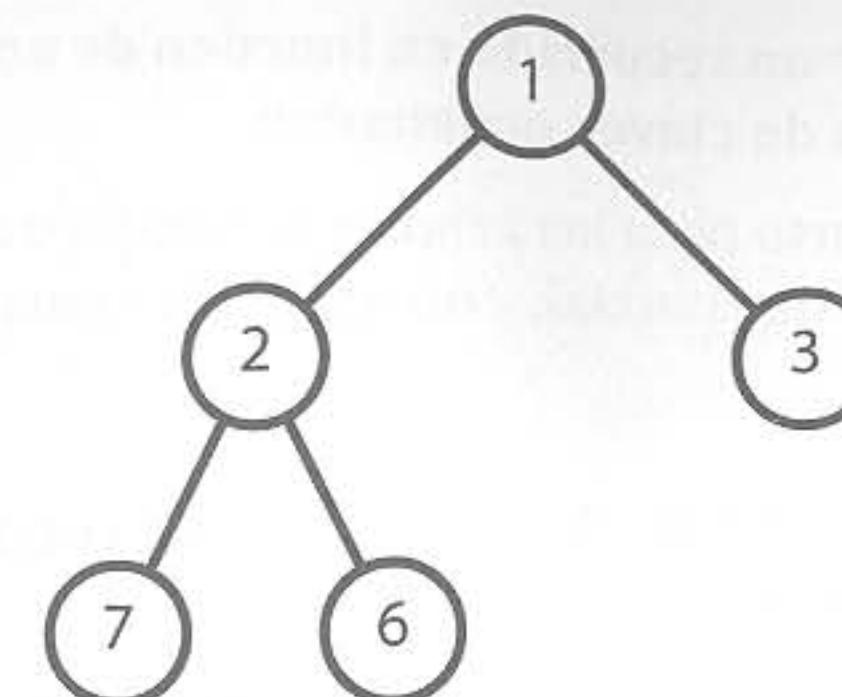
**(P.119)** El siguiente árbol es un montículo mínimo y también es un leftist mínimo:



Verdadero. Cumple las condiciones de montículo mínimo (árbol mínimo y árbol binario completo) y también cumple las de leftist mínimo.

**(P.120)** Para todo nodo de un árbol leftist, se cumple que la altura de su hijo izquierdo es menor que la de su hijo derecho.

Falso. El siguiente contraejemplo contradice esta afirmación:

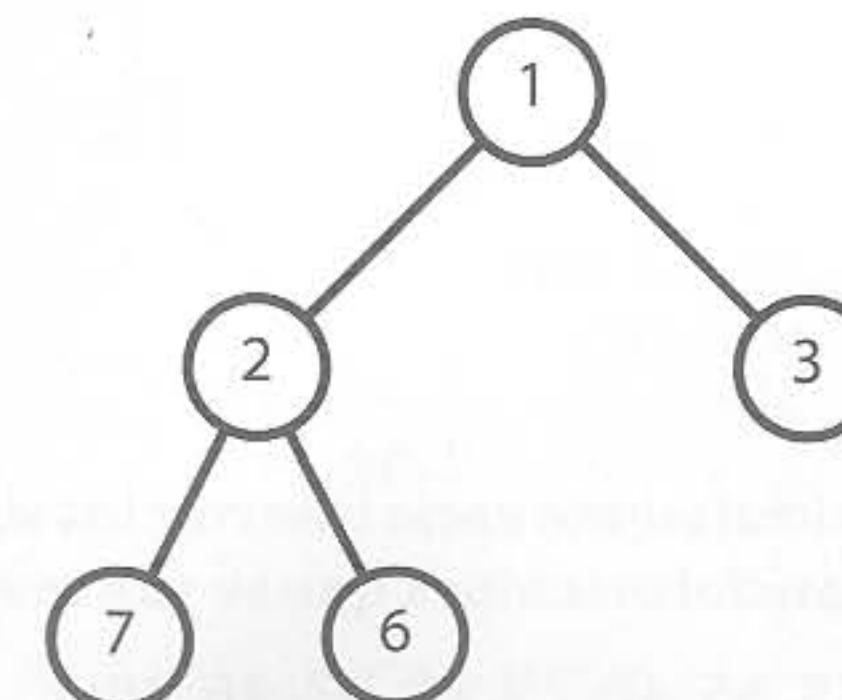


**(P.121)** El montículo o heap mínimo es un árbol binario completo que además es árbol mínimo.

Verdadero. Es la propia definición de montículo mínimo, por lo que la etiqueta del nodo raíz de un árbol mínimo es la más pequeña del conjunto, mientras que la del árbol máximo es la mayor, con lo que se adivina claramente su aplicación para la representación de las colas de prioridad.

**(P.122)** Para todo nodo de un árbol leftist, se cumple que el número de nodos de su hijo izquierdo es menor que el de su hijo derecho.

Falso. El siguiente contraejemplo contradice esta afirmación:



**(P.123)** Un árbol 2-3-4 cumple las condiciones para ser también un árbol leftist.

Falso. En el árbol leftist se exige que sea un árbol binario, condición que no cumplen los árboles 2-3-4.

**(P.124) Al realizar un recorrido en inorden de un montículo obtenemos una sucesión de claves ordenadas.**

Falso. Esto sería cierto para los árboles de búsqueda. El montículo utiliza una relación de orden parcial, y no total, tal y como tienen los árboles de búsqueda.

**(P.125) Un árbol binario lleno cumple las condiciones para ser también un árbol leftist.**

Verdadero. Ya que se cumpliría para cada nodo que el camino mínimo del hijo izquierdo sea mayor o igual que el del hijo derecho.

**(P.126) En un montículo doble de altura  $h$  se pueden almacenar  $2^h - 1$  claves.**

Falso. El número máximo vendría limitado por  $2^h - 2$  ya que en la raíz no se almacena ningún elemento.

**(P.127) Todo árbol binario de búsqueda lleno es un árbol leftist mínimo.**

Falso. La condición de árbol de búsqueda es contradictoria con la condición exigida en los árboles mínimos.

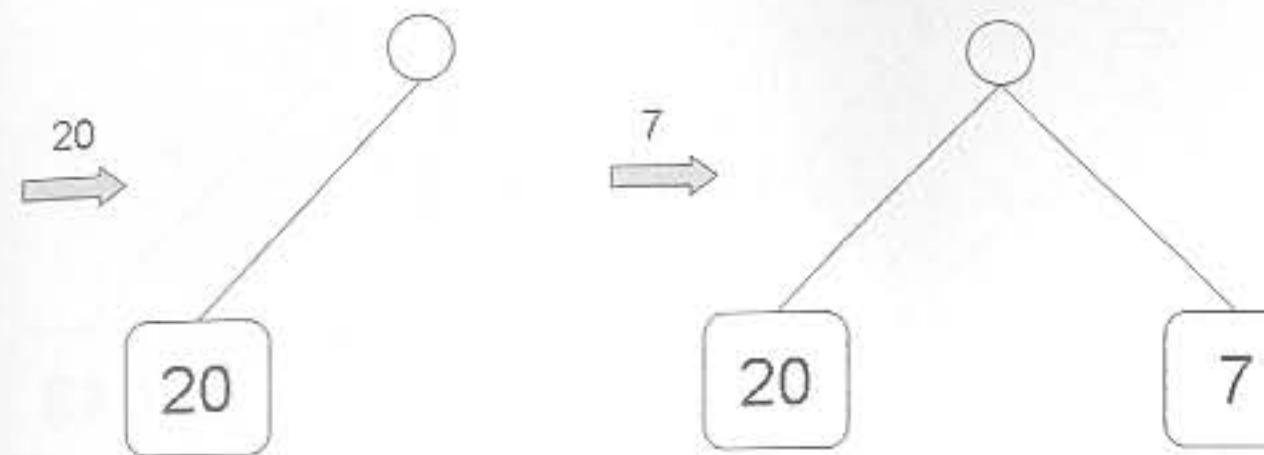
### 6.3.2. Ejercicios

**(E.35) En un DEAP inicialmente vacío insertar los siguientes elementos, indicando las transformaciones que se van realizando:**

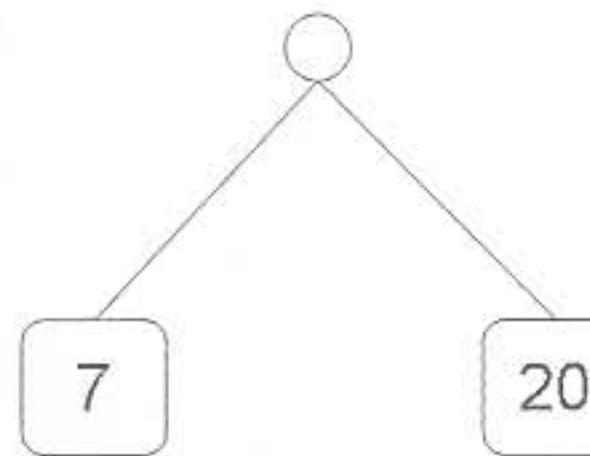
**20, 7, 13, 31, 43, 55, 65, 40, 27, 29, 33, 46, 30, 1**

**SOLUCIÓN:**

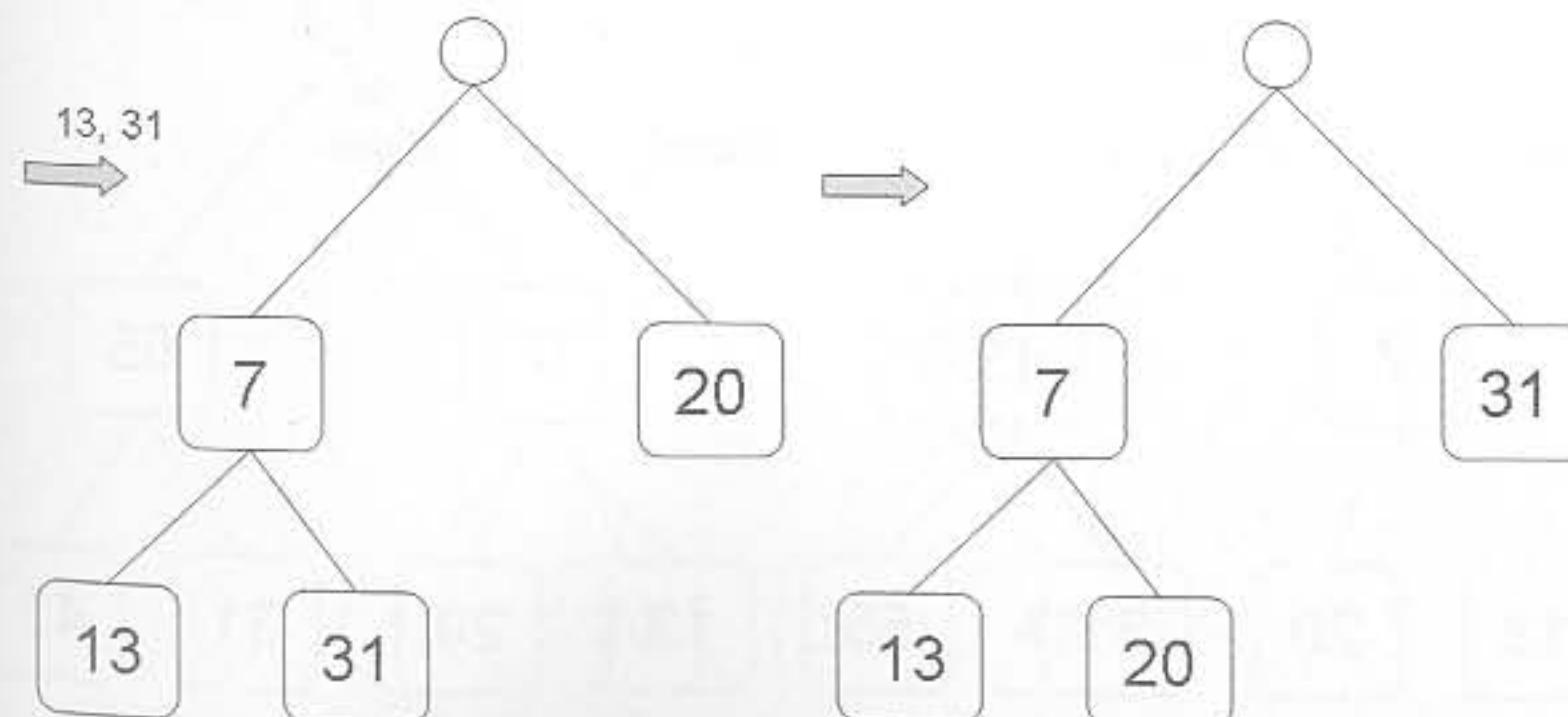
El montículo doble tiene un nodo raíz que no contiene elementos y luego dos montículos, uno a la izquierda que será un montículo mínimo y otro a la derecha que será un montículo máximo. Insertamos en primer lugar los elementos 20 y 7.



Comprobamos la condición e intercambiamos las etiquetas

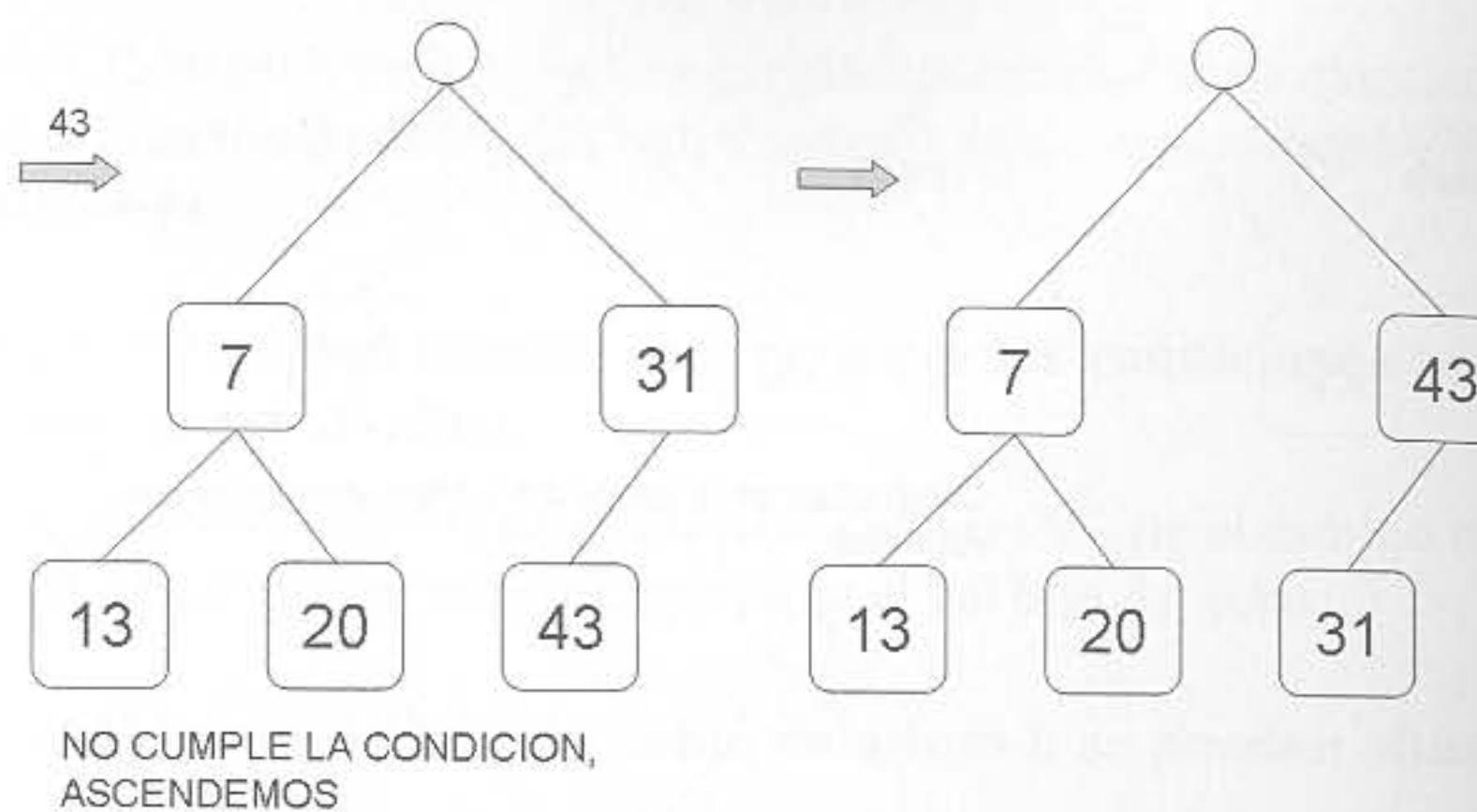


insertamos el 13 y el 31

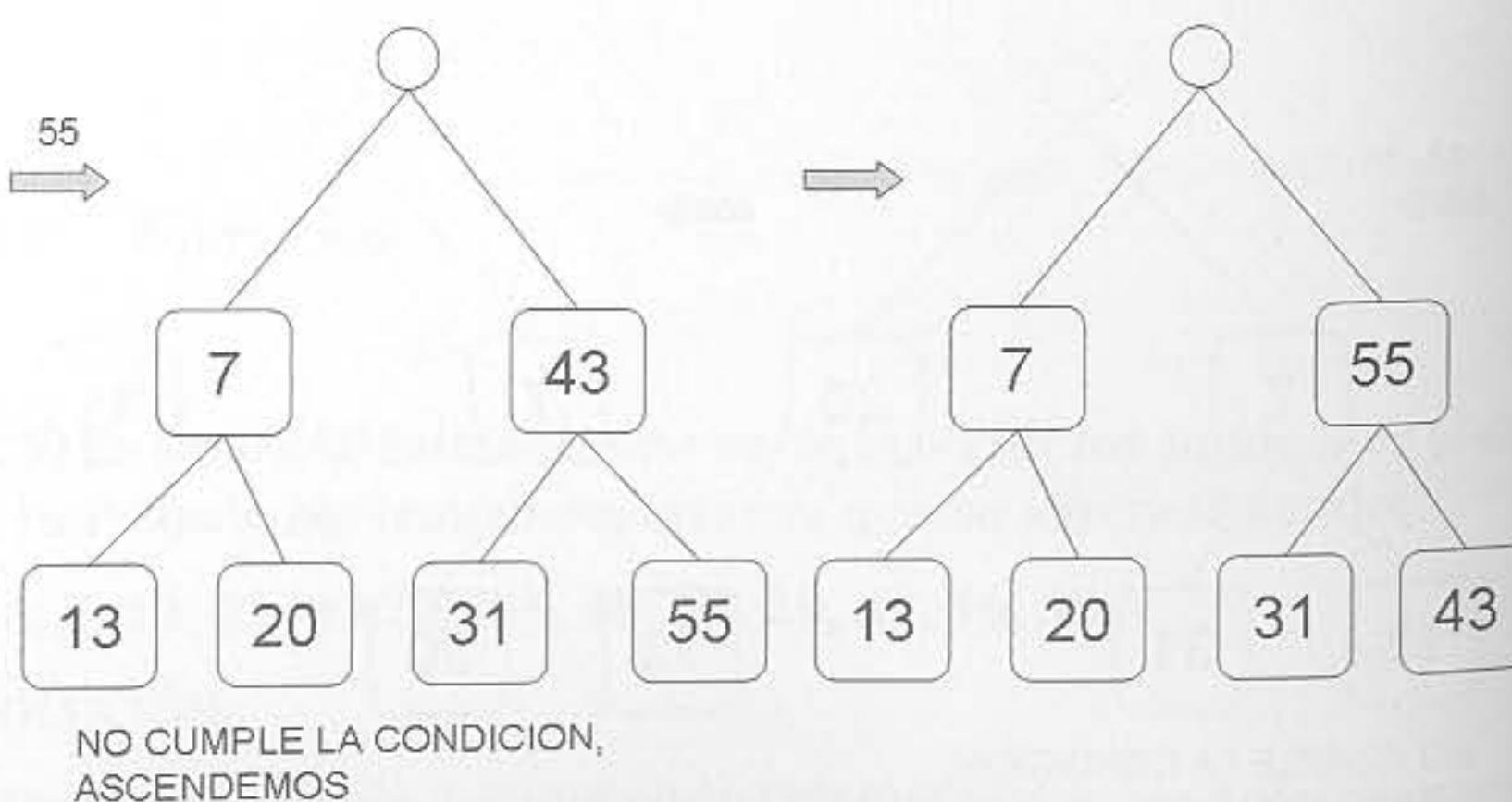


NO CUMPLE LA CONDICION,  
INTERCAMBIAMOS

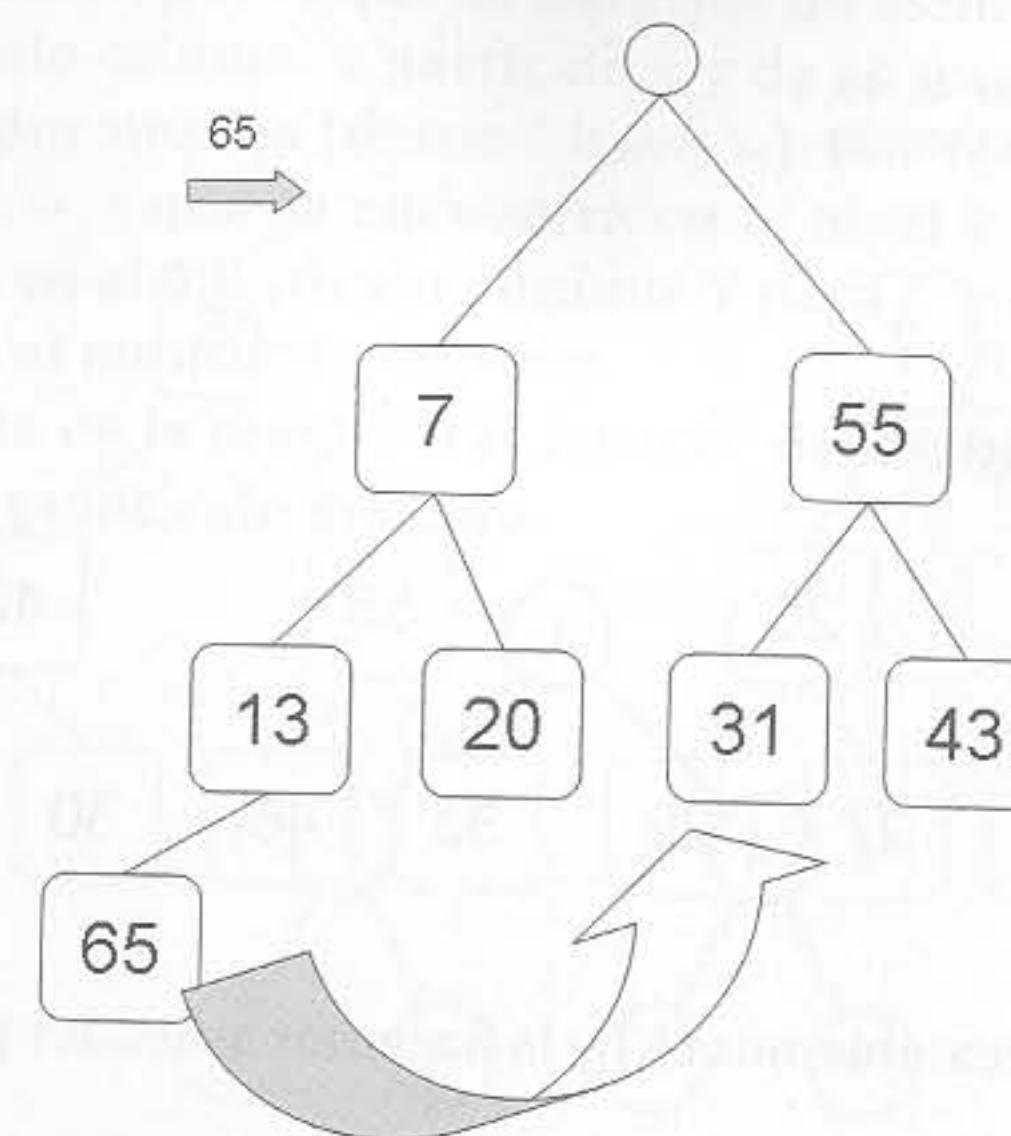
insertamos el 43



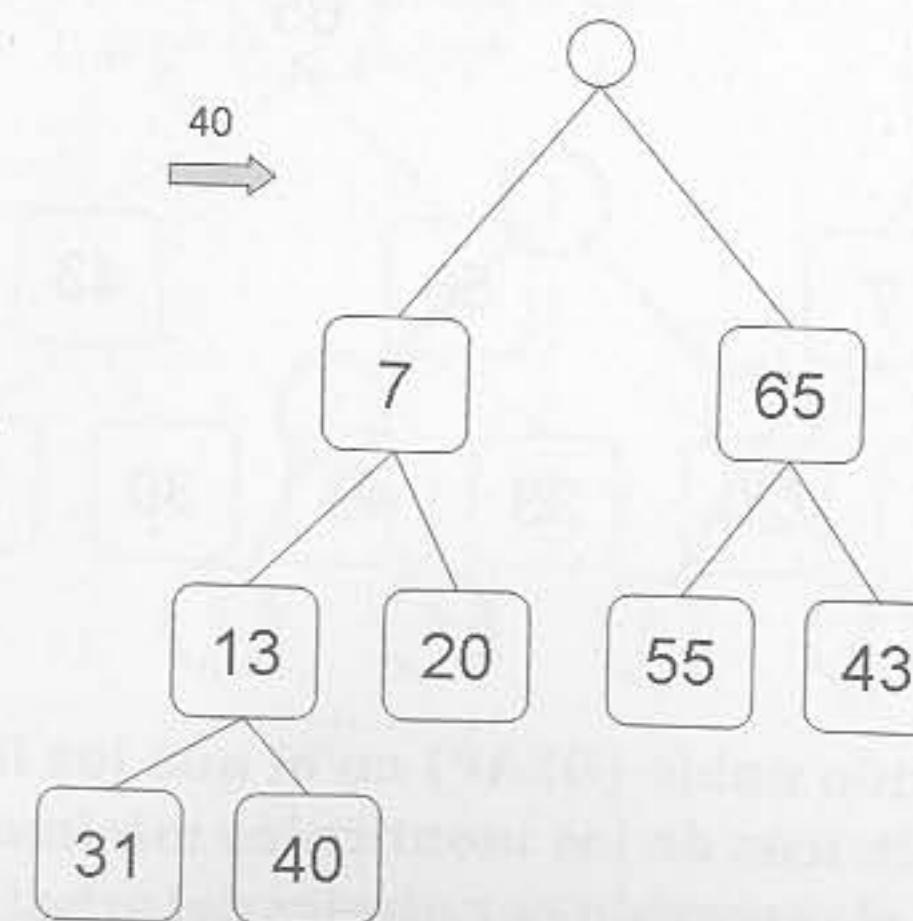
insertamos el 55



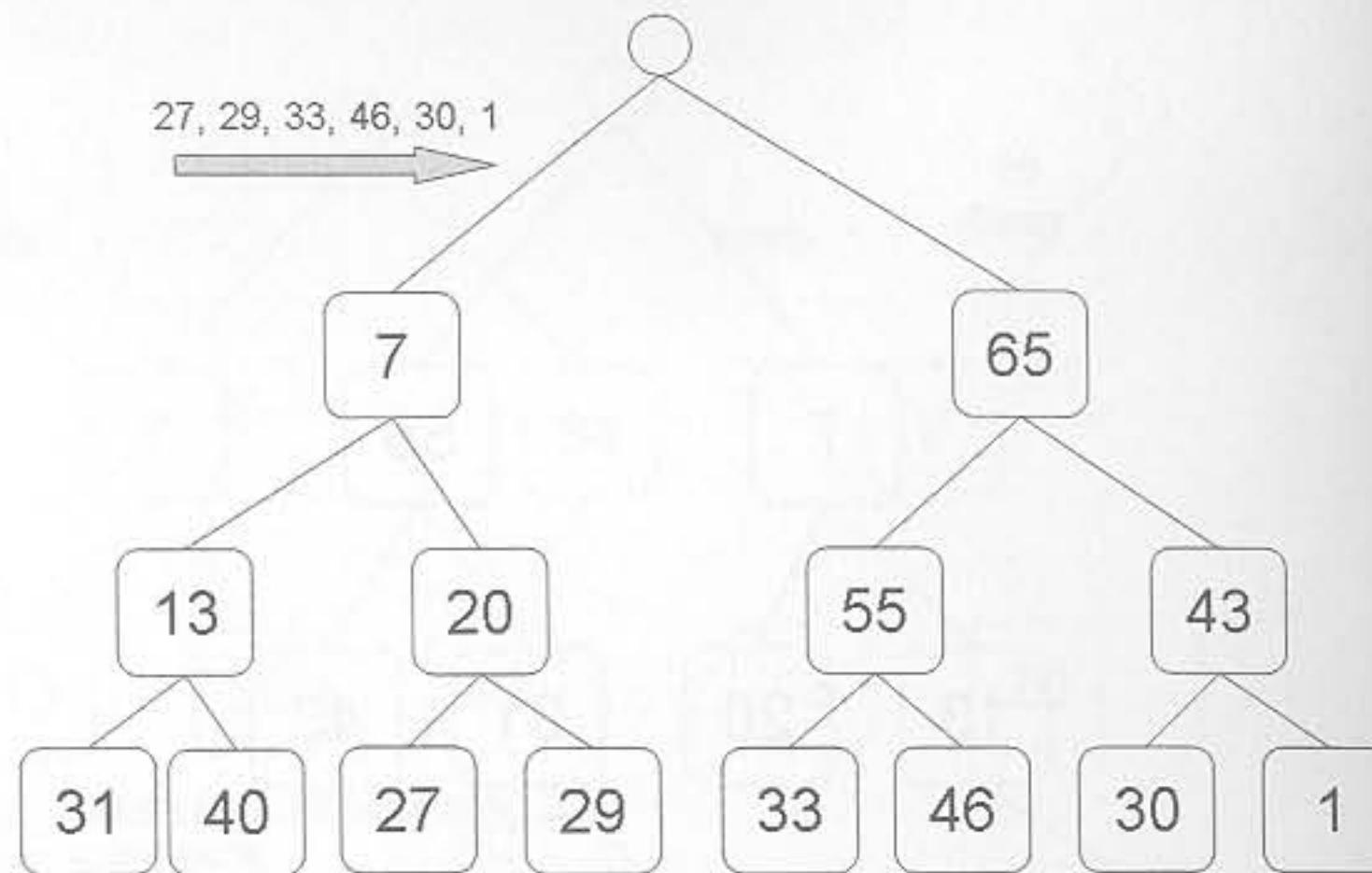
insertamos el 65



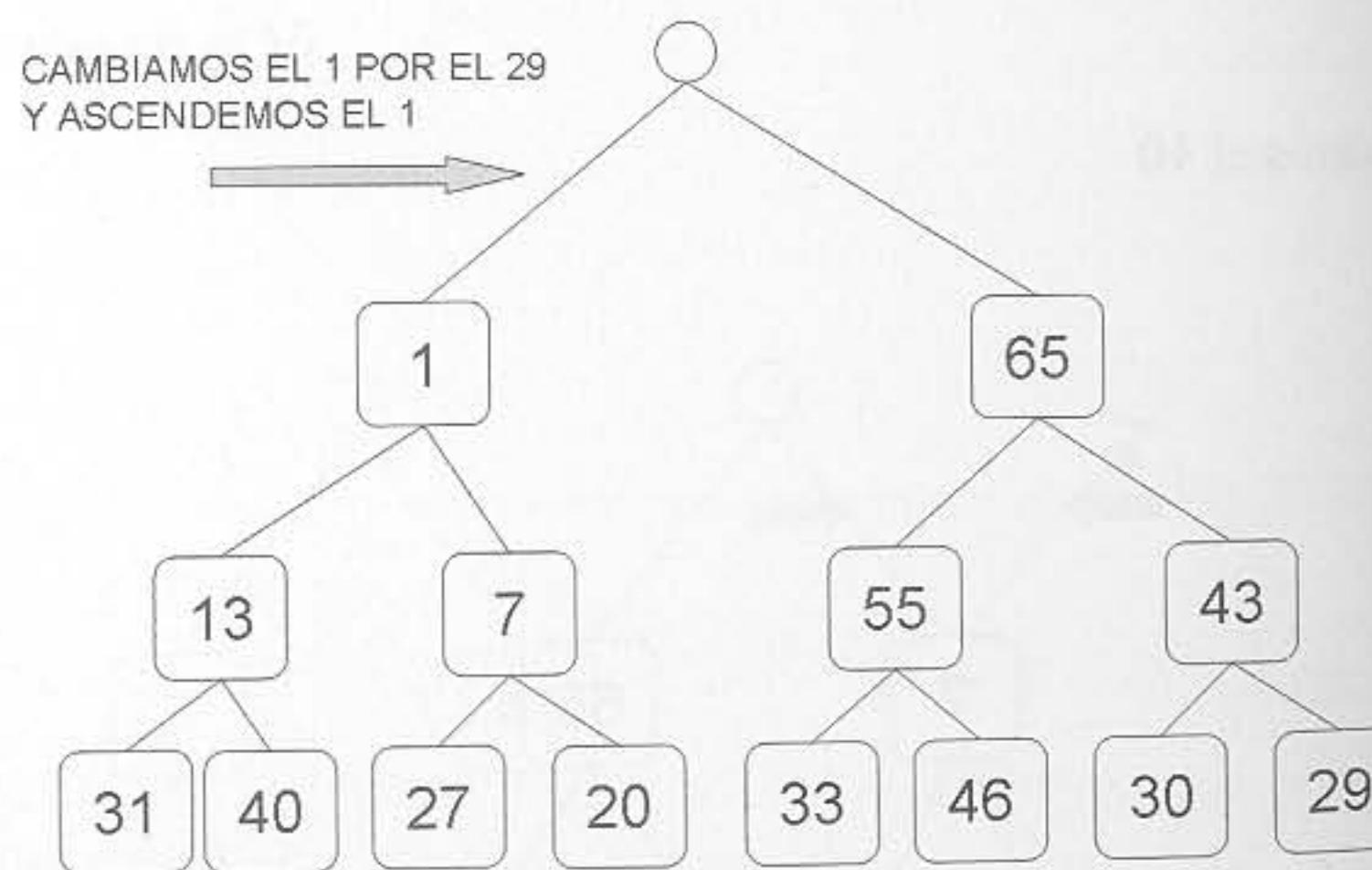
insertamos el 40



insertamos el 27, 29, 33, 46, 30 y el 1

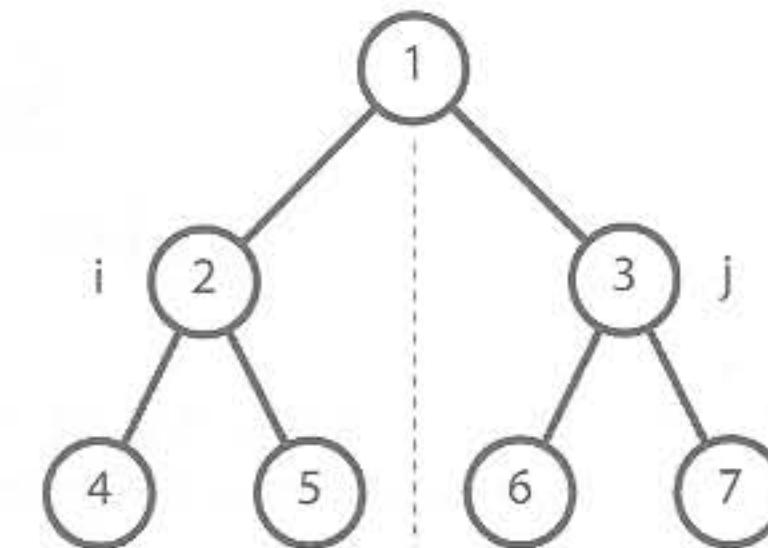


finalmente intercambiamos el 1 y lo hacemos ascender por el montículo mínimo



(E.36) Sea un montículo doble (DEAP) en el que los índices  $i$  y  $j$  referencian nodos simétricos de los montículos mínimo y máximo respectivamente según el recorrido por niveles del árbol (tal y como se muestra en el siguiente árbol, desde 1 hasta  $n$ , siendo  $n$  el número de elementos del DEAP). Suponiendo que  $i$  está en el nivel  $k$  del árbol, obtén razonadamente las siguientes fórmulas:

- A) La que obtiene  $j$  (su nodo simétrico) a partir de  $i$  y de  $k$ .  
 B) La condición por la que se sabe que un elemento se encuentra en el montículo mínimo a partir de  $k$  y de su numeración,  $t$ , según el recorrido por niveles (desde 1 hasta  $n$ ). Por ejemplo, para el nodo número  $t = 4$  que se encuentra en el nivel  $k = 3$ , se sabe que se encuentra en el montículo mínimo. Y para  $t = 6$  sabemos que se encuentra en el montículo máximo.  
 C) Igual que en la pregunta B, a partir de  $k$  saber cuándo  $t$  se encuentra en el montículo máximo.

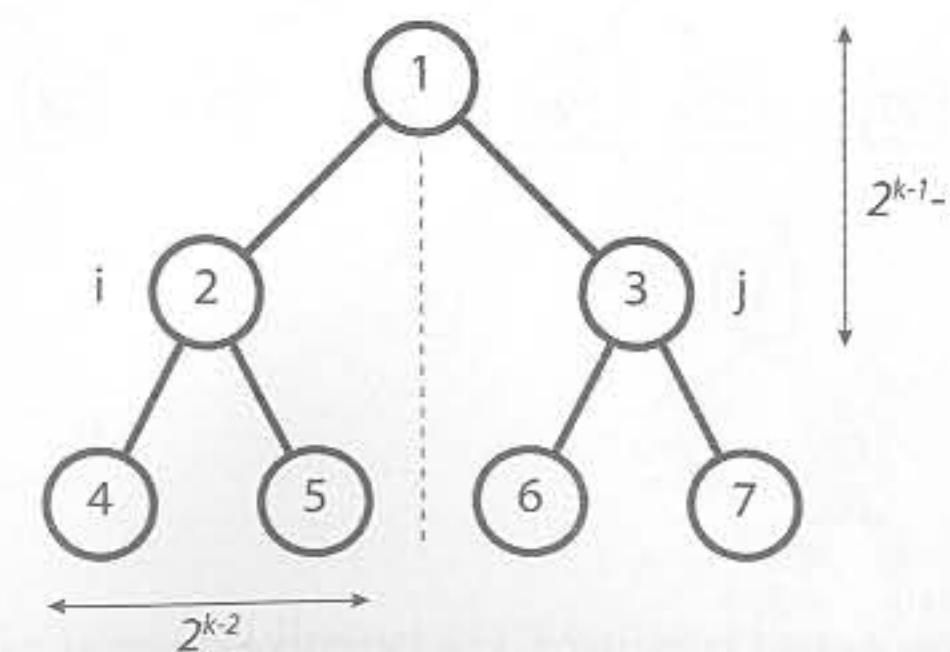


#### SOLUCIÓN:

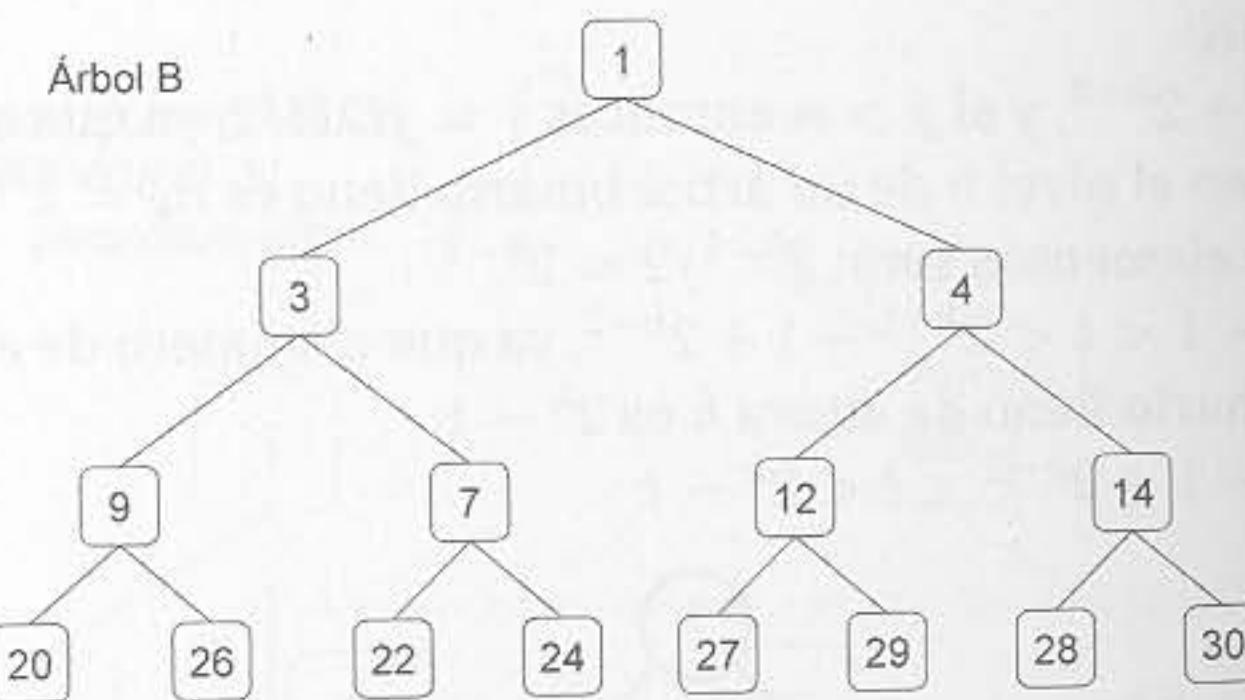
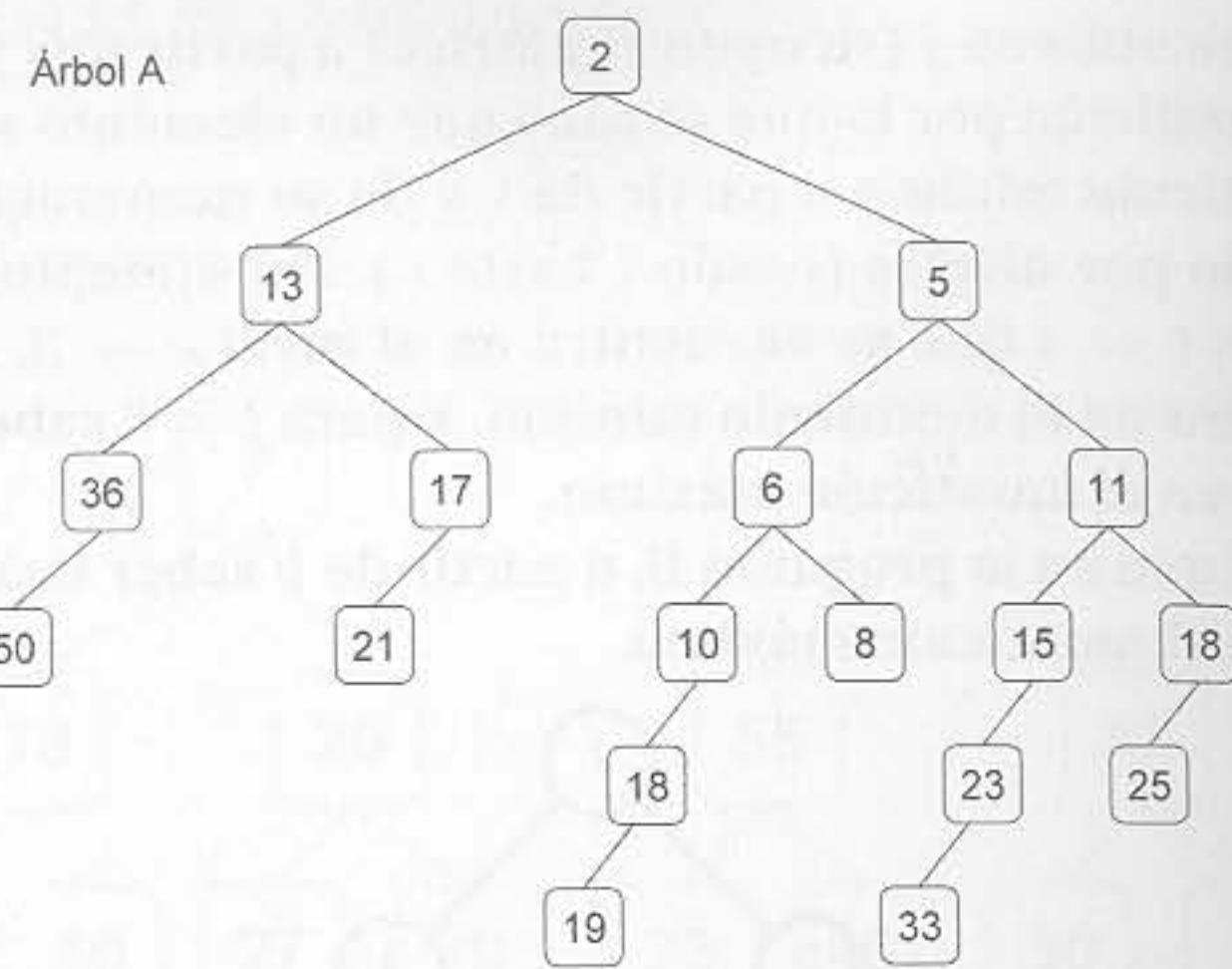
A)  $j = i + 2^{k-2}$ , y si  $j > n$  entonces  $j = j \text{ DIV } 2$ , ya que el número de elementos en el nivel  $k$  de un árbol binario lleno es  $n_k = 2^{k-1}$ . Entonces la mitad de elementos será:  $2^{k-1}/2 = 2^{k-2}$ .

B)  $2^{k-1} - 1 < t < 2^{k-1} - 1 + 2^{k-2}$ , ya que el número de elementos de un árbol binario lleno de altura  $k$  es  $2^k - 1$ .

C)  $2^{k-1} - 1 + 2^{k-2} < t < 2^k - 1$

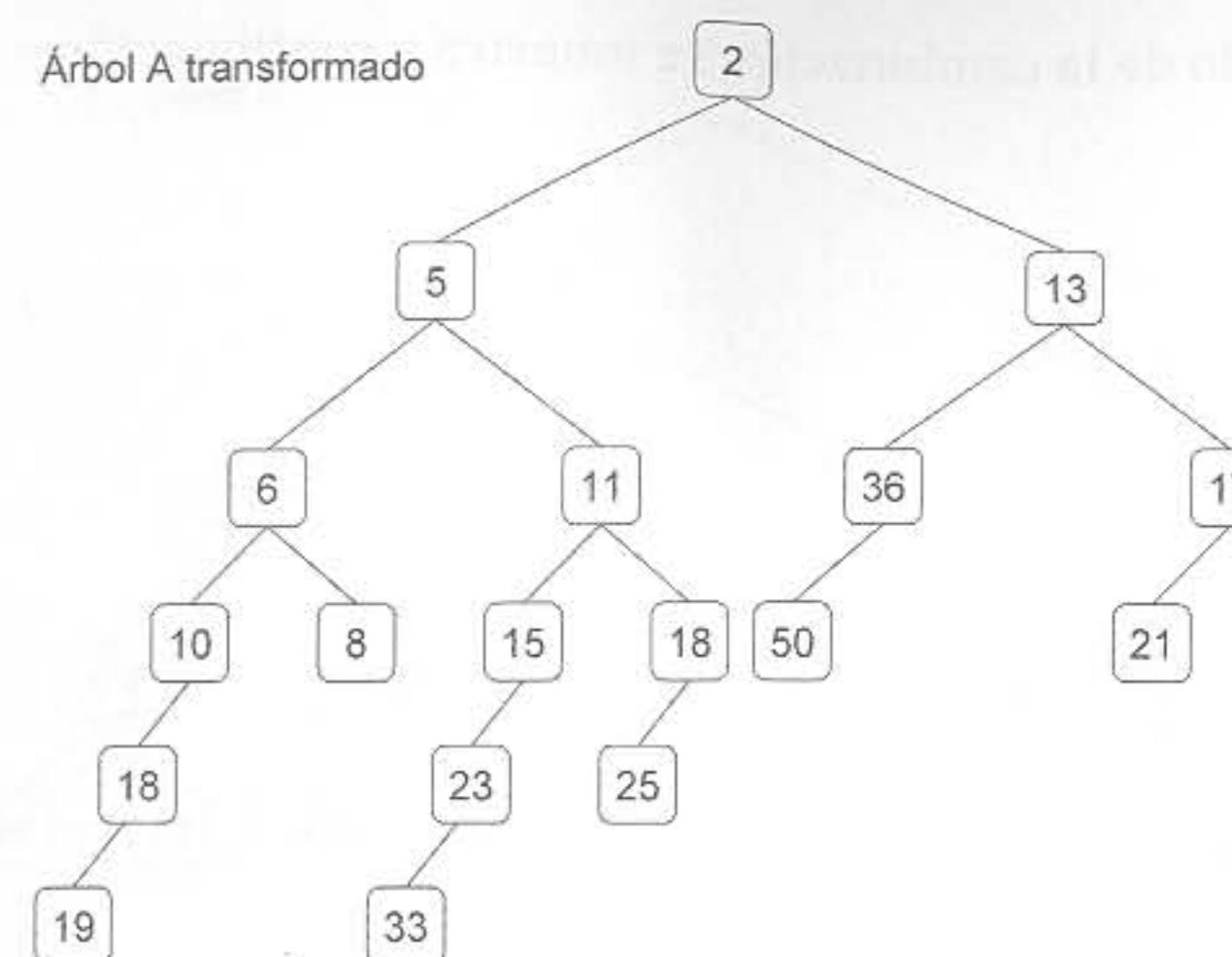


(E.37) Dados los siguientes árboles, comprueba si son izquierdistas mínimos (leftist); en caso negativo realiza las transformaciones necesarias para que lo sean. Realiza la combinación de ambos.



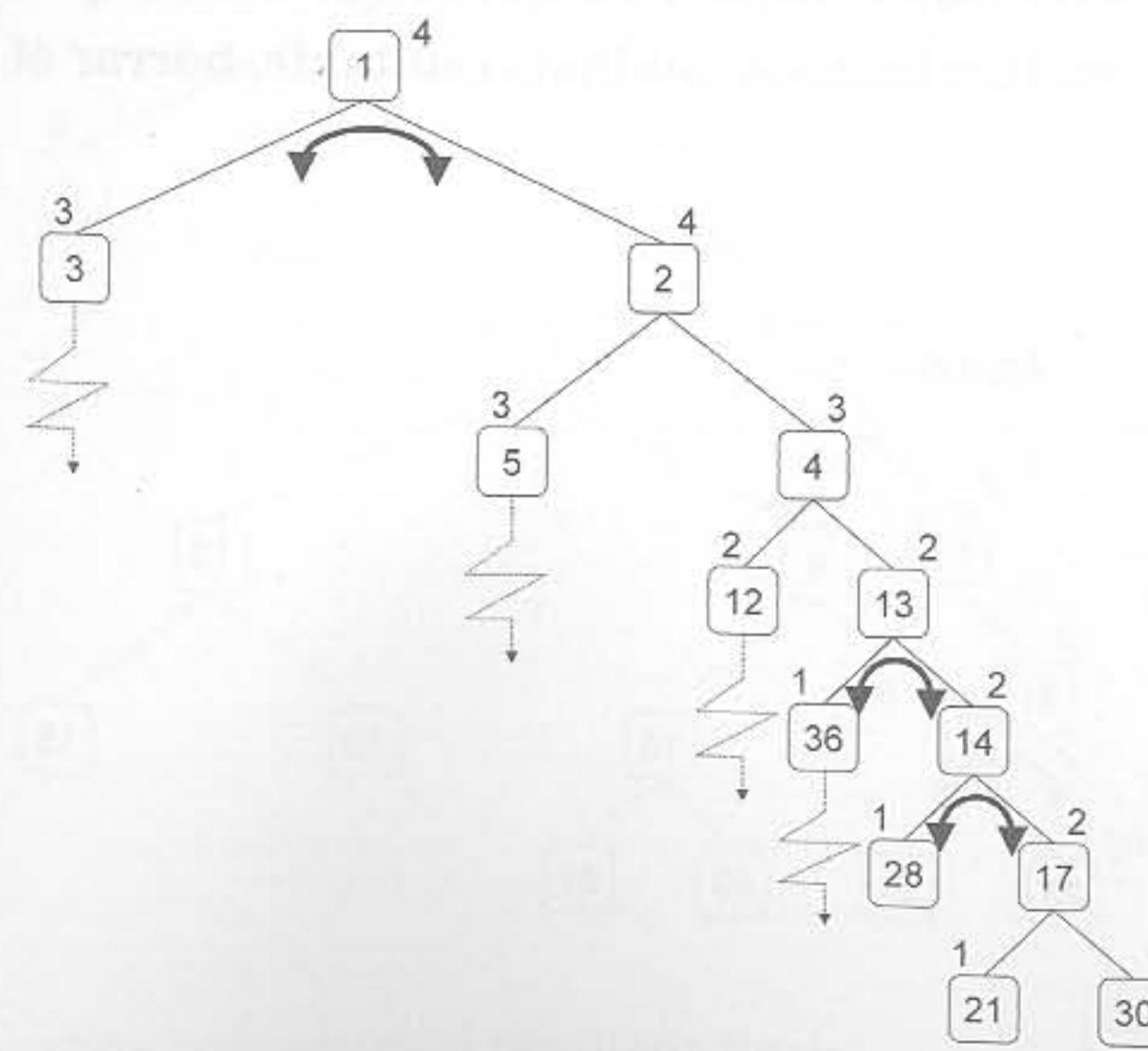
### SOLUCIÓN:

El árbol A no es un árbol izquierdista porque para el nodo con clave 2, el camino mínimo de su hijo derecho es mayor (3) que el de su hijo izquierdo (2). Para solucionar este problema, la transformación a realizar consiste en intercambiar ambos subárboles, quedando tal y como se muestra a continuación:

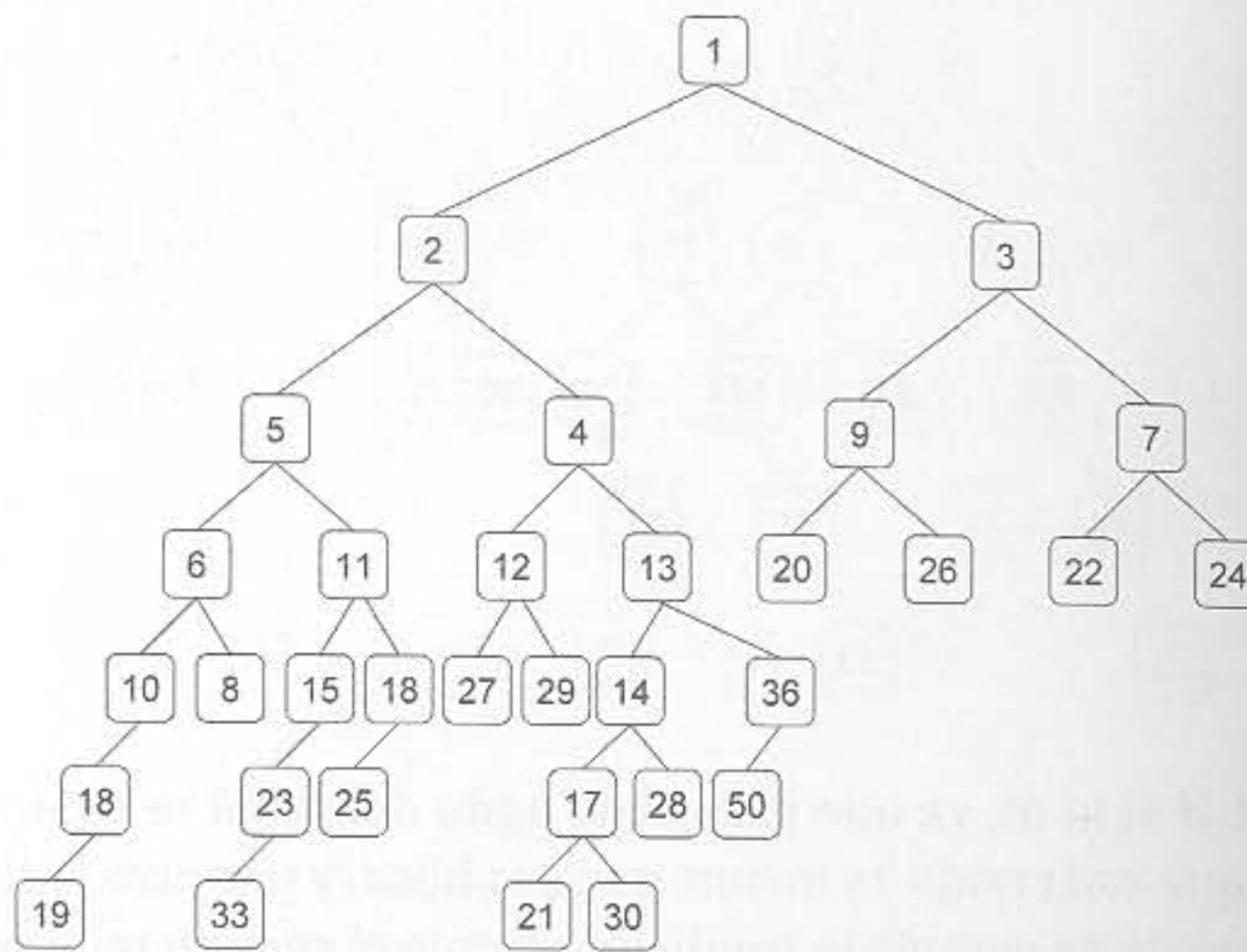


El árbol B sí lo es, ya que para cada nodo del árbol se cumple la condición de que cada nodo es menor que sus hijos, y por otro lado también para cada nodo se cumple la condición de que el camino mínimo del hijo izquierdo es mayor o igual que el del hijo derecho.

En la próxima figura se muestra el resultado intermedio de la combinación, en el que aparecen los caminos mínimos encima de los nodos y están indicados los intercambios a realizar entre subárboles.

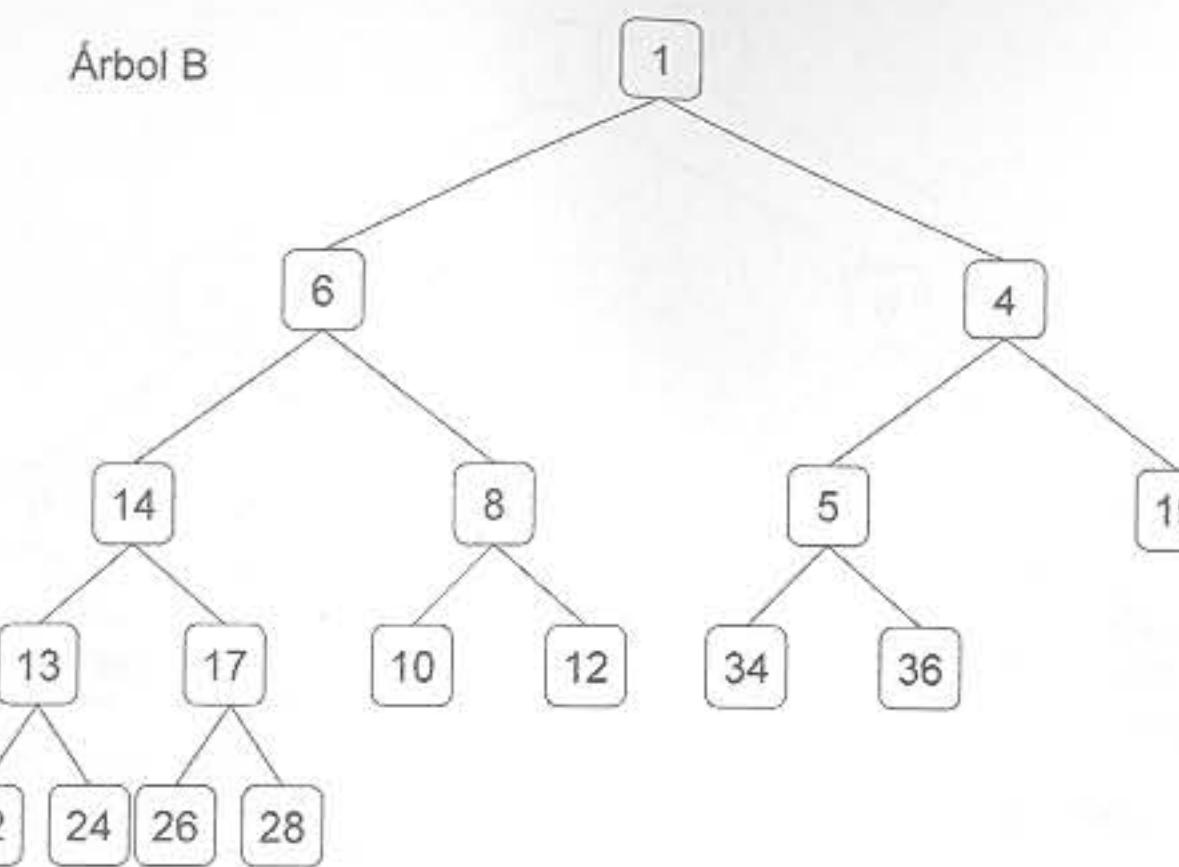
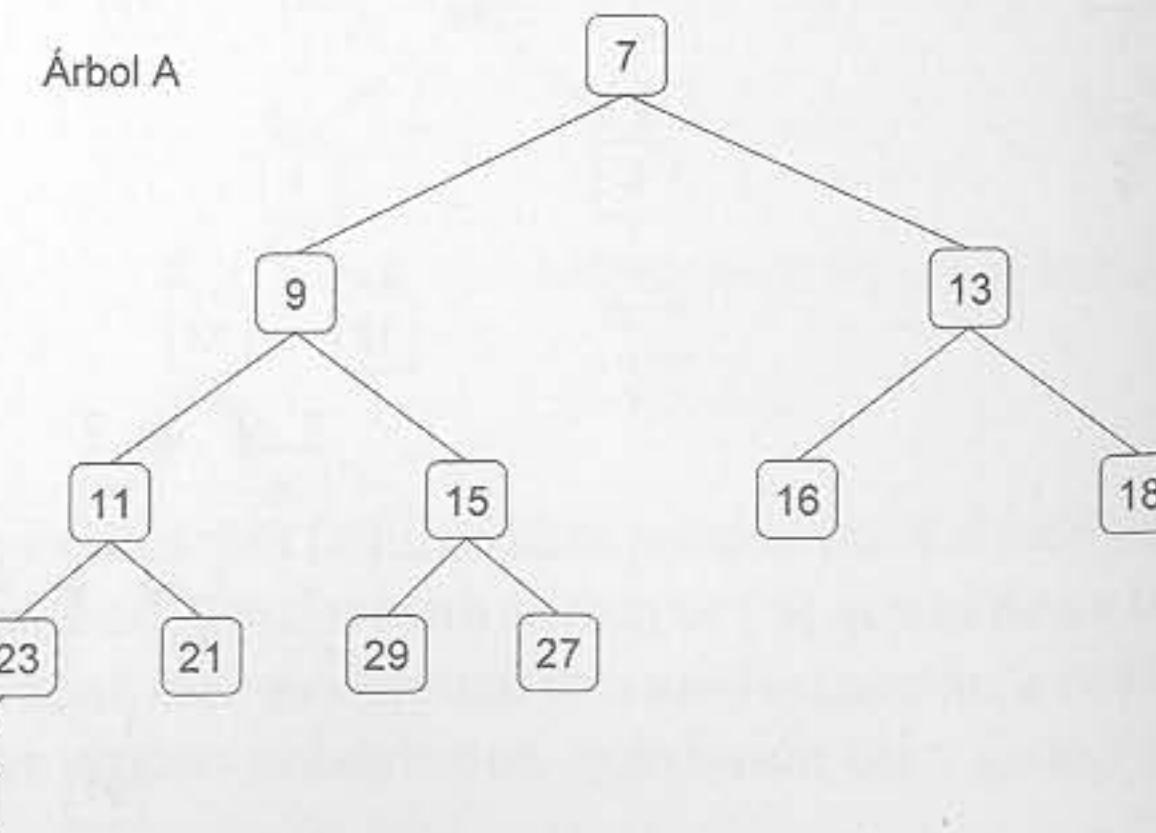


El resultado de la combinación se muestra a continuación:



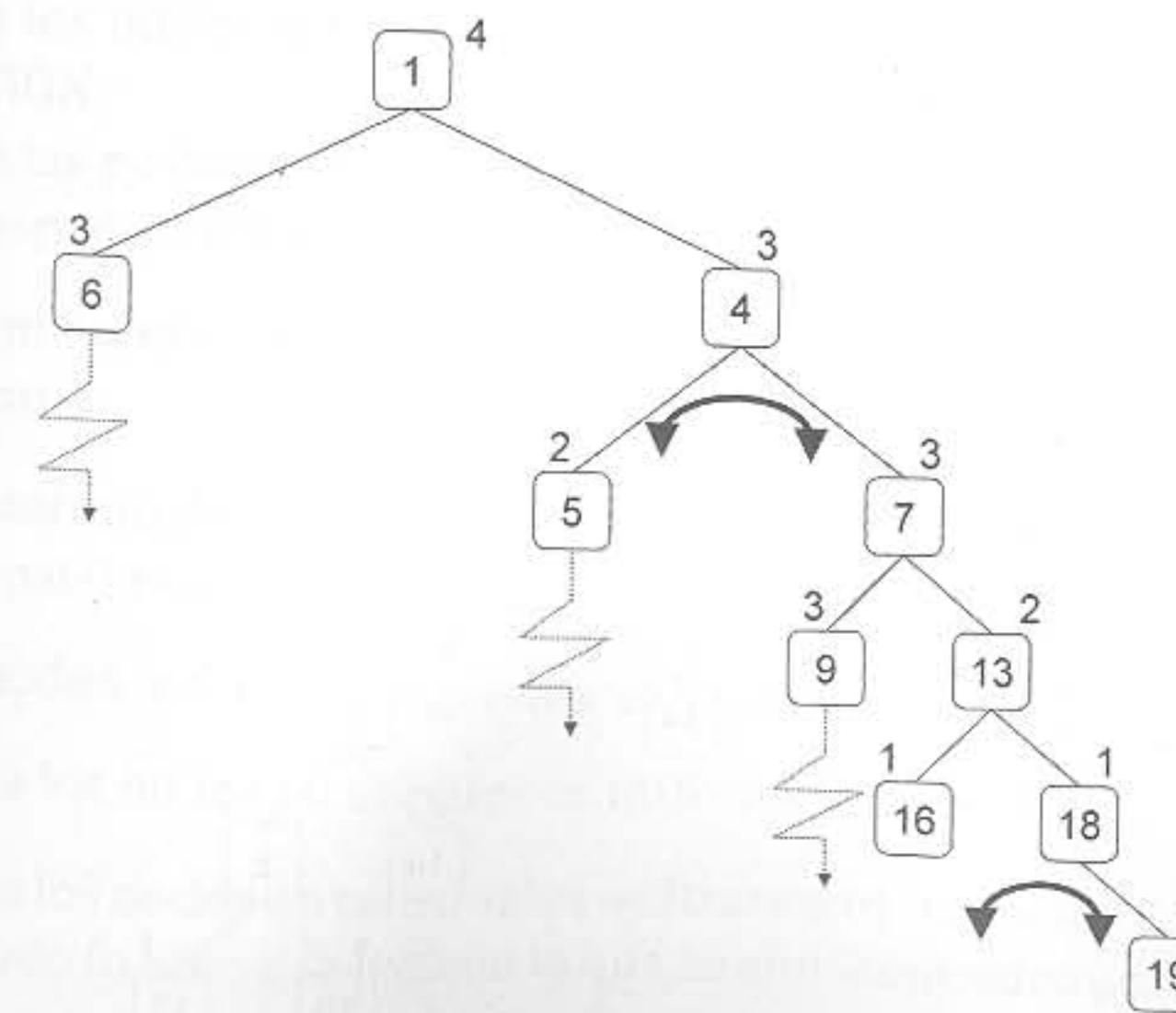
**(E.38) Dados los siguientes árboles izquierdistas mínimos:**

- A) Combinar el árbol izquierdista A con el árbol izquierdista B.
- B) Del árbol izquierdista mínimo resultante, borrar el elemento mínimo.

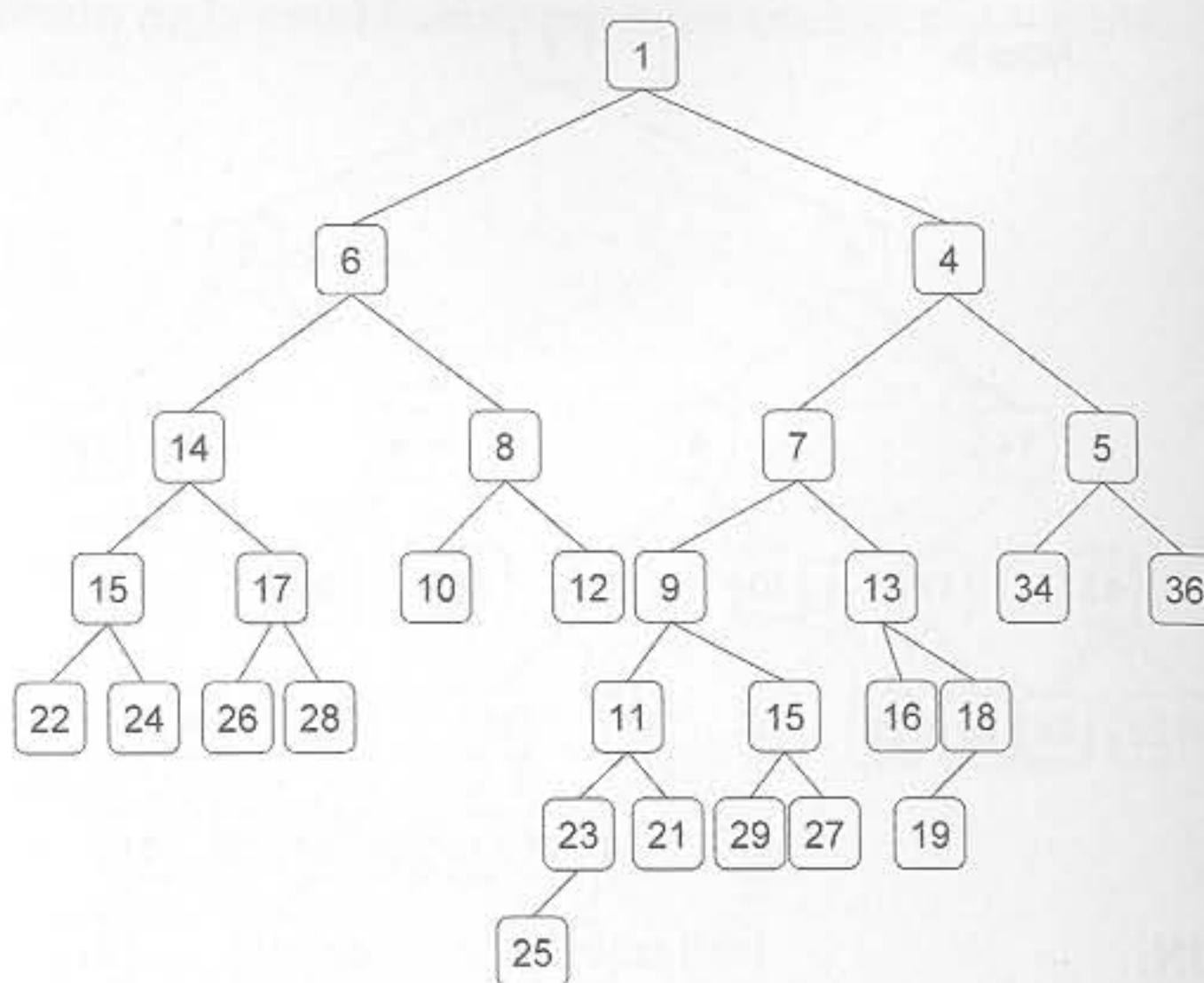


SOLUCIÓN:

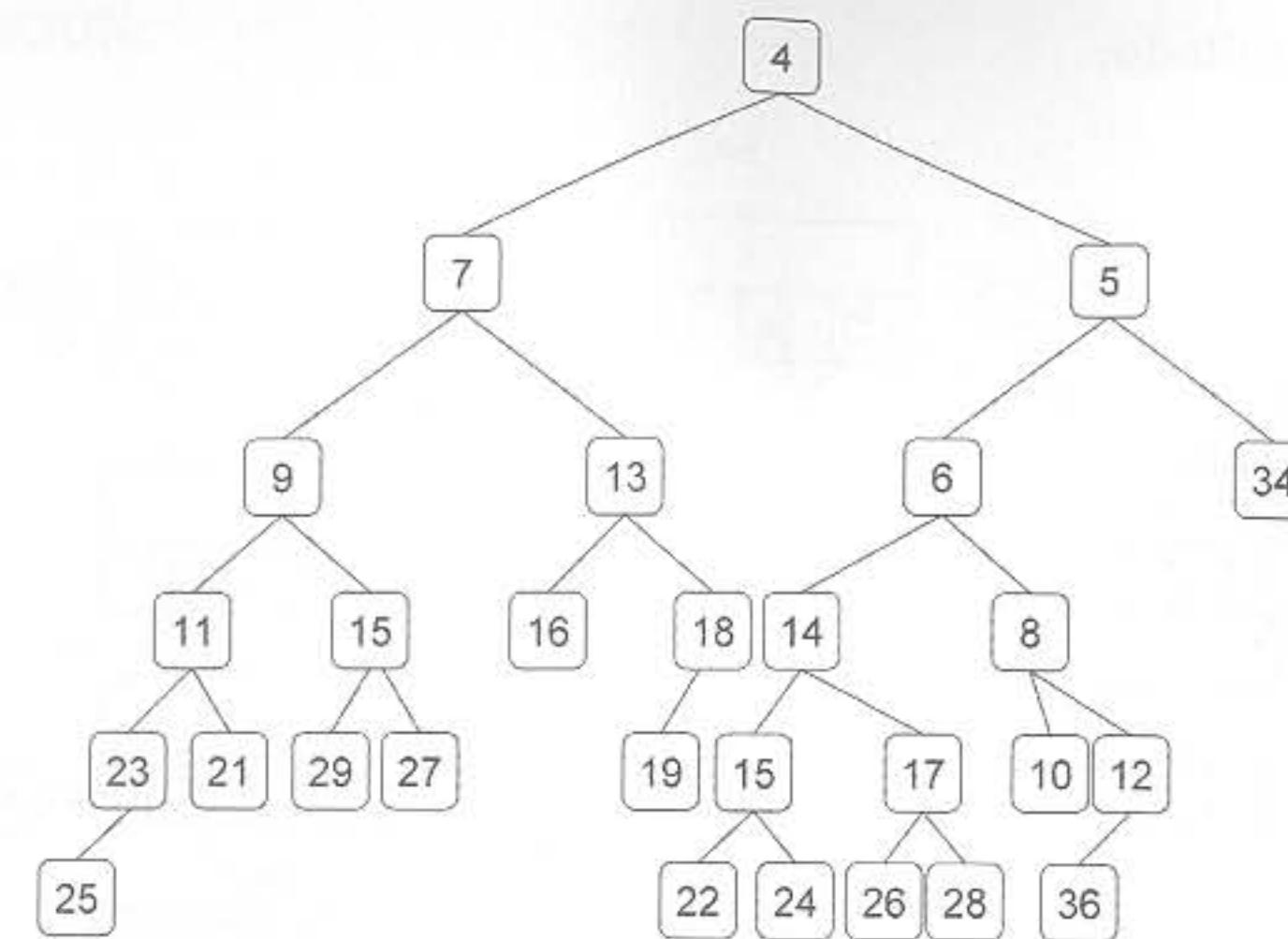
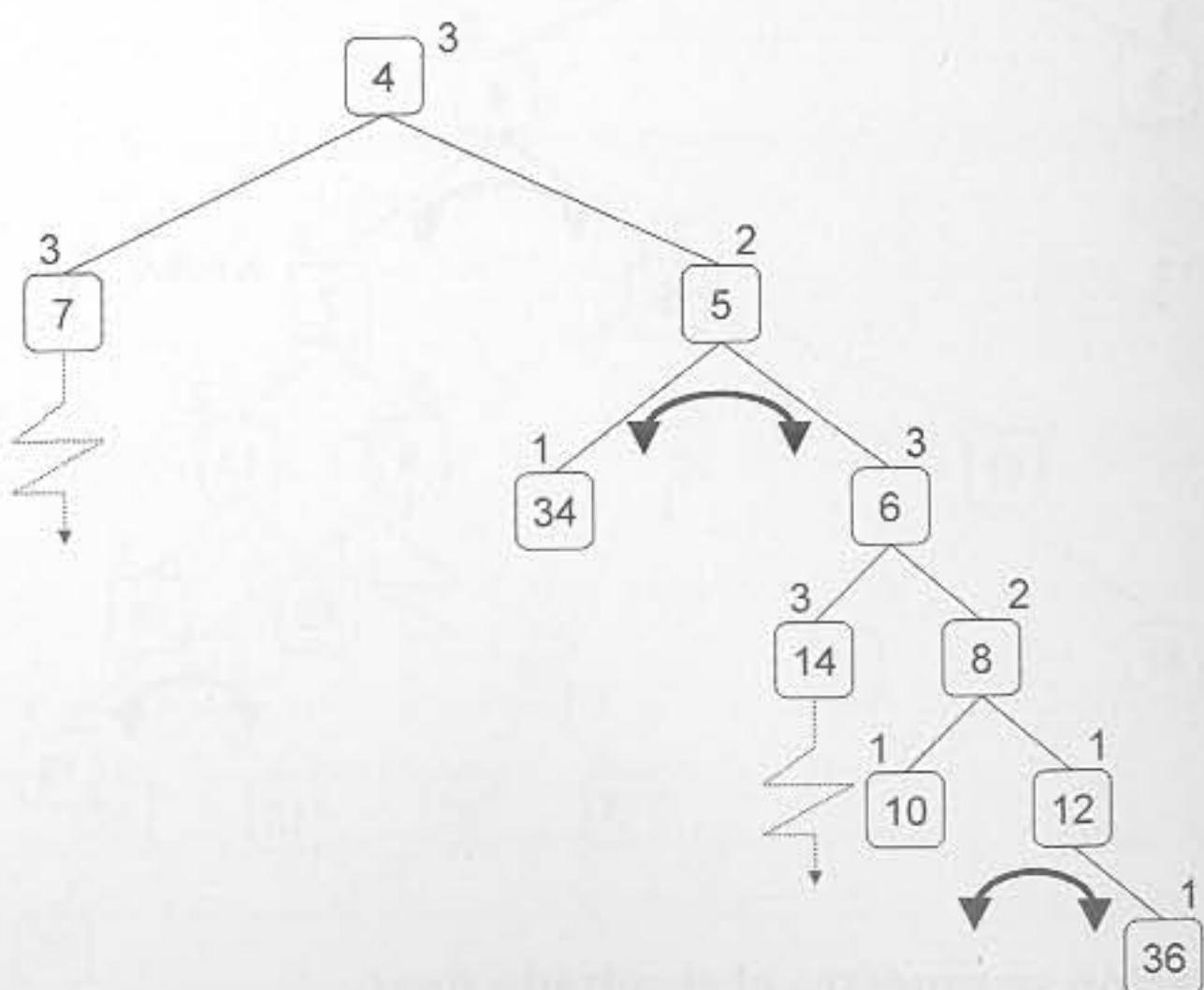
- A) Se muestra el resultado intermedio de la combinación, en el que aparecen los caminos mínimos encima de los nodos, y están indicados los intercambios a realizar entre subárboles.



A continuación se muestra el resultado final:



B) Se muestran los resultados igual que en el apartado anterior.



**(E.39)** Construye un diccionario hexadecimal que contenga las siguientes palabras:

1234, 1233, 4567, 123AB, 234, ABCD, 2349, 23495, ABC1, ABD3.

Construirlo utilizando un TAD trie con la representación más eficiente espacialmente. Describe la estructura de los nodos terminales, la de los nodos no terminales y la función de cada uno de ellos.

SOLUCIÓN:

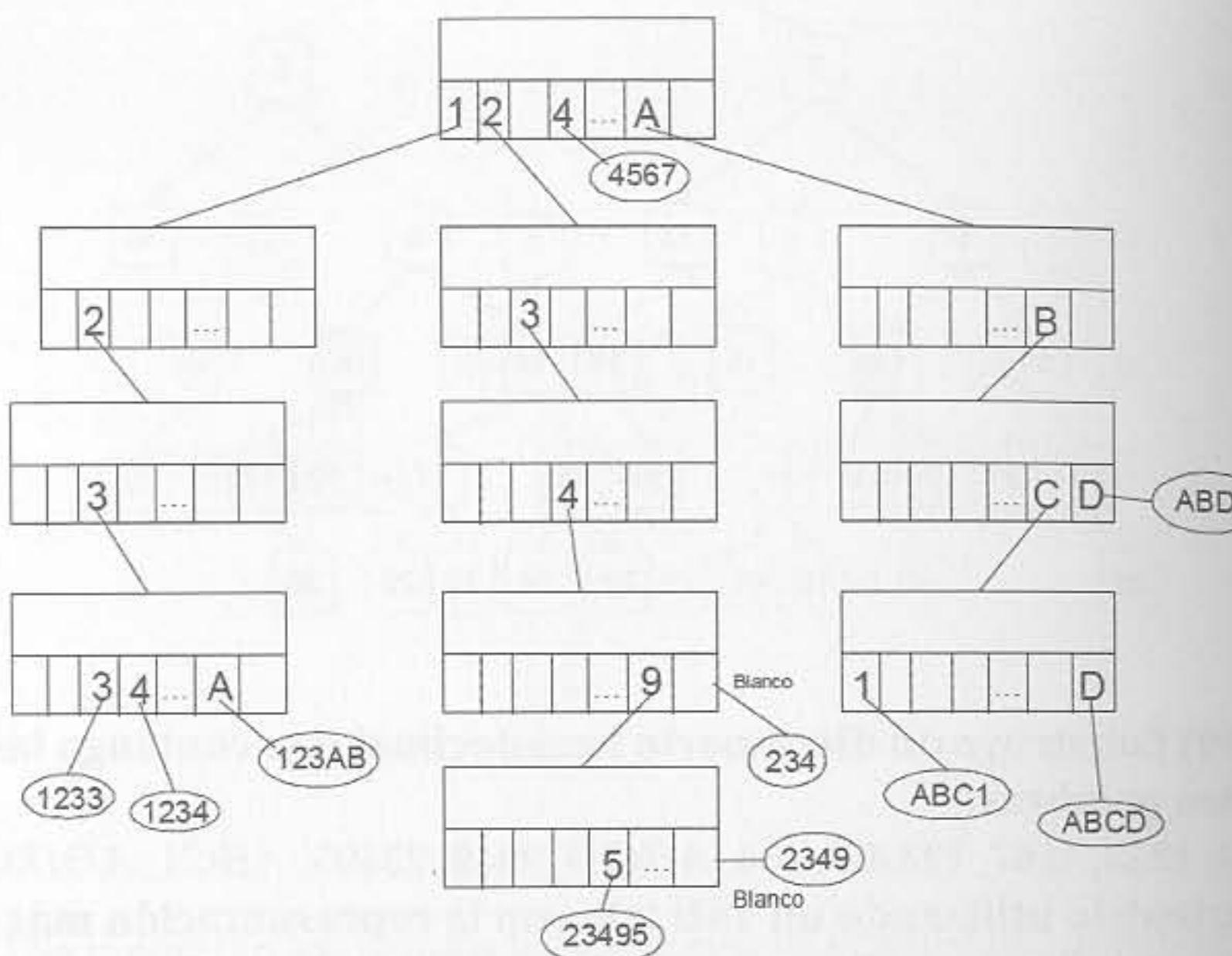
Una de las posibles representaciones que sea eficiente desde el punto de vista espacial, sería la que utiliza dos tipos de nodos:

- Terminales: donde almacenaremos la palabra completa del diccionario.
- No terminales: serán los nodos que nos representan los prefijos de las palabras.

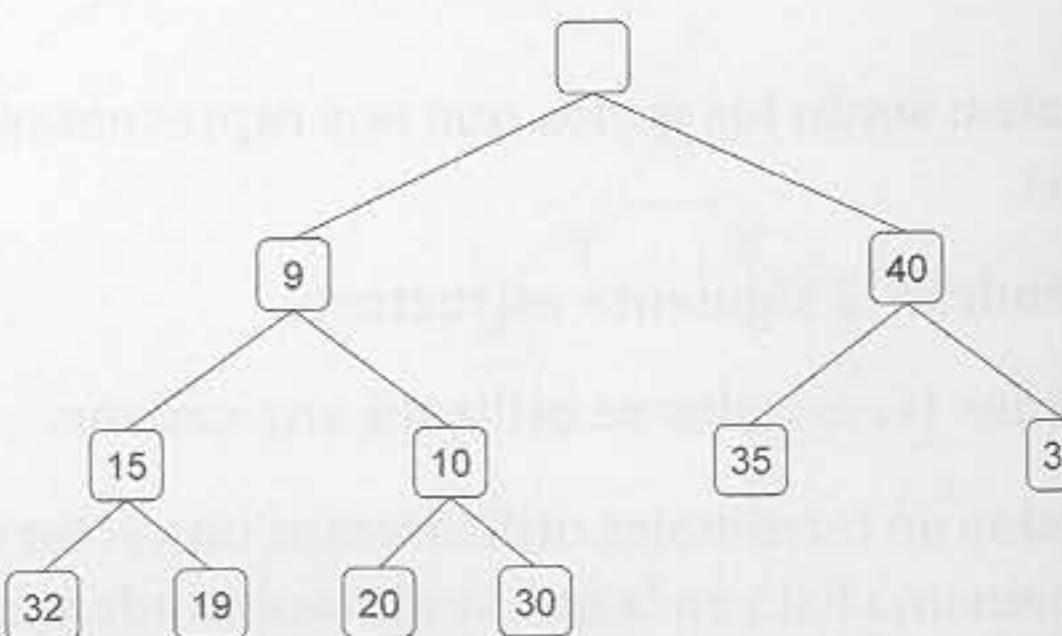
Estos nodos tendrán la siguiente estructura:

- Para los nodos terminales se utilizará una cadena.
- Para los nodos no terminales utilizaremos un vector de 17 punteros a nodo (o bien una lista en la que se almacene además del puntero el propio carácter hexadecimal), que representará a cada uno de los 16 caracteres hexadecimales y la posición 17 que representará al símbolo *blanco* o *final de cadena*.

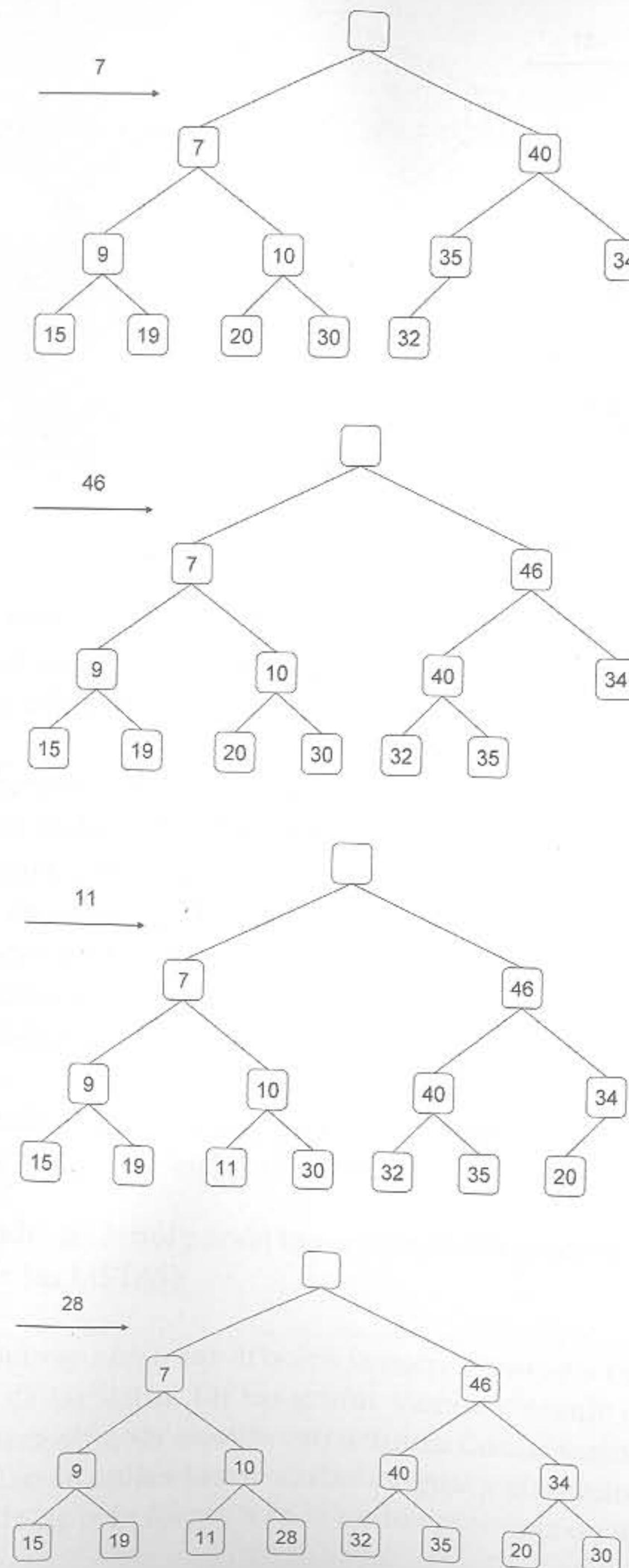
Árbol resultado:

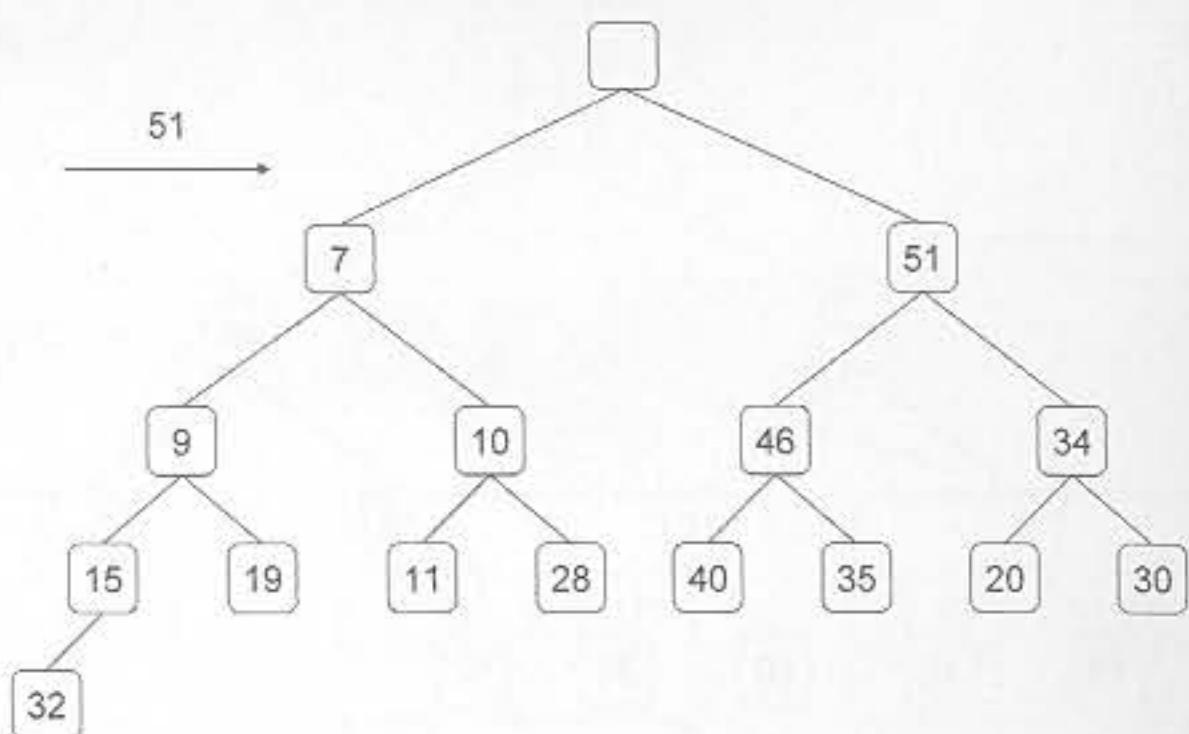


**(E.40) Insertar en el siguiente montículo doble los siguientes elementos: 7, 46, 11, 28, 51**



SOLUCIÓN:





## 7

## Tipo grafo

En los problemas originados en ciencias de la computación, matemáticas, ingeniería y muchas otras disciplinas, a menudo es necesario representar relaciones arbitrarias entre objetos de datos. Esto se consigue con los grafos.

En este apartado vamos a introducir el concepto de grafo, pero lo haremos partiendo de las estructuras de datos estudiadas anteriormente, es decir, los árboles y las listas. El elemento común entre todos ellos es la presencia de un conjunto de nodos o vértices, los cuales presentan unas relaciones entre sí como son la de predecesor y sucesor de un nodo. Veamos la evolución de estas relaciones estructurales entre nodos analizando las de los árboles:

1. Todo nodo del árbol tiene un único predecesor excepto la raíz que no tiene (igual que en las LISTAS).
2. Cada nodo del árbol puede tener más de un sucesor (relajación respecto de las LISTAS).

Es fácil observar cómo los árboles surgen como una relajación de las restricciones de las listas. En los grafos vamos a seguir relajando estas condiciones para obtener nuevas estructuras. Concretamente, dejaremos la segunda relación aplicada a los árboles igual y eliminaremos la primera, permitiendo de esta forma a cada nodo tener más de un predecesor.

## 7.1. Preguntas

**(P.128)** Al representar un grafo de  $N$  vértices y  $K$  aristas con una lista de adyacencia, la operación que halla la adyacencia de salida de un vértice, tiene una complejidad temporal en su peor caso de  $O(N)$ .

Verdadero. La operación se realiza al acceder a la lista del vértice (un solo paso), y a continuación se recorrerá cada nodo de la lista, que tendrá una longitud máxima de  $N$ , ya que no se pueden repetir vértices.

**(P.129)** Al representar un grafo dirigido de  $N$  vértices y  $K$  aristas con una matriz de adyacencia, la matriz será simétrica respecto la diagonal principal.

Falso. Esta afirmación se cumpliría con los grafos no dirigidos.

**(P.130)** Los arcos de retroceso de un recorrido en profundidad de un grafo dirigido nos indican la presencia de un ciclo.

Verdadero. Los arcos de retroceso siempre nos indicarán la presencia de un ciclo en el grafo.

**(P.131)** Al representar un grafo de  $N$  vértices y  $K$  aristas con una lista de adyacencia, la operación de hallar la adyacencia de entrada de un vértice, tiene una complejidad de  $O(K)$ .

Verdadero. Hay que recorrer cada arista del grafo para comprobar si tiene como destino el vértice, por ello sería la complejidad de  $O(K)$  o de otro modo  $O(N^2)$ .

**(P.132)** Un árbol extendido de un grafo dirigido siempre es un árbol binario.

Falso. Cada nodo del árbol extendido puede tener más de dos hijos.

**(P.133)** La siguiente secuencia de nodos de un grafo dirigido es un ciclo: 1, 2, 3, 2, 1.

Falso. Un ciclo es un camino simple, por lo que no se pueden repetir nodos, tal y como ocurre con el 2.

**(P.134)** Un bosque extendido en profundidad de un grafo dirigido también es un grafo acíclico dirigido.

Verdadero. Sólo al añadir los arcos de retroceso dejaría de ser acíclico.

**(P.135)** En un grafo dirigido pueden existir infinitas aristas para un número  $n$  de vértices.

Falso. Esta condición se cumpliría sólo para los multigrafos.

**(P.136)** Un bosque extendido en profundidad de un grafo dirigido al que se le añaden los arcos de avance y de cruce es un grafo acíclico dirigido.

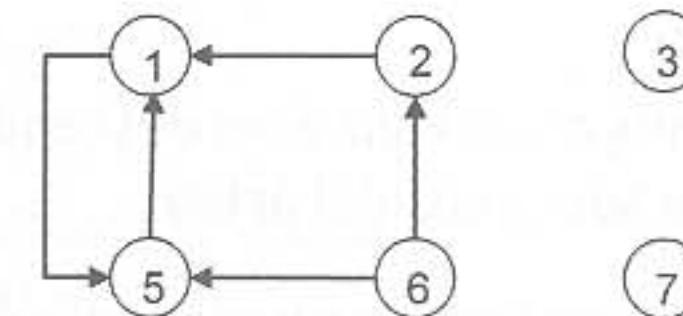
Verdadero. Sólo al añadir los arcos de retroceso dejaría de ser acíclico.

**(P.137)** Los árboles extendidos de un grafo dirigido tienen que ser necesariamente un árbol binario.

Falso. Cada nodo del árbol extendido puede tener más de dos hijos.

**(P.138)** Sea  $G$  un grafo dirigido de  $n$  vértices. Si  $G$  tiene  $n-1$  aristas, entonces nunca podría tener un ciclo.

Falso. El siguiente contraejemplo contradice esta afirmación:



## 7.2. Ejercicios

**(E.41)** Modificar el siguiente algoritmo que realiza el recorrido en profundidad de un grafo no dirigido para que además de realizar este recorrido, escriba el tipo de cada arista del grafo según la clasificación de aristas del bosque extendido en profundidad.

```

ALGORITMO DFS(v: TVértice, G: TGrafo, visitados:
  TConjuntoDeTVértice)
  VAR w: TVértice;
  MÉTODO
    Visitar(v);
    Insertar(visitados, v);
    para todo w ∈ AdyacenciasSalida(v, G) hacer
      si no Pertenece(visitados, w) entonces
        DFS(w, G, visitados);
    fsi
  
```

```
fpara
FMÉTODO
```

**SOLUCIÓN:**

Una de las formas de recorrer un grafo es la conocida como DFS (*Depth First Search*) o recorrido en profundidad. Este recorrido consiste en, partiendo de un nodo  $v$ , recorrer cada camino que parte de  $v$  hasta que finalicen (*profundizar* en ellos) repitiendo a continuación el proceso con cada nuevo nodo no visitado.

En un grafo dirigido, cuando se calcula el recorrido en profundidad nos podemos encontrar con cuatro tipos de arcos:

- Arcos del árbol: nos conducen a vértices que no han sido visitados anteriormente. Forman lo que se conoce como un bosque extendido en profundidad para el grafo.
- Arcos de retroceso: nos conducen a uno de sus antecesores en el bosque extendido.
- Arcos de avance: nos conducen a un descendiente propio del bosque extendido y no son arcos del árbol.
- Arcos de cruce: nos conducen a un vértice que no es ni antecesor ni sucesor en el bosque, o sea, que une dos ramas del árbol.

Sin embargo, en un grafo NO dirigido solo tenemos aristas del árbol, que nos conducen a vértices no visitados anteriormente, y aristas de retroceso (los arcos de avance y de cruce se funden en los de retroceso), que nos conducen a vértices visitados anteriormente. Por ello, el único cambio a realizar en el algoritmo original sería el que aparece resaltado a continuación, de modo que cuando se alcance un vértice que esté en el conjunto *visitados* (lo que significa que ya ha sido visitado anteriormente) entonces se clasificará como *arista de retroceso*.

```
ALGORITMO DFS(v: TVértice, G: TGrafo, visitados:
TConjuntoDeTVértice)
VAR w: TVértice;
MÉTODO
    Visitar(v);
    Insertar(visitados, v);
    para todo w ∈ AdyacenciaSalida(v, G) hacer
        si Pertenece(visitados, w) entonces
```

```
        escribir(v, w);
        escribir('ARISTA DE RETROCESO');
    sino
        escribir(v, w);
        escribir('ARISTA DEL ÁRBOL');
        DFS(w, G, visitados);
    fsi
fpara
FMÉTODO
```

**(E.42) Modificar el siguiente algoritmo que realiza el recorrido en anchura de un grafo no dirigido para que además de realizar este recorrido, escriba el tipo de cada arista del grafo según la clasificación de aristas del bosque extendido en anchura.**

```
ALGORITMO BFS(v: TVértice, G: TGrafo, visitados:
TConjuntoDeTVértice)
VAR w1, w2: TVértice; Q: TCola(TVértice);
MÉTODO
    visitados.Insertar(v);
    Q.Encolar(v);
    Visitar(v);
    mientras no Q.EsVacia() hacer
        w1 = Q.Cabeza();
        Q.Desencolar();
        para todo w2 ∈ AdyacenciaDeSalida(w1) hacer
            si no visitados.Pertenece(w2) entonces
                Visitar(w2);
                Q.Encolar(w2);
                visitados.Insertar(w2);
            fsi
        fpara
    fmientras
FMÉTODO
```

**SOLUCIÓN:**

Una de las formas de recorrer un grafo es la conocida como BFS (*Breadth First Search*) o recorrido en anchura. Este recorrido consiste en, partiendo de un nodo  $v$ , primero visitar todos los vértices adyacentes a  $v$  y después, sucesivamente, los adyacentes a éstos últimos.

En un grafo NO dirigido, cuando se calcula el recorrido en anchura nos podemos encontrar con dos tipos de arcos (aristas):

- Arcos del árbol: nos conducen a vértices que no han sido visitados anteriormente. Forman lo que se conoce como un bosque extendido en anchura para el grafo.
- Arcos de retroceso: nos conducen a uno de sus antecesores en el bosque extendido.

Esto significa que los arcos de avance y de cruce se funden en los de retroceso, que nos conducen a vértices visitados anteriormente. Por ello, el único cambio a realizar en el algoritmo original sería el que aparece resaltado a continuación, de modo que cuando se alcance un vértice que esté en el conjunto *visitados* (lo que significa que ya ha sido visitado anteriormente) entonces se clasificará como *arista de retroceso*.

```

ALGORITMO BFS(v: TVértice, G: TGrafo, visitados:
    TConjuntoDeTVértice)
VAR w1, w2: TVértice; Q: TCola(TVértice);
MÉTODO
    visitados.Insertar(v);
    Q.Encolar(v);
    Visitar(v);
    mientras no Q.EsVacia() hacer
        w1 = Q.Cabeza();
        Q.Desencolar();
        para todo w2 ∈ AdyacenciaDeSalida(w1) hacer
            si no visitados.Pertenece(w2) entonces
                escribir(w1, w2);
                escribir('ARISTA DEL ÁRBOL');
                Visitar(w2);
                Q.Encolar(w2);
                visitados.Insertar(w2);
            sino
                escribir(w1, w2);
                escribir('ARISTA DE RETROCESO');
            fsi
        fpara
    fmientras
FMÉTODO

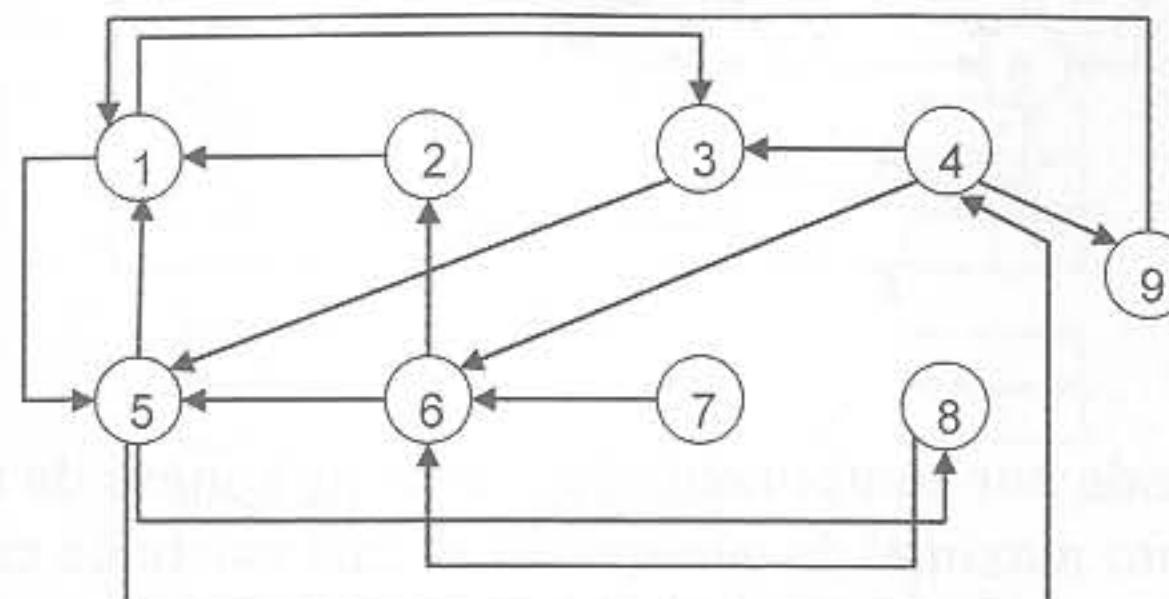
```

(E.43) Dado el siguiente grafo dirigido, obtén:

A) El bosque extendido en profundidad comenzando por el vértice 1 expresado en forma de árbol (ante la posibilidad de elegir continuar por un nodo u otro, elegiremos siempre el que menor etiqueta tenga).

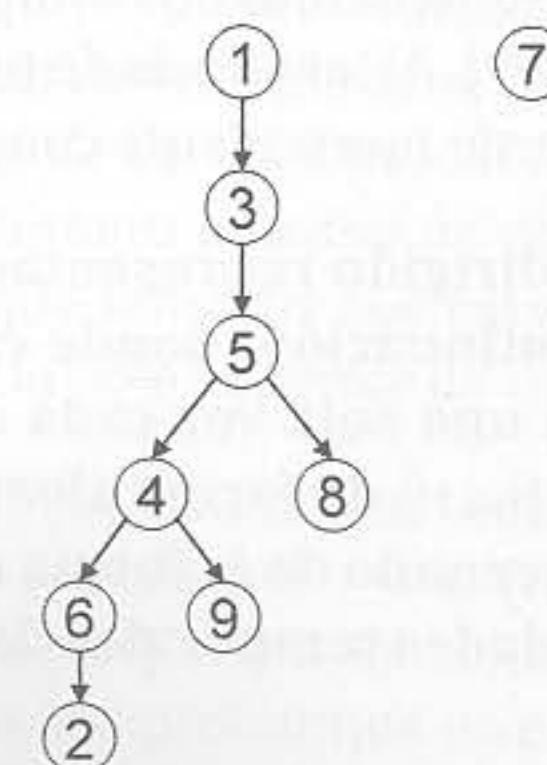
B) Identifica los distintos tipos de arcos del grafo (arco de árbol (A), de avance (Av), de retroceso (R), de cruce (Cr)).

C) Extrae las componentes fuertemente conexas del grafo anterior. ¿Es un grafo fuertemente conexo?

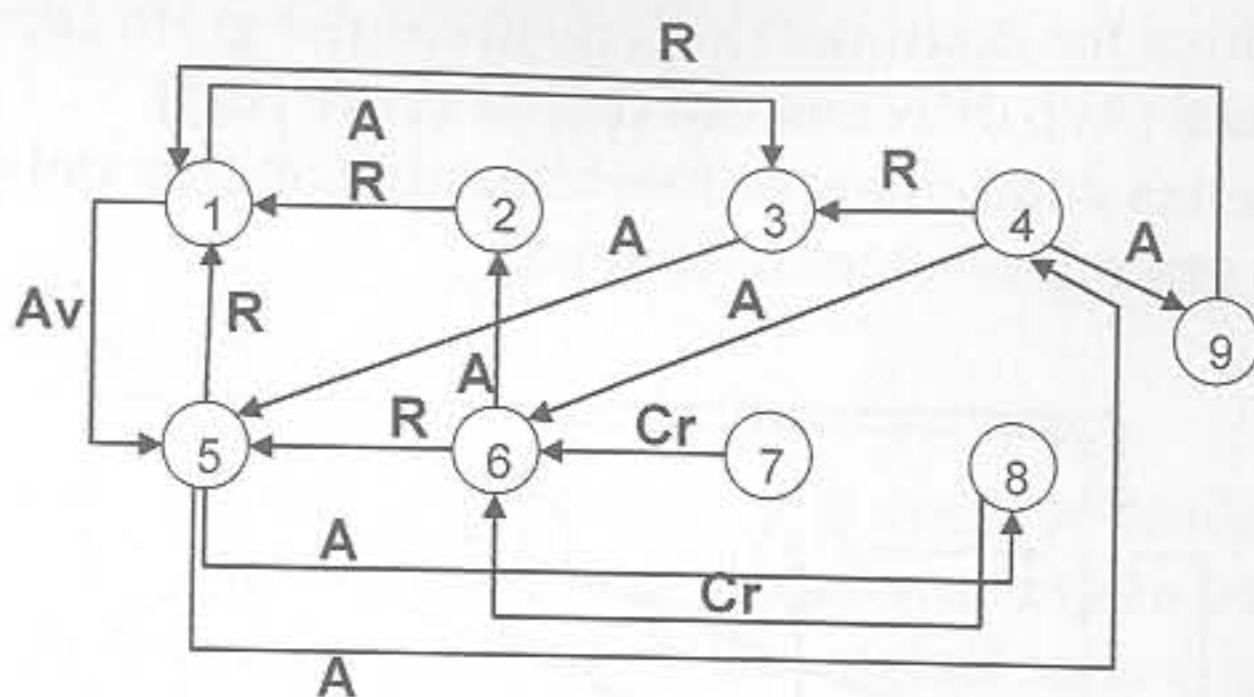


SOLUCIÓN:

A) Siguiendo el criterio de que ante la posibilidad de elegir continuar por un nodo u otro, elijamos siempre el de menor etiqueta, es decir, considerando la adyacencia de salida ordenada de menor a mayor, tendríamos los dos siguientes árboles en el bosque extendido en profundidad. El recorrido en profundidad (el orden en que visitaríamos los vértices) sería el siguiente:  $DFS(1) = 1, 3, 5, 4, 6, 2, 9, 8$



B) Basándonos en el recorrido en profundidad del apartado anterior, y siguiendo el mismo criterio de recorrido de la adyacencia de salida, obtendríamos la siguiente clasificación:



C) Se entiende por componente fuertemente conexa de un grafo dirigido al conjunto maximal de vértices en el cual existe un camino que va desde cualquier vértice del conjunto hasta cualquier otro vértice también del conjunto.

Para determinar las componentes fuertemente conexas de un digrafo  $G = (V, A)$  se pueden seguir los siguientes pasos:

- 1)  $V$  se divide en clases de equivalencia  $V_i$ ,  $1 \leq i \leq r$ , tales que  $v$  y  $w$  son equivalentes SI Y SOLO SI existe un camino de  $v$  a  $w$  y otro de  $w$  a  $v$ .
- 2) Sea  $A_i$ ,  $1 \leq i \leq r$ , conjunto de arcos con cabeza y cola en  $V_i$ .
- 3) Los grafos  $G_i = (V_i, A_i)$  son componentes fuertemente conexas de  $G$ .

Aplicando este algoritmo obtenemos dos componentes fuertemente conexas:  $\{7\}$  y  $\{1, 2, 3, 4, 5, 6, 8, 9\}$ . Al tener más de una componente entonces se concluye que no es un grafo fuertemente conexo.

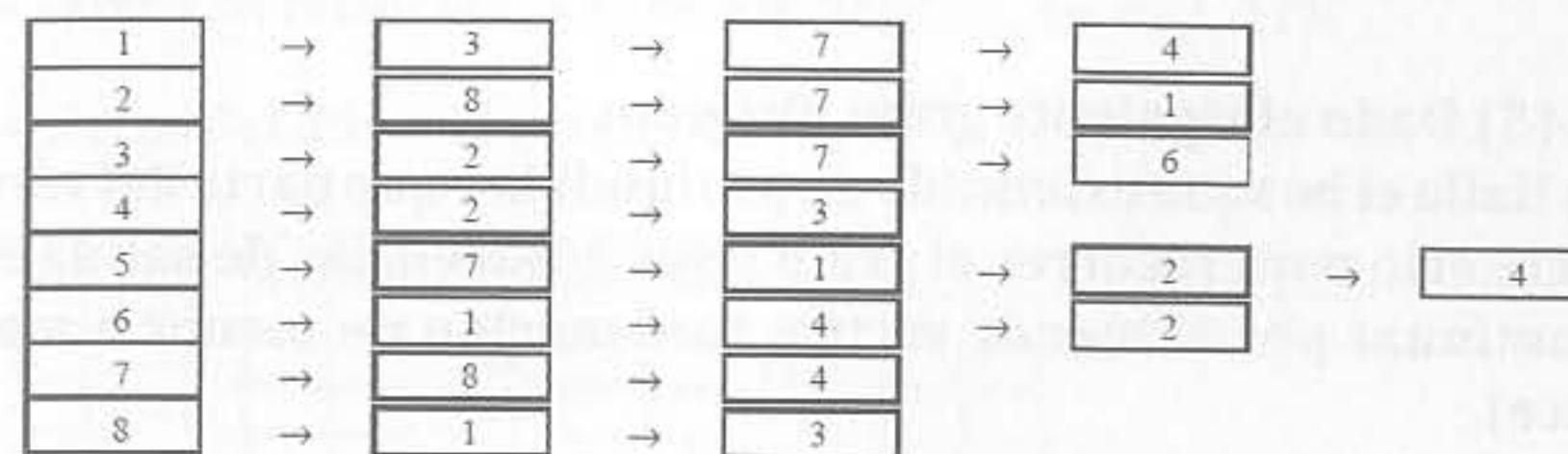
**(E.44)** Sea el grafo no dirigido representado por la lista de adyacencia que aparece a continuación, donde cada lista estará desordenada y se almacenará una sola vez cada arco  $(v, w)$ , es decir, se almacenará sólo  $(v, w)$  o  $(w, v)$  de forma aleatoria.

**A)** Dado un grafo almacenado de la forma descrita, calcula la cota superior de las complejidades temporales de las operaciones:

- $Pertenec(v, w)$
- $Adyacencia(v)$
- $Insertar(v, w)$

en función del número  $n$  de vértices del grafo.

**B)** Realiza el recorrido en profundidad de dicho grafo y la clasificación de arcos, partiendo del vértice 1. Para cada vértice se continuará el recorrido escogiendo el menor vértice no visitado de su adyacencia.



SOLUCIÓN:

A) Siendo  $n$  el número de vértices del grafo:

- $Pertenec(v, w) : O(n)$ . El acceso al vector se realiza en tiempo constante,  $O(1)$ , ya se acceda por  $v$  o  $w$  (buscando la arista  $(v, w)$  o  $(w, v)$  respectivamente). A continuación habría que realizar la búsqueda lineal dentro de la lista de vértices, cuyo tamaño máximo será  $n$ , de ahí la complejidad lineal en función del número de vértices.
- $Adyacencia(v) : O(n^2)$ . Para realizar esta operación tendríamos que recorrer la lista de vértices de  $n$ , y a continuación revisar todas las restantes listas de vértices buscando la aparición del vértice  $v$ . Sabiendo que el número máximo de vértices en un grafo no dirigido (el caso peor que tenemos que estudiar) es de  $n(n - 1)/2$ , nos quedaríamos con la cota superior de complejidad de  $n^2$ .
- $Insertar(v, w) : O(n)$ . El acceso al vector se realiza en tiempo constante,  $O(1)$ , ya se acceda por  $v$  o  $w$  (insertando la arista  $(v, w)$  o  $(w, v)$  respectivamente). A continuación habría que buscar en la lista de vértices para comprobar que no existiese previamente ya que no se permiten múltiples ocurrencias de la misma arista, operación que tendrá un coste lineal en función del número de vértices (el tamaño máximo de la lista).

B) La solución es:

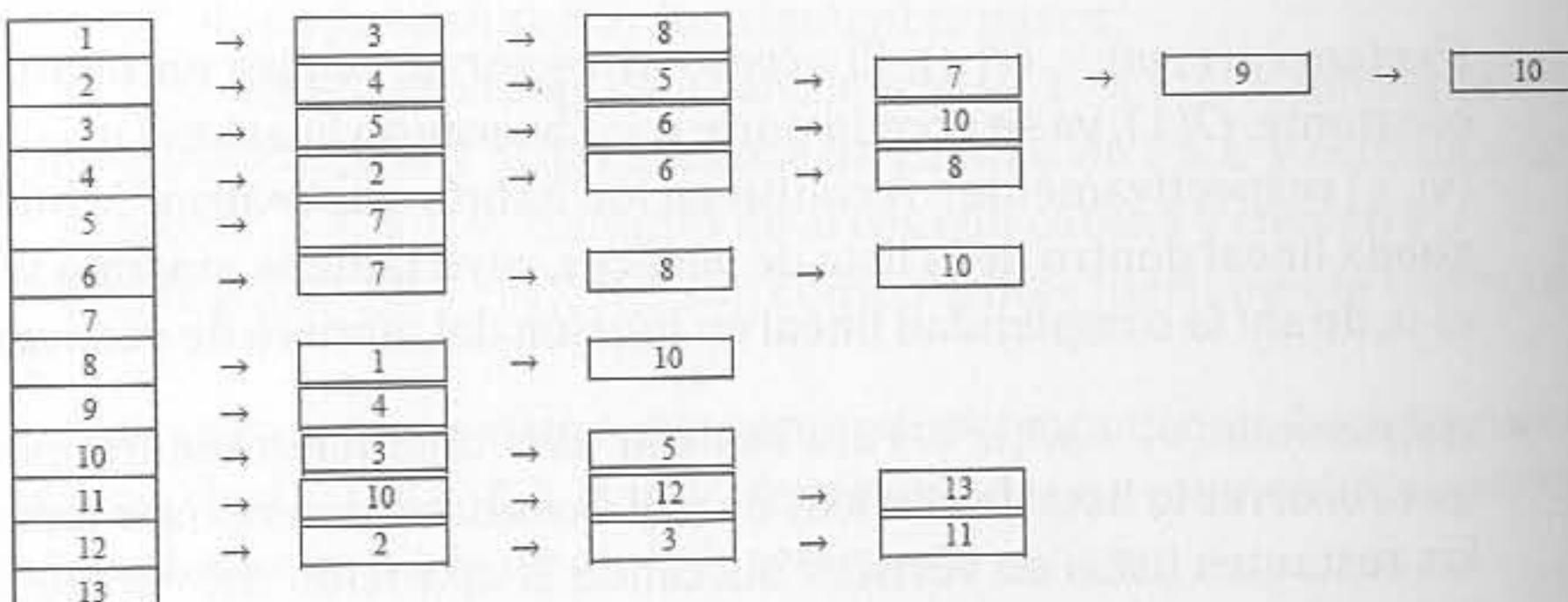
- Recorrido en profundidad: 1, 2, 3, 4, 5, 7, 8, 6.
- Arcos del árbol: (1, 2) (2, 3) (3, 4) (4, 5) (5, 7) (7, 8) (4, 6).
- Arcos de retroceso: los restantes.

(E.45) Dado el siguiente grafo dirigido:

A) Halla el bosque extendido en profundidad que parte del vértice 1. El criterio para recorrer el grafo y sus adyacencias de salida es el de continuar por el menor vértice (ordenación de menor a mayor vértice).

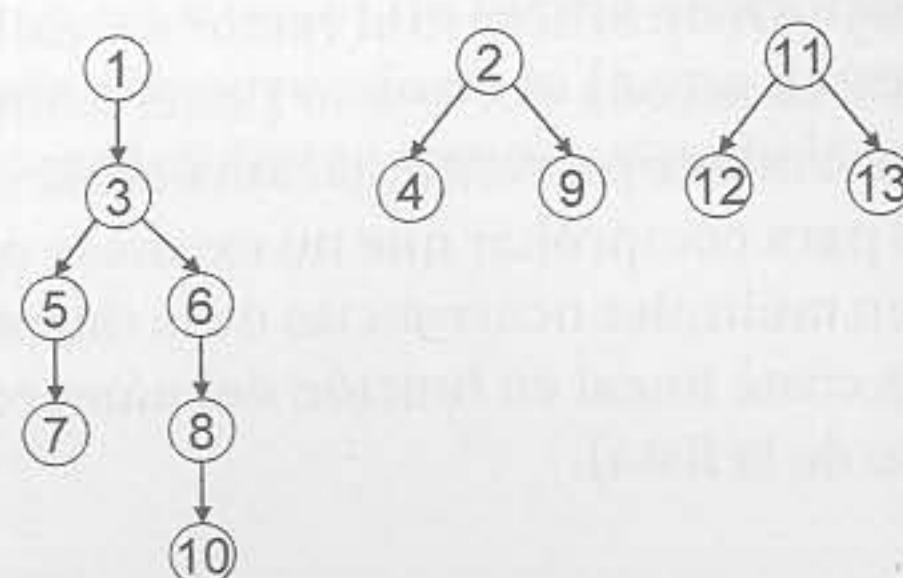
B) Clasifica los arcos de dicho grafo.

C) Escribe (según la definición de ciclo, no valdrá dibujarlos) los ciclos que presenta dicho grafo.



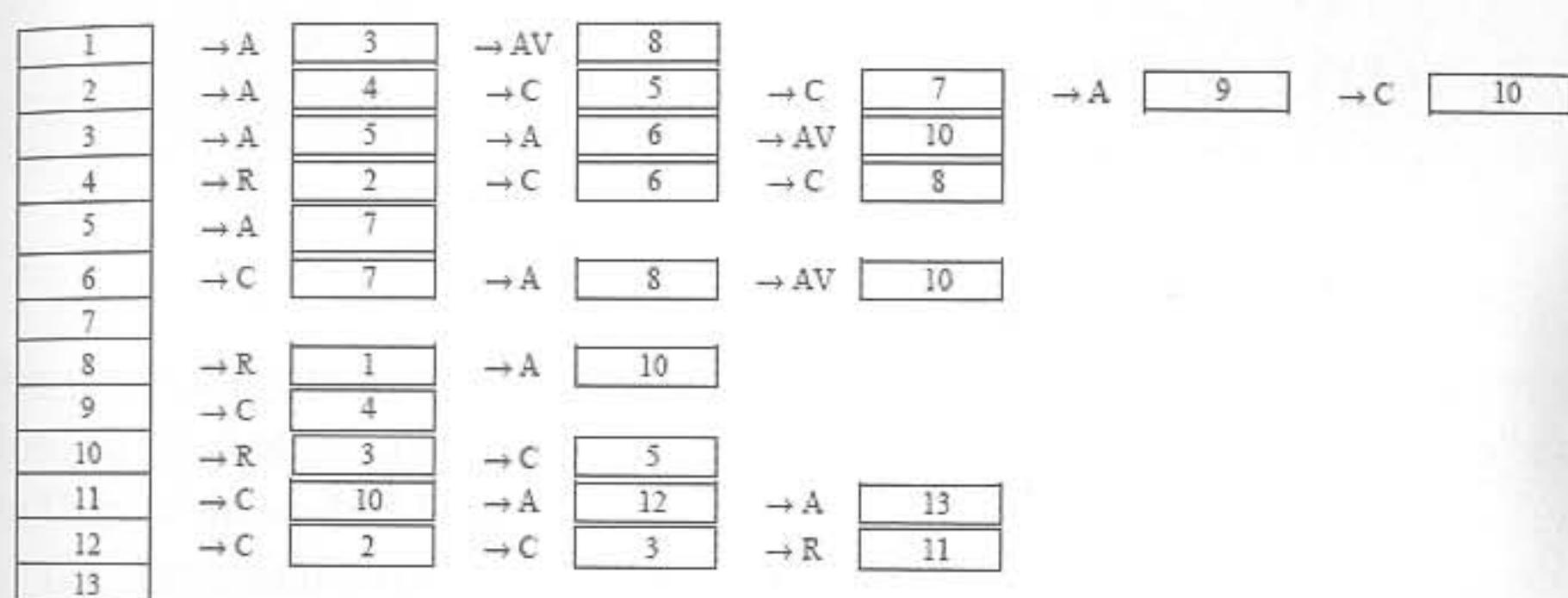
SOLUCIÓN:

A)



B) Clasifica los arcos de dicho grafo:

- Arcos de avance: <1, 8>, <3, 10>, <6, 10>
- Arcos de cruce: <2, 5>, <2, 7>, <2, 10>, <4, 6>, <4, 8>, <6, 7>, <9, 4>, <10, 5>, <11, 10>, <12, 2>, <12, 3>
- Arcos de retroceso: <4, 2>, <8, 1>, <10, 3>, <12, 11>
- Arcos del árbol: los restantes



C) Un ciclo es un camino simple en el que sólo se repiten el vértice primero y último. Por ello se presentan los siguientes caminos (secuencia de vértices):

- 1, 3, 6, 8, 1
- 3, 6, 8, 10, 3
- 2, 4, 2
- 11, 12, 11
- 1, 8, 1
- 3, 6, 10, 3
- 2, 9, 4, 2
- 10, 3, 10

## Exámenes

En este capítulo se presentan cuatro exámenes sin solución similares a los realizados en la asignatura “Programación y Estructuras de Datos” de la Universidad de Alicante.

Cada examen se compone de dos partes: el test y los ejercicios de desarrollo. El test se compone de 14 a 16 preguntas, cuya respuesta es verdadero o falso.

Cada pregunta posee dos numeraciones:

- La primera indica el número de orden de la pregunta dentro del examen. Para cada examen, esta numeración empieza desde 1.
- La segunda numeración es única para todos los exámenes e indica la posición de la pregunta en el conjunto de todas las preguntas que figuran en este libro. Esta numeración permite localizar rápidamente la solución y explicación de la pregunta en la primera parte del libro.

Respecto a la parte de los ejercicios de desarrollo, en cada examen suelen haber 3 o 4 ejercicios. Los ejercicios también aparecen con dos numeraciones con el mismo significado que la numeración de las preguntas.

## 8.1. Examen 1

### 8.1.1. Test

1. (Pregunta 128, página 138)

Al representar un grafo de  $N$  vértices y  $K$  aristas con una lista de adyacencia, la operación que halla la adyacencia de salida de un vértice, tiene una complejidad de  $O(N)$ .

2. (Pregunta 82, página 107)

El nodo de un árbol B m-camino de búsqueda con  $m = 4$  puede tener como máximo 4 hijos no vacíos.

3. (Pregunta 62, página 87)

El número mínimo de elementos que se puede almacenar en un árbol 2-3 de altura  $h$  coincide con el número de elementos que hay en un árbol binario completo de altura  $h$ .

4. (Pregunta 115, página 119)

En el proceso de inserción en un montículo máximo insertaremos en la siguiente posición libre para que siga siendo un árbol binario lleno.

5. (Pregunta 71, página 91)

En un árbol 2-3-4 con  $n$  elementos, la altura de dicho árbol se encuentra entre  $\log_4(n + 1)$  y  $\log_2(n + 1)$ .

6. (Pregunta 35, página 68)

En un árbol binario, el camino mínimo de la raíz es igual a la altura del árbol.

7. (Pregunta 75, página 102)

En un árbol rojo-negro ningún camino desde la raíz a las hojas tiene dos o más hijos negros consecutivos.

8. (Pregunta 49, página 75)

La complejidad de las operaciones de equilibrado de los árboles AVL sugiere que deben utilizarse sólo si las inserciones son considerablemente más frecuentes que las búsquedas.

## 9. (Pregunta 102, página 115)

La complejidad temporal de las operaciones de inserción y búsqueda en un árbol de búsqueda digital están en función del número de bits de la clave.

## 10. (Pregunta 1, página 24)

Sea el siguiente TAD:

```
MÓDULO NATURALEXAMEN
TIPO natural
OPERACIONES
    uno: → natural;
    siguiente: natural → natural;
    sumar: natural natural → natural;
FMÓDULO
```

Si  $N$  es un natural, entonces  $N = \text{sumar}(\text{uno}, \text{siguiente}(\text{uno}))$  es un uso sintácticamente incorrecto de la operación *sumar*.

## 11. (Pregunta 89, página 112)

La representación de conjuntos mediante vectores de bits tiene una complejidad espacial proporcional al tamaño del conjunto universal.

## 12. (Pregunta 15, página 46)

Sea el método *Primera* perteneciente a la clase *TLista* que devuelve la primera posición de la lista que lo invoca:

```
class TLista {
public:
    ...
private:
    TNodo *lis;
}

TPosicion TLista::Primera() {
    TPosicion p;
    p.pos = lis;
    return p;
}
```

En la instrucción `return p;` del método *Primera* se invoca al constructor de copia de *TPosicion*.

## 13. (Pregunta 107, página 117)

Sea el TAD Unión-Búsqueda implementado por medio de vectores de bits. La operación

$\text{busqueda}(\text{Elemento}) \rightarrow \text{Conjunto}$

que nos devuelve todos los elementos del conjunto al que pertenece *Elemento*, tiene una complejidad  $O(k)$ , siendo  $k$  el número de conjuntos en el TAD.

## 14. (Pregunta 99, página 114)

Sea una tabla de dispersión cerrada con función de dispersión

$$H(x) = x \bmod B$$

con  $B = 100$  y  $x$  un número natural entre 1 y 2000. Sólo hay un valor de  $x$  que haga  $H(x) = 4$ .

## 8.1.2. Ejercicios

## 1. (Ejercicio 1, página 28)

Sea la sintaxis y semántica de las siguientes operaciones del tipo árbol binario (AB) cuyos elementos son números naturales:

```
VAR i, d: AB; x: Natural;

Crea_AB () → AB

Enraizar(AB, Natural, AB) → AB

Raiz(AB) → Natural
Raiz(Crea_AB()) = Error_Natural()
Raiz(Enraizar(i, x, d)) = x

EsVacio(AB) → booleano
EsVacio(Crea_AB()) = CIERTO
EsVacio(Enraizar(i, x, d)) = FALSO

HijoIz(AB) → AB
HijoIz(Crea_AB()) = Crea_AB()
HijoIz(Enraizar(i, x, d)) = i
```

```
HijoDe(AB) → AB
HijoDe(Crea_AB()) = Crea_AB()
HijoDe(Enraizar(i, x, d)) = d
```

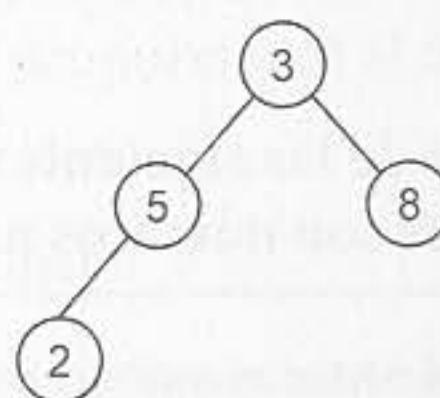
Se define la operación *Suma\_AB* que recibe un árbol binario y devuelve un *Natural* que representa la suma de las etiquetas de todos los nodos del árbol:  $\text{Suma\_AB}(AB) \rightarrow \text{Natural}$ .

A) Escribir únicamente con dos ecuaciones la semántica de esta operación.

B) Se define la siguiente operación *Examen(AB)*. Explicar qué hace esta operación detallando el comportamiento de las dos ecuaciones que aparecen a continuación:

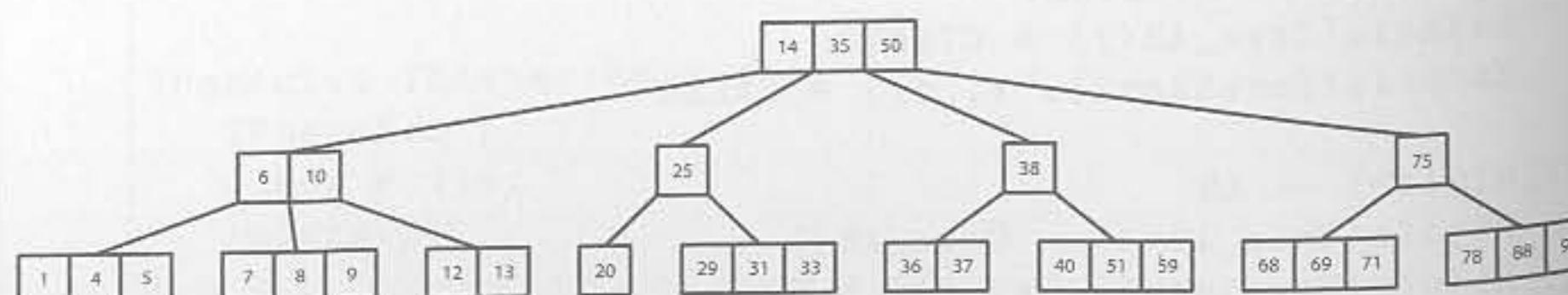
```
Examen(AB) → AB
Examen(Crea_AB()) = Crea_AB()
Examen(Enraizar(i, x, d)) = Enraizar(Examen(i), x +
    Suma_AB(i) + Suma_AB(d), Examen(d))
```

C) Dibujar el árbol binario resultante tras aplicar la operación *Examen* sobre el árbol binario:



## 2. (Ejercicio 31, página 104)

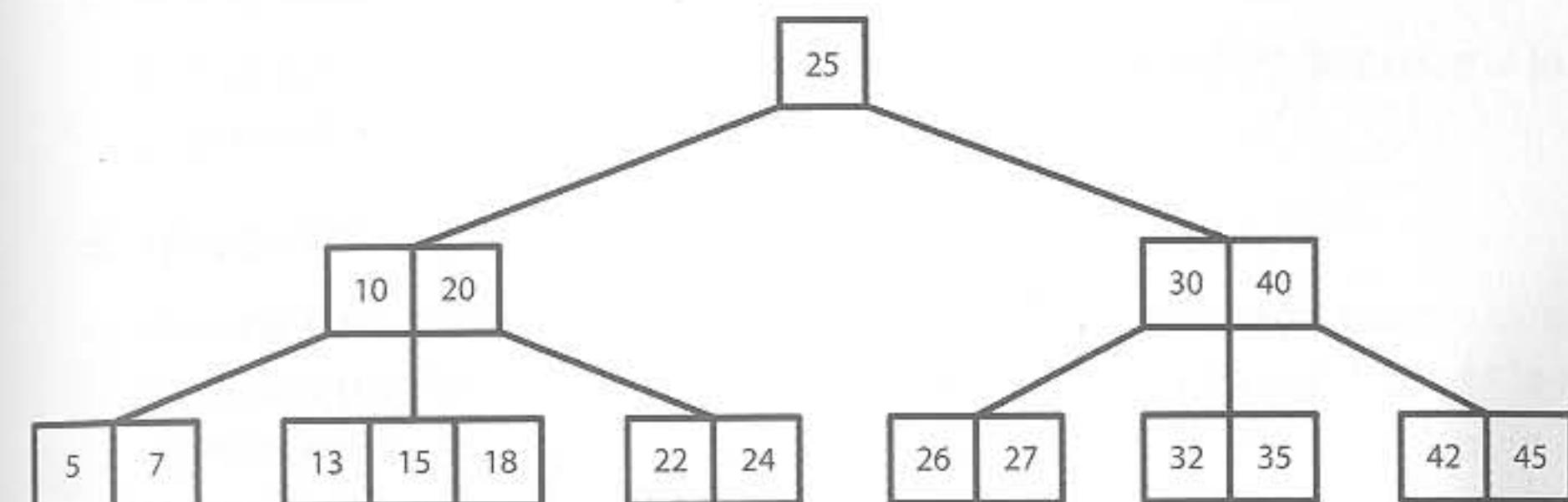
Transformar el siguiente árbol 2-3-4 en su correspondiente árbol rojo-negro. Generar los dos árboles posibles que salen al aplicar los distintos criterios de transformación.



## 3. (Ejercicio 33, página 109)

Dado el siguiente árbol B con  $m = 5$ , realizar el borrado de los elementos 25 y 24 (en este orden) indicando el número de rotaciones y combinaciones que han sido necesarias.

NOTA: Para borrar un ítem que esté en un nodo interior, se sustituirá por el mayor de su rama izquierda. En el caso de disponer de dos hermanos, se consultará el hermano de la derecha.



## 8.2. Examen 2

### 8.2.1. Test

1. (Pregunta 2, página 24)

Una operación consultora devuelve un valor del tipo definido.

2. (Pregunta 16, página 46)

En layering los métodos de la clase derivada pueden acceder a la parte pública de la clase base.

3. (Pregunta 24, página 56)

En una cola representada a partir de una lista enlazada simple con un único puntero al principio de la lista (*cabeza de la cola*), todas las operaciones de la cola (*Cabeza*, *Encolar*, *Desencolar* y *EsVacia*) se realizan en tiempo constante.

4. (Pregunta 50, página 75)

Un árbol AVL es un árbol binario de búsqueda en el que la diferencia de nodos entre el subárbol izquierdo y derecho es como máximo uno.

5. (Pregunta 63, página 87)

Existe un único árbol 2-3 de altura 3 que representa a las etiquetas del 1 al 9.

6. (Pregunta 64, página 88)

Las operaciones de inserción y borrado de un árbol 2-3 son las mismas que las de un árbol B de grado 3.

7. (Pregunta 76, página 102)

Todas las operaciones (*insercion*, *borrado* y *busqueda*) de un árbol rojo-negro se realizan en un tiempo  $O(\log_2 n)$ , siendo  $n$  el número de nodos.

8. (Pregunta 100, página 114)

Cuando implementamos un TAD Tabla de dispersión cerrada se usa una función de dispersión  $H$  tal que  $H(x)$  devolverá un valor desde

0 hasta  $B$ , siendo  $B$  el número finito de clases en las que dividimos el conjunto.

9. (Pregunta 108, página 117)

Sea el TAD Unión-Búsqueda implementado por medio de vectores de elementos. La operación

$$\text{union}(\text{Identif}, \text{Identif}) \rightarrow \text{Identif}$$

tiene una complejidad  $O(n)$ , siendo  $n$  el tamaño del vector.

10. (Pregunta 116, página 119)

Un montículo mínimo es un árbol binario lleno que además es árbol mínimo.

11. (Pregunta 117, página 119)

Para todo nodo de un árbol Leftist, se cumple que la altura de su hijo izquierdo es mayor o igual que la de su hijo derecho.

12. (Pregunta 129, página 138)

Al representar un grafo dirigido de  $N$  vértices y  $K$  aristas con una matriz de adyacencia, la matriz será simétrica respecto la diagonal principal.

13. (Pregunta 130, página 138)

Los arcos de retroceso de un recorrido en profundidad de un grafo dirigido nos indican la presencia de un ciclo.

14. (Pregunta 36, página 69)

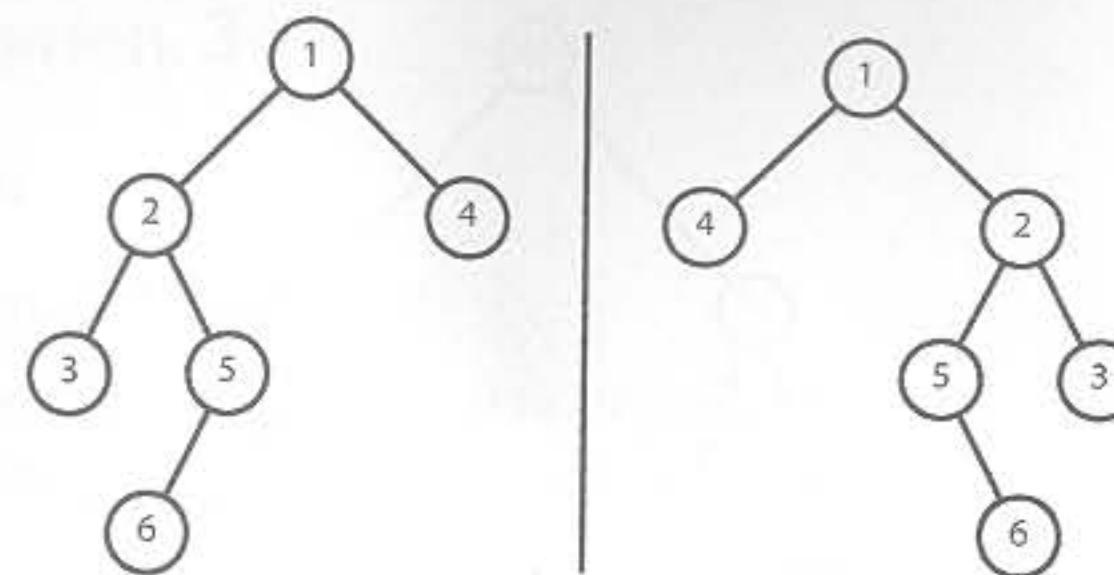
En un árbol binario lleno, el camino mínimo de la raíz (longitud del camino más corto de ese nodo hasta un árbol vacío) es igual a la altura del árbol.

## 8.2.2. Ejercicios

1. (Ejercicio 2, página 30)

Sea un árbol binario cuyas etiquetas son números naturales. Especificar la sintaxis y la semántica de la operación simétricos que comprueba que 2 árboles binarios son simétricos.

NOTA: Este es un ejemplo de 2 árboles simétricos



2. (Ejercicio 41, página 139)

Modificar el siguiente algoritmo que realiza el recorrido en profundidad de un grafo no dirigido para que además de realizar este recorrido, escriba el tipo de cada arista del grafo según la clasificación de aristas del bosque extendido en profundidad.

```

ALGORITMO DFS(v: TVértice, G: TGrafo, visitados:
    TConjuntoDeTVértice)
VAR w: TVértice;
MÉTODO
    Visitar(v);
    Insertar(visitados, v);
    para todo w ∈ AdyacenciaSalida(v, G) hacer
        si no Pertenece(visitados, w) entonces
            DFS(w, G, visitados);
        fsi
    fpara
FMÉTODO

```

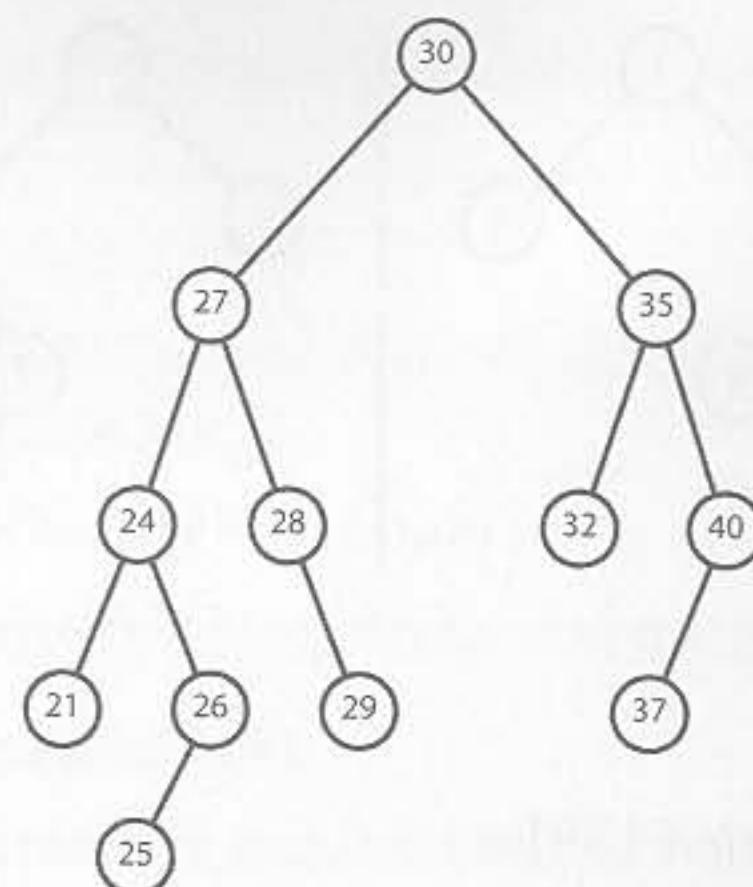
3. (Ejercicio 35, página 122)

En un DEAP inicialmente vacío insertar los siguientes elementos, indicando las transformaciones que se van realizando:

20, 7, 13, 31, 43, 55, 65, 40, 27, 29, 33, 46, 30, 1

4. (Ejercicio 26, página 79)

Haced el borrado de la etiqueta 32 del siguiente árbol AVL paso a paso, indicando en cada momento el factor de equilibrio de cada nodo y los tipos de rotación necesarios para reequilibrar.



## 8.3. Examen 3

### 8.3.1. Test

1. (Pregunta 17, página 46)

En herencia pública, la parte privada de la clase base es accesible desde los métodos de la clase derivada.

2. (Pregunta 3, página 24)

Para el tratamiento de errores en la especificación, se añaden funciones constantes que devuelven un valor del tipo que causa el error.

3. (Pregunta 25, página 56)

Una lista es una secuencia de cero o más elementos de cualquier tipo de la forma:  $e_p, e_{sig(p)}, \dots$

4. (Pregunta 31, página 66)

Un camino en un árbol es una secuencia  $a_1, a_2, \dots, a_s$  de árboles tal que para todo  $i \in \{1, \dots, s-1\}$ ,  $a_{i+1}$  es subárbol de  $a_i$ .

5. (Pregunta 37, página 69)

A partir del recorrido por niveles de un árbol binario completo se puede obtener el árbol al que representa.

6. (Pregunta 38, página 69)

El mayor ítem almacenado en un árbol binario de búsqueda siempre se encuentra en un nodo hoja.

7. (Pregunta 51, página 75)

En un árbol AVL siempre que se inserte una etiqueta hay que realizar una rotación.

8. (Pregunta 65, página 88)

En un árbol 2-3 la altura del árbol sólo aumenta cuando todas las hojas del árbol son de grado tres.

9. (Pregunta 77, página 102)

Un árbol rojo-negro es un árbol binario de búsqueda que representa a un árbol 2-3-4.

10. (Pregunta 90, página 112)

La complejidad temporal de la búsqueda de un elemento en un conjunto de cardinalidad  $n$ , representado como una lista, es  $O(n)$ .

11. (Pregunta 101, página 115)

En la dispersión cerrada sólo se producen colisiones entre claves sinónimas.

12. (Pregunta 109, página 117)

Sea el TAD Unión-Búsqueda implementado por medio de árboles representados como vectores. La operación

$\text{union}(\text{Conjunto}, \text{Conjunto}) \rightarrow \text{Conjunto}$

tiene una complejidad  $O(1)$ .

13. (Pregunta 118, página 120)

En un HEAP MÁXIMO los elementos de las hojas son los que tienen mayores valores en sus claves.

14. (Pregunta 35, página 68)

En un árbol binario, el camino mínimo de la raíz (longitud del camino más corto hasta un árbol vacío) es igual a la altura del árbol.

15. (Pregunta 103, página 115)

El proceso de búsqueda en un árbol de búsqueda digital es igual que el del árbol binario de búsqueda.

16. (Pregunta 131, página 138)

Al representar un grafo de  $N$  vértices y  $K$  aristas con una lista de adyacencia, la operación de hallar la adyacencia de entrada de un vértice, tiene una complejidad de  $O(K)$ .

### 8.3.2. Ejercicios

1. (Ejercicio 3, página 31)

Dada la siguiente especificación formal de operaciones de listas de acceso por posición:

```
crear() → lista
inscabeza(lista, item) → lista
longitud(lista) → natural
```

A) Explicar qué hace la operación  $X$ , cuya sintaxis y semántica aparecen a continuación:

```
X(lista) → lista
VAR i: item; l: lista;

X(crear()) = crear()
si(longitud(l) == 0) entonces
    X(inscabeza(l, i)) = crear()
sino
    X(inscabeza(l, i)) = inscabeza(X(l), i)
fsi
```

B) Simplificar la siguiente expresión: ( $IC = \text{inscabeza}$ )  
 $X(IC(IC(IC(IC(crear(), a), b), c), d))$

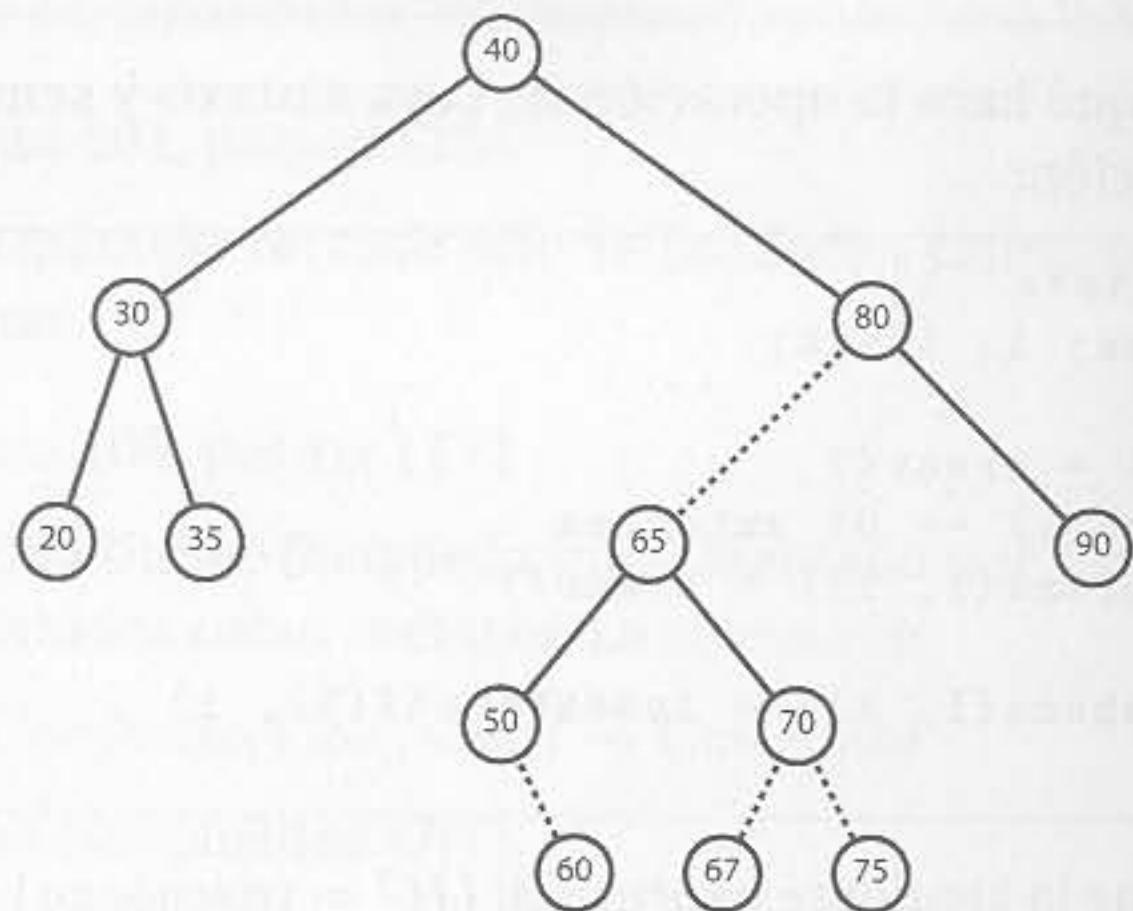
2. (Ejercicio 42, página 141)

Modificar el siguiente algoritmo que realiza el recorrido en anchura de un grafo no dirigido para que además de realizar este recorrido, escriba el tipo de cada arista del grafo según la clasificación de aristas del bosque extendido en anchura.

```
ALGORITMO BFS(v: TVértice, G: TGrafo, visitados:
    TConjuntoDeTVértice)
VAR w1, w2: TVértice; Q: TCola(TVértice);
MÉTODO
    visitados.Insertar(v);
    Q.Encolar(v);
    Visitar(v);
    mientras no Q.EsVacia() hacer
        w1 = Q.Cabeza();
        Q.Desencolar();
        para todo w2 ∈ AdyacenciaDeSalida(w1) hacer
            si no visitados.Pertenece(w2) entonces
                Visitar(w2);
                Q.Encolar(w2);
                visitados.Insertar(w2);
        fsi
    fpara
    fmientras
FMÉTODO
```

## 3. (Ejercicio 32, página 105)

Dado el siguiente árbol rojo-negro:



A) Realizar la inserción de la clave 72, detallando los cambios de color y rotaciones empleadas.

B) Escribir en C++ el código de la función

```
int TarbolRN::Altura234(void)
```

que devuelve la altura del árbol 2-3-4 equivalente al rojo-negro sobre el que trabaja la función. Añadir una breve explicación de 5 líneas como máximo de su funcionamiento. La representación del árbol rojo-negro y de su nodo será la siguiente:

```
class TarbolRN {
    Altura234(void)
    ...
private:
    Tnodo *p;
};
```

```
class Tnodo {
    int clave;
    TarbolRN iz, de;
    int colorIZ, colorDE;
    //Rojo=0, Negro=1
};
```

## 4. (Ejercicio 34, página 116)

Sea una tabla de dispersión cerrada de tamaño  $B = 11$  inicialmente vacía.

A) Insertar los siguientes elementos (en este orden estricto):

23, 14, 10, 15, 3, 5, 7, 8, 36, 47, 4

La estrategia de redispersión a utilizar es la que usa una segunda función de dispersión:

$$K(x) = (x \bmod (B - 1)) + 1.$$

B) ¿Qué valor debe tomar  $B$  para que se asegure que la estrategia de redispersión permita explorar todas las posiciones de la tabla?

## 8.4. Examen 4

### 8.4.1. Test

1. (Pregunta 40, página 70)

El máximo número de nodos en un nivel  $i - 1$  de un árbol binario es  $2^{i-2}$ ,  $i \geq 2$ .

2. (Pregunta 18, página 47)

Para que se ejecute el constructor de copia de la clase base, éste debe ser invocado explícitamente en la fase de inicialización de la clase derivada.

3. (Pregunta 52, página 75)

El mínimo número de nodos que ha de tener un árbol binario de altura 4 para ser equilibrado respecto a la altura es 7.

4. (Pregunta 44, página 73)

Suponiendo que tenemos un árbol binario de búsqueda lleno con  $n$  elementos, la búsqueda del elemento número  $n/2$  según la relación de orden se realiza en tiempo logarítmico.

5. (Pregunta 26, página 56)

Una lista de acceso por posición es una secuencia de cero o más elementos que pueden ser de distinto tipo.

6. (Pregunta 53, página 76)

Un árbol completo es un árbol completamente equilibrado.

7. (Pregunta 67, página 88)

Dado un árbol 2-3 con  $n$  elementos (*items*) con todos sus nodos del tipo 3-Nodo, la complejidad de la operación de búsqueda de un elemento es  $O(\log_3(n + 1))$ .

8. (Pregunta 79, página 103)

Todas las operaciones (inserción y búsqueda) de un árbol rojo-negro se realizan en un tiempo  $O(\log_2 n)$ , siendo  $n$  el número de elementos (*items*).

## 9. (Pregunta 86, página 108)

El nodo de un árbol B m-camino de búsqueda con  $m = 100$  puede tener como máximo 99 claves.

## 10. (Pregunta 89, página 112)

La representación de conjuntos mediante vectores de bits tiene una complejidad espacial proporcional al tamaño del conjunto universal.

## 11. (Pregunta 91, página 112)

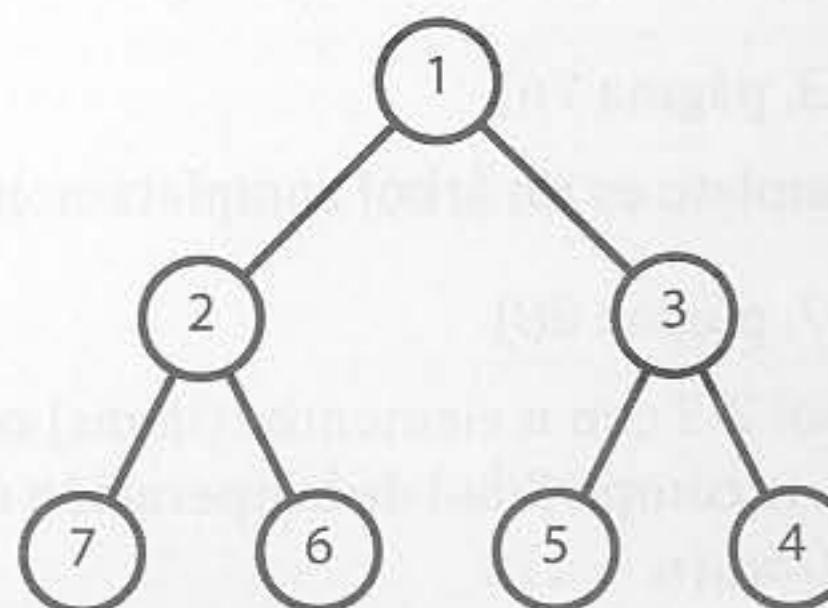
Cuando utilizamos una tabla de dispersión cerrada de tamaño  $B$ , el número de elementos del conjunto está limitado a  $B - 1$  elementos.

## 12. (Pregunta 110, página 118)

Sea el TAD Unión-Búsqueda implementado por medio de árboles representados como vectores aplicando la regla compensada para la unión. En:  $union(i, j)$ , el identificador  $i$  quedará como raíz del nuevo árbol, si su árbol tiene menos elementos que el árbol formado por el identificador  $j$ .

## 13. (Pregunta 119, página 120)

El siguiente árbol es un montículo mínimo y también es un leftist mínimo:



## 14. (Pregunta 120, página 120)

Para todo nodo de un árbol leftist, se cumple que la altura de su hijo izquierdo es menor que la de su hijo derecho.

## 15. (Pregunta 13, página 42)

La complejidad temporal de las operaciones de inserción y búsqueda en un árbol de búsqueda digital es  $O(n)$  siendo  $n$  el número de elementos del conjunto.

## 16. (Pregunta 132, página 138)

Un árbol extendido de un grafo siempre es un árbol binario.

## 8.4.2. Ejercicios

## 1. (Ejercicio 4, página 32)

Sea el conjunto de los números naturales con las operaciones de cero y sucesor:

MÓDULO	NATURAL
USA	Bool
TIPO	natural
OPERACIONES	
cero:	$\rightarrow$ natural
suc:	natural $\rightarrow$ natural

Define la semántica de las nuevas operaciones  $==$  y  $<=$  que permitan una ordenación de los elementos del conjunto:

## SINTAXIS

$==$ : natural, natural  $\rightarrow$  Bool

$<=$ : natural, natural  $\rightarrow$  Bool

## 2. (Ejercicio 27, página 82)

Construye un árbol binario de búsqueda lleno que contenga los números enteros del 1 al 15. Utilizando el árbol obtenido, borra realizando todos los pasos el elemento 7, suponiendo que:

A) es un árbol AVL

B) es un árbol 2-3 (con todos los nodos del tipo 2-Nodo)

C) un árbol 2-3-4 (con todos los nodos del tipo 2-Nodo)

## 3. (Ejercicio 36, página 126)

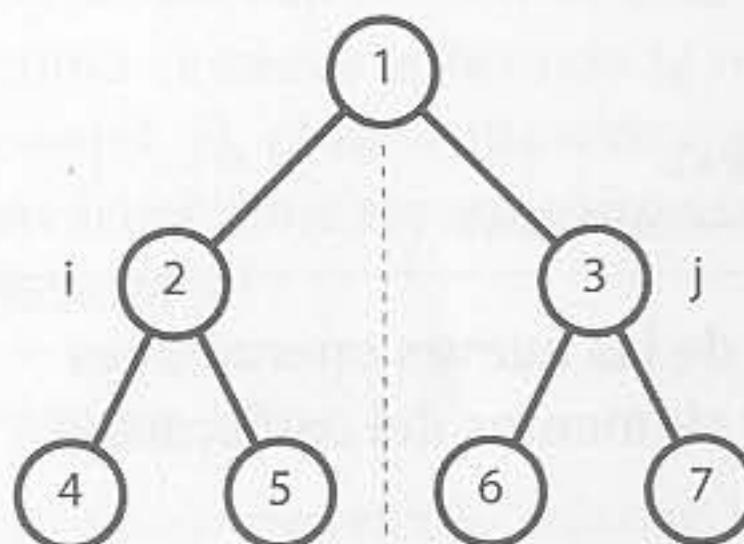
Sea un montículo doble (DEAP) en el que los índices  $i$  y  $j$  referencian nodos simétricos de los montículos mínimo y máximo respectivamente según el recorrido por niveles del árbol (tal y como se muestra en el

siguiente árbol, desde 1 hasta  $n$ , siendo  $n$  el número de elementos del DEAP). Suponiendo que  $i$  está en el nivel  $k$  del árbol, obtén razonadamente las siguientes fórmulas:

A) La que obtiene  $j$  (su nodo simétrico) a partir de  $i$  y de  $k$ .

B) La condición por la que se sabe que un elemento se encuentra en el montículo mínimo a partir de  $k$  y de su numeración,  $t$ , según el recorrido por niveles (desde 1 hasta  $n$ ). Por ejemplo, para el nodo número  $t = 4$  que se encuentra en el nivel  $k = 3$ , se sabe que se encuentra en el montículo mínimo. Y para  $t = 6$  sabemos que se encuentra en el montículo máximo.

C) Igual que en la pregunta B, a partir de  $k$  saber cuándo  $t$  se encuentra en el montículo máximo.



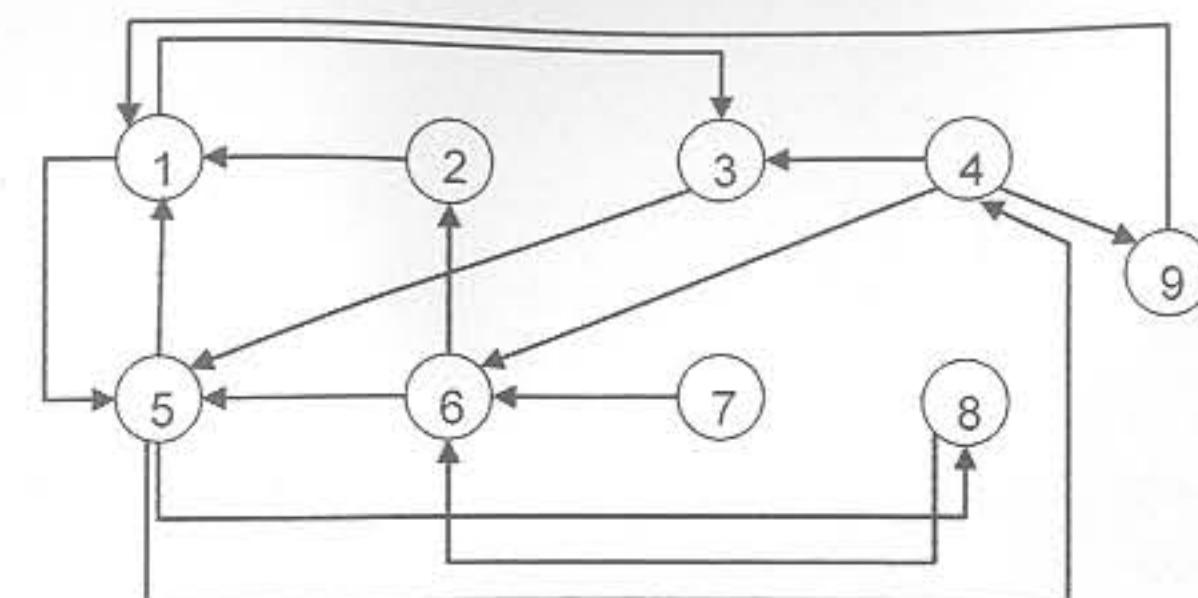
#### 4. (Ejercicio 43, página 142)

Dado el siguiente grafo dirigido, obtén:

A) El bosque extendido en profundidad comenzando por el vértice 1 expresado en forma de árbol (ante la posibilidad de elegir continuar por un nodo u otro, elegiremos siempre el que menor etiqueta tenga).

B) Identifica los distintos tipos de arcos del grafo (arco de árbol (A), de avance (Av), de retroceso (R), de cruce (Cr)).

C) Extrae las componentes fuertemente conexas del grafo anterior. ¿Es un grafo fuertemente conexo?



## Solución de las preguntas de tipo test de los exámenes

En este capítulo se presentan los mismos exámenes que los presentados en el Capítulo 8, pero con la correspondiente solución. Además, cada pregunta contiene una referencia a la explicación que indica cuál es la respuesta correcta y por qué.

## 9.1. Examen 1

1. Verdadero (Pregunta 128, página 138)
2. Verdadero (Pregunta 82, página 109)
3. Falso (Pregunta 62, página 87)
4. Falso (Pregunta 115, página 119)
5. Verdadero (Pregunta 71, página 91)
6. Falso (Pregunta 35, página 68)
7. Falso (Pregunta 75, página 102)
8. Falso (Pregunta 49, página 75)
9. Verdadero (Pregunta 102, página 115)
10. Falso (Pregunta 1, página 24)
11. Verdadero (Pregunta 89, página 112)
12. Verdadero (Pregunta 15, página 46)
13. Falso (Pregunta 107, página 117)
14. Falso (Pregunta 99, página 114)

## 9.2. Examen 2

1. Falso (Pregunta 2, página 24)
2. Verdadero (Pregunta 16, página 46)
3. Falso (Pregunta 24, página 56)
4. Falso (Pregunta 50, página 75)
5. Falso (Pregunta 63, página 87)
6. Verdadero (Pregunta 64, página 88)
7. Verdadero (Pregunta 76, página 102)
8. Falso (Pregunta 100, página 114)
9. Verdadero (Pregunta 108, página 117)
10. Falso (Pregunta 116, página 119)
11. Falso (Pregunta 117, página 119)
12. Falso (Pregunta 129, página 138)
13. Verdadero (Pregunta 130, página 138)
14. Verdadero (Pregunta 36, página 69)

## 9.3. Examen 3

1. Falso (Pregunta 17, página 46)
2. Verdadero (Pregunta 3, página 24)
3. Falso (Pregunta 25, página 56)
4. Verdadero (Pregunta 31, página 66)
5. Verdadero (Pregunta 37, página 69)
6. Falso (Pregunta 38, página 69)
7. Falso (Pregunta 51, página 75)
8. Falso (Pregunta 65, página 88)
9. Verdadero (Pregunta 77, página 102)
10. Verdadero (Pregunta 90, página 112)
11. Falso (Pregunta 101, página 115)
12. Falso (Pregunta 109, página 117)
13. Falso (Pregunta 118, página 120)
14. Falso (Pregunta 35, página 68)
15. Falso (Pregunta 103, página 115)
16. Verdadero (Pregunta 131, página 138)

## 9.4. Examen 4

1. Verdadero (Pregunta 40, página 70)
2. Verdadero (Pregunta 18, página 47)
3. Verdadero (Pregunta 52, página 75)
4. Falso (Pregunta 44, página 73)
5. Falso (Pregunta 26, página 56)
6. Falso (Pregunta 53, página 76)
7. Verdadero (Pregunta 67, página 88)
8. Verdadero (Pregunta 79, página 113)
9. Verdadero (Pregunta 86, página 102)
10. Verdadero (Pregunta 89, página 112)
11. Falso (Pregunta 91, página 112)
12. Falso (Pregunta 110, página 118)
13. Verdadero (Pregunta 119, página 120)
14. Falso (Pregunta 120, página 120)
15. Falso (Pregunta 13, página 42)
16. Falso (Pregunta 132, página 138)

## Bibliografía recomendada

A continuación se incluye una serie de libros recomendados agrupados en tres categorías: C++, Especificación algebraica y Tipos abstractos de datos y C++.

### C++

- Sergio Luján Mora. *C++ paso a paso*. Publicaciones de la Universidad de Alicante, 2006. ISBN: 84-7908-888-5
- Harvey M. Deitel, Paul J. Deitel. *Cómo programar en C++*. Pearson Educación, 2003. ISBN: 970-26-0254-8
- Harvey M. Deitel, Paul J. Deitel. *Cómo programar en C/C++ y Java*. Prentice Hall, 2004. ISBN: 970-26-0531-8

### Especificación algebraica

- Ricardo Peña Marí. *Diseño de programas: Formalismo y abstracción*. Pearson Educación, 2005. ISBN: 84-205-4191-5
- Xavier Franch Gutiérrez. *Estructuras de datos: Especificación, diseño e implementación*. Edicions UPC, Colección Politext, 2001. ISBN: 84-8301-002-X

### Tipos abstractos de datos y C++

- Antonio Garrido Carrillo, Joaquín Fernández Valdivia. *Abstracción y estructuras de datos en C++*. Delta, Publicaciones Universitarias, 2006. ISBN: 84-96477-26-6
- Frank M. Carrano, Janet J. Prichard. *Data abstraction and problem solving with C++ : walls and mirrors*. Addison-Wesley, 2002. ISBN: 0-201-74119-9
- Adam Drozdek. *Data structures and algorithms in C++*. Brooks/Cole, 2001. ISBN: 0-534-37597-9
- Ellis Horowitz, Sartaj Sahni, Dinesh Mehta. *Fundamentals of data structures in C++*. Computer Science Press, 1997. ISBN: 0-7167-8292-8
- Walter Savitch. *Resolución de problemas con C++ : el objetivo de la programación*. Pearson Educación, 2000. ISBN: 968-444-416-8

## Sobre los autores

**Sergio Luján Mora** Profesor Titular de Universidad del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante. Obtuvo su grado de Doctor Ingeniero en Informática en la Universidad de Alicante (España) en el año 2005. Su tesis doctoral estaba centrada en el modelado multidimensional de los almacenes de datos con UML.

Sus temas principales de investigación incluyen la accesibilidad y la usabilidad web, el desarrollo de las aplicaciones web, el diseño de almacenes de datos, el e-learning y la programación orientada a objetos. Es profesor de las asignaturas "Programación y Estructuras de Datos" y "Programación en Internet" en las titulaciones de Ingeniería Informática de la Universidad de Alicante.

Ha publicado diversos trabajos de investigación en diversas conferencias y revistas de alto impacto tanto nacionales como internacionales. Entre ellos destacan los congresos ER, UML o DOLAP. Entre las revistas internacionales indizadas en el JCR destacan DKE, JCS o JDBM. Además, ha publicado diversos libros relacionados con la programación y diseño de páginas web.

Ha participado en diversos proyectos de investigación financiados por entidades públicas, como "Entornos hipermediales para la migración y desarrollo de código", financiado por el Ministerio de Ciencia y Tecnología y "Desarrollo de una Metodología de aseguramiento de usabilidad y accesibilidad de Interfaces Web (Demeter)", financiado por la Consellería de Educación de la Generalitat Valenciana.

# Índice alfabético

## A

adyacencia de entrada, 138  
adyacencia de salida, 138, 143, 146  
altura de un árbol, 68, 69, 91, 158  
árbol 8-9, 82, 87-89, 103, 108, 167, 169  
árbol 8-9-10, 82, 89-92, 96, 98, 100, 102-104, 106-108, 121, 164, 169  
árbol AVL, 75-80, 82, 169  
árbolB, 88, 107-109, 168  
árbol binario, 28, 30, 41, 68, 70, 79, 103, 108, 121, 138, 139, 167, 169  
árbol binario completo, 68-70, 87, 121  
árbol binario de búsqueda, 48, 69, 73, 75, 102, 115  
árbol binario de búsqueda lleno, 73, 82, 122, 167, 169  
árbol binario enhebrado, 74  
árbol binario lleno, 68-70, 119, 122, 158

árbol completamente equilibrado, 76, 77, 167  
árbol completo, 76, 167  
árbol de búsqueda digital, 42, 115, 169  
árbol de Fibonacci, 76, 77  
árbol extendido de un grafo, 138, 139, 169  
árbol general, 67  
árbol izquierdista, 119-122, 127, 130, 168  
árbolleftist, véase árbol izquierdista  
árbol multicamino de búsqueda, 67  
árbol rojo-negro, 102-105, 164, 167  
arco de avance, 139, 140, 143, 170  
arco de cruce, 140, 143, 170  
arco de retroceso, 138, 140, 142, 143, 158, 170  
arco del árbol, 140, 141, 143, 170

## B

bosque extendido en anchura, 141

Es autor del libro "C++ paso a paso", editado por Publicaciones de la Universidad de Alicante en el año 2006.

Correo electrónico: [sergio.lujan@ua.es](mailto:sergio.lujan@ua.es)

Twitter: [sergiolujanmora](#)

Página personal: <http://www.dlsi.ua.es/~slujan/>

**Antonio Ferrández Rodríguez** Profesor Titular de Universidad del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante. Obtuvo su grado de Doctor Ingeniero en Informática en la Universidad de Alicante (España) en el año 1998. Su tesis doctoral estaba centrada en el procesamiento del lenguaje natural y la resolución de problemas lingüísticos, como por ejemplo la anáfora o la elipsis, para lo que se utilizó muchos de los conceptos de tipos de datos presentados en este libro.

De su experiencia docente relacionada con este libro se destaca la impartida en las asignaturas de "Tipos Abstractos de Datos", "Programación y Estructuras de Datos" y "Explotación de la Información", en las titulaciones de Ingeniería Informática de la Universidad de Alicante desde el año 1993.

Es autor del libro "Tipos abstractos de datos", editado por Librería Compás en el año 1994.

Correo electrónico: [antonio@dlsi.ua.es](mailto:antonio@dlsi.ua.es)

Página personal: <http://www.dlsi.ua.es/~antonio/antonio.html>

**Jesús Peral Cortés** Profesor Titular de Universidad del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante. Obtuvo su grado de Doctor Ingeniero en Informática en la Universidad de Alicante (España) en el año 2001. Su tesis doctoral estaba centrada en el procesamiento del lenguaje natural y la resolución de problemas lingüísticos, como el tratamiento de la anáfora pronominal y su generación en otros idiomas.

De su experiencia docente relacionada con este libro se destaca la impartida en las asignaturas de "Tipos Abstractos de Datos" y "Programación y Estructuras de Datos" en las titulaciones de Ingeniería Informática de la Universidad de Alicante desde el año 1995.

Ha sido director del programa de doctorado "Aplicaciones de la Informática" que imparte el Departamento de Lenguajes y Sistemas Informáticos

ticos de 2002 a 2009 y secretario académico del mismo departamento desde 2009 hasta 2013.

Miembro de la "Sociedad Española para el Procesamiento del Lenguaje Natural". Ha desarrollado e impartido numerosas ponencias y cursos relacionados con el Procesamiento del Lenguaje Natural (PLN), concretamente con los sistemas de búsqueda de respuestas y la resolución de problemas lingüísticos. Investigador de varios proyectos de I+D tanto públicos como privados. Autor de más de 40 artículos en revistas y congresos tanto nacionales como internacionales relacionados con la temática del PLN. Ha sido miembro de comités organizadores de diferentes conferencias nacionales e internacionales.

Correo electrónico: [jperal@dlsi.ua.es](mailto:jperal@dlsi.ua.es)

Página personal: <http://www.dlsi.ua.es/~jperal/jperal.html>

**Antonio Requena Jiménez** Profesor Asociado del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante. Obtuvo su grado de Ingeniero en Informática en la Universidad de Alicante (España) en el año 1997.

Ha sido profesor de diferentes asignaturas y entre ellas destaca "Programación y Estructuras de Datos" desde el año 1997.

Coautor de varios artículos científicos y participante en diferentes congresos relacionados con sistemas de reconocimiento de formas no supervisadas y algoritmos de búsqueda. Es socio cofundador de la empresa IT Robotics, S.L. dedicada a la robótica industrial y visión artificial. Esta empresa ha servido para la realización práctica de diferentes investigaciones realizadas en el entorno del reconocimiento de formas y que han dado lugar a más de 20 patentes y modelos de utilidad.

Correo electrónico: [requena@dlsi.ua.es](mailto:requena@dlsi.ua.es)

bosque extendido en profundidad, 138, 139, 142, 143, 146, 170  
breadth first search, véase recorrido en anchura de un grafo

**C**

cabeza de la cola, 56  
camino de búsqueda, 77, 92, 96  
camino en un árbol, 66, 68, 69, 158  
ciclo en un grafo, 138, 139, 146, 158  
cima de la pila, 57  
clase base, 46, 47, 167  
clase derivada, 46-48, 167  
claves sinónimas, 113, 115  
cola, 56  
colisión de claves, 113, 115  
combinación, 96, 98  
complejidad espacial, 112, 168  
complejidad temporal, 42, 56, 57, 112, 113, 115, 116, 138, 169  
conjunto, 32, 169  
conjunto universal, 112, 118, 168  
constructor, 48  
constructor de copia, 46, 47, 167

**D**

deap, véase montículo doble  
depth first search, véase recorrido en profundidad de un grafo  
destructor, 48  
diccionario, 133  
digrafo, 144

**E**

escala de complejidades, 42  
especificación, véase especificación algebraica  
especificación algebraica, 24-27, 31

**F**

factor de carga, 113  
factor de equilibrio, 75, 79  
fondo de la cola, 56  
función constante, 24  
función de redispersión, 112-114

**G**

grado de un árbol, 67  
grafo acíclico dirigido, 138, 139  
grafo dirigido, 138, 139, 142, 146, 170  
grafo fuertemente conexo, 143, 170  
grafo no dirigido, 139, 141, 144

**H**

heap máximo, véase montículo máximo  
heap mínimo, véase montículo mínimo  
herencia privada, 47  
herencia pública, 46  
hexadecimal, 133

**I**

implementación, 27, 56, 57

**L**

layering, 46  
lista, 56, 57

lista de acceso por posición, 31, 33, 39, 56, 167  
lista de adyacencia, 138, 144  
lista enlazada, 56, 57  
lista enlazada simple, 56  
lista ordenada, 50, 56  
lista ordenada doblemente enlazada, 50

**M**

matriz de adyacencia, 138, 158  
montículo doble, 122, 126, 134, 169  
montículo máximo, 119, 120, 126, 169  
montículo mínimo, 119-121, 126, 168, 169

**O**

operación auxiliar, 26  
operación constructora generadora, 25  
operación consultora, 24  
operador corchete, 47

**P**

parte privada, 46-48  
parte pública, 46, 47  
pila, 36, 56  
private, 47  
protected, 47, 48  
public, 47  
puntero this, véase this

**R**

recorrido de un árbol, 68-70  
recorrido en anchura de un grafo, 141

recorrido en inorden, 68-70, 74, 122  
recorrido en postorden, 68-70  
recorrido en preorden, 68, 69, 103  
recorrido en profundidad, 69  
recorrido en profundidad de un grafo, 138-140, 143, 145  
recorrido por niveles, 68, 69, 98  
rotación, 75-77, 79-81, 85, 88, 96-98, 100

**S**

semántica, 25, 27-32, 169  
sintaxis, 25, 27, 28, 30, 31  
sobrecarga de operador, 47

**T**

tabla de dispersión abierta, 113  
tabla de dispersión cerrada, 112-116, 165, 168  
TAD, véase tipo abstracto de datos this, 48  
tipo abstracto de datos, 24-27  
trie, 115, 116, 133

**U**

unión-búsqueda, 117-119, 168

**V**

valor del tipo, 24  
vector, 57  
vector de bits, 112, 117, 168  
vector ordenado, 56

## **Ejercicios resueltos sobre Programación y estructuras de datos**

En el libro se recogen una colección de ejercicios y preguntas de test resueltos que abarcan todos los tipos de datos estudiados en el temario de la asignatura. Se engloban en cuatro grandes bloques: tipos lineales (vectores, listas, pilas y colas); tipo árbol (árbol binario, árbol binario de búsqueda, árbol enhebrado, árbol AVL, árbol 2-3, árbol 2-3-4, árbol B y árbol rojo-negro); tipo conjunto (tablas de dispersión y colas de prioridad); tipo grafo.

El problema fundamental en el diseño e instrumentación de grandes proyectos o aplicaciones informáticas es reducir su complejidad. Los atributos que permiten simplificar dicha complejidad son los siguientes: legibilidad, corrección, eficiencia, facilidad de mantenimiento y reutilización. A través de la abstracción de datos se pueden obtener estos objetivos.

Los autores, pertenecientes al Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante, tienen una dilatada experiencia docente y en estas páginas intentan que el lector adquiera las técnicas y habilidades necesarias para poder resolver con éxito cualquier ejercicio planteado sobre estructuras de datos.

 PUBLICACIONES  
UNIVERSIDAD DE ALICANTE



<http://publicaciones.ua.es>