UNIVERSIDAD DE DEUSTO

Facultad de Informática

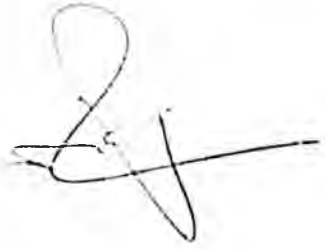
DISEÑO DE UN ARRAY BIDIMENSIONAL DINÁMICO IMPLEMENTADO MEDIANTE LISTAS ENLAZADAS Y ÁRBOLES AVL

Tesis doctoral presentada por Iñaki VARELA LEGARRETA

Dirigida por el Dr. Máximo LLAGUNO ELLACURÍA

El Director

El Doctorando



Bilbao, Septiembre de 1994

AGRADECIMIENTOS

A Arantza, mi mujer, por la paciencia demostrada y por el cariño con que me ha cuidado durante todo el tiempo de elaboración de esta Tesis.

A mis padres, Emilio y Ana Mª. Su ejemplo de sacrificio y esfuerzo ha sido punto de referencia constante a lo largo de mi carrera.

A mis hermanos, Ana, Emilio e Itziar, por los ánimos que me han dado.

A mi Director, Patxi Llaguno Ellacuria. Me regaló el libro de Wirth "Algoritmos + Estructuras de Datos = Programas", cuya lectura fue el punto de partida de esta Tesis. Su sabia experiencia y rigor científico me han resultado imprescindibles.

A José María López-Arcas y Lostalet, Doctor en CC.EE. y Empresariales y en Derecho. Trabajar a su lado supuso para mí una experiencia inolvidable, y un enriquecimiento personal y profesional continuo.

A Juan Ramón Fernández Abiega, Doctor en Informática, por su apoyo constante, por su enorme capacidad de trabajo y por todos los ratos estupendos que hemos pasado juntos.

A mis alumnos y colegas de la Facultad de Informática, y a mis compañeros de Norbolsa. Mi deber diario con ellos me ha servido de estímulo constante.

A todos los que me han ayudado en los últimos detalles, en especial, Mª José Gil, Amaya Miñambres, Tontxu Campos y Jorge Zabalbeitia, y a todos los que, de una u otra forma, han colaborado y que sería imposible citar nominalmente.

Para todos, mi sincero reconocimiento.

ÍNDICE

Resumen. III

Relación de figuras. VI

1. Introducción. 1

2. Implementación del ABD mediante Listas Enlazadas Simples y Dobles. 8

2.1. Listas Enlazadas. Definición y Tipos. 8

2.2. Estructura del ABD en base a listas. 15

2.3. Operaciones de Manipulación del ABD. 23

2.3.1. Recorrido por filas. 27
2.3.2. Recorrido por columnas. 29
2.3.3. Acceso a un elemento referenciado por sus índices. 31
2.3.4. Búsqueda de un elemento (o todos) en base a un campo de información. 39
2.3.5. Inserción de un elemento. 43
2.3.6. Borrado de un elemento referenciado por sus índices. 54
2.3.7. Borrado de un elemento (o todos) en base a un campo de información. 60
2.3.8. Modificación de un elemento referenciado por sus índices. 64
2.3.9. Modificación de un elemento (o todos) en base a un campo de información. 68
2.3.10. Eliminación de una fila. 71
2.3.11. Eliminación de una columna. 74
2.3.12. Modificación de un índice de fila. 77
2.3.13. Modificación de un índice de columna. 84

3. Implementación del ABD mediante Árboles Binarios de Búsqueda Equilibrados AVL. 91

3.1. Árboles Binarios. Definición y Terminología. 91

3.2. Árbol Equilibrado. AVL. 94

3.3. Replanteamiento de la implementación del ABD. 104

3.4. Estructura del ABD en base a árboles AVL. 106

3.5. Operaciones de Manipulación del ABD. 111

3.5.1. Recorrido por filas. 112
3.5.2. Recorrido por columnas. 114
3.5.3. Acceso a un elemento referenciado por sus índices. 116
3.5.4. Búsqueda de un elemento (o todos) en base a un campo de información. 122
3.5.5. Inserción de un elemento. 125
3.5.6. Borrado de un elemento referenciado por sus índices. 136
3.5.7. Borrado de un elemento (o todos) en base a un campo de información. 142
3.5.8. Modificación de un elemento referenciado por sus índices. 145
3.5.9. Modificación de un elemento (o todos) en base a un campo de información. 148
3.5.10. Eliminación de una fila. 150
3.5.11. Eliminación de una columna. 153
3.5.12. Modificación de un índice de fila. 156
3.5.13. Modificación de un índice de columna. 161

4. Conclusiones. 167

4.1. Análisis de las aportaciones originales de la investigación. 167

4.2. Lineas futuras de interés. 170

5. Bilbliografía. 172

6. Anexos. 177

A. Algoritmos en pseudocódigo de las operaciones de manejo del ABD implementado mediante listas enlazadas. 177

B. Algoritmos en pseudocódigo de las operaciones de manejo del ABD implementado mediante árboles AVL. 212

C. Codificación en Pascal del TAD ABD implementado mediante listas y mediante árboles AVL. 255

RESUMEN

Un array o matriz bidimensional tradicional es una estructura utilizada para el almacenamiento de información homogénea dispuesta en forma tabular, que garantiza el acceso inmediato a cada elemento de la misma en base a dos índices que referencian su posición. Esta estructura presenta los problemas siguientes: su naturaleza estática exige prefijar su dimensionamiento previamente a la incorporación de los elementos; la definición de las series de índices tiene restricciones que derivan en la infrautilización del espacio; se producen anomalías por accesos incorrectos con índices fuera de su rango; y el desperdicio de espacio en memoria originado, para matrices con un bajo porcentaje de ocupación, junto con tiempos de respuesta elevados en procesos de recorrido. En este trabajo de investigación se ha diseñado una nueva estructura: el Array Bidimensional Dinámico (ABD), que es un tipo abstracto de datos construido en base a punteros, lo que determina su carácter versátil. Esta estructura supera todos los inconvenientes del array estático, permitiendo además su reajuste dinámico en tiempo de ejecución en función de las actualizaciones de sus componentes. El mayor rendimiento se obtiene en matrices poco densas debido a la reducción en el espacio de almacenamiento y en el tiempo de ejecución de los procesos, al no existir posiciones reservadas inutilizadas. En primer lugar, se ha implementado el ABD en base a listas enlazadas ortogonales, constituyendo una red de listas dobles con inserción al final entrelazadas para representar los elementos y listas simples para los índices. En las listas, la búsqueda de un componente presenta un orden O(n), lo que significa que el tiempo es proporcional al número de elementos procesados. Los tratamientos de búsqueda intervienen en bastantes operaciones de manejo de la estructura, por lo que para mejorar su rendimiento se ha diseñado una segunda implementación del ABD utilizando árboles binarios de búsqueda equilibrados AVL para registrar los elementos y los índices. Estos árboles garantizan la búsqueda de un componente en un tiempo máximo del orden $O(\log_2 n)$, permitiendo mejorar los tiempos de respuesta en las operaciones que manejan la estructura. Estas operaciones de manipulación se definen y desarrollan en ambas implementaciones.

ABSTRACT

A bidimensional array or matrix is a structure used for storing homogeneous information displayed in a tabular way. It ensures an inmediate access to each item by two indices which provide a reference for its position. This structure presents the following problems: its static nature requires establishing its dimension before incorporating the elements; the definition of the series of indices is restricted, what brings about poor use of the space; faults may occur due to wrong access through indices out of rank; and space in memory may be wasted for matrixes with a low percentage of occupation, as well as a long response time in round processing. In this research work a new structure has been designed, the Bidimensional Dynamic Array (BDA), which is an abstract type of data based on pointers, what produces its versatil nature. This structure leaves behind all the problems of the static array, providing a dynamic readjustment in time of running, depending on how the components are updated. The highest performance is obtained in sparse matrixes due to a reduction in storing space and in time of data processing, due to only are reserved the used positions. Firstly, the BDA has been implemented based on orthogonal linked lists, forming a network of double with final insertion interconnected lists to represent the elements and simple lists for the indices. On the lists, the search for an element presents an order O(n) what means that time is proportional to the number of elements processed. The search treatments take part in a number of operations to handle the structure. Therefore, in order to improve its performance, a second implementation of BDA has been designed, using AVL balanced searching binary trees to record the elements and the indices. These trees ensure the search for a component in a maximum time of the order $O(\log_2 n)$, improving the response time in operations to handle the structure. These manipulation operations are defined and developed in both implementations.

LABURPENA

Array-a, ohizko bi neurritako matrizea, alegia, tauleko irudiarekin azaltzen den mota bereko informazioa gordetzeko erabiltzen da. Modu honek edozein datura zuzeneko bidea eskeintzen du bere bi lekuko ezagurrietara jo ezkero. Nola nahi ere, egitura honek arazoak hauek, besteak beste, ditu: egonkorra denez, datuak sartu baino lehenago definitu egin behar dira neurriak; ezaugarrien mugapenak askotan tokiaren azpierabilpena erakartzen du; ezaugarriak neurritik kanpo daudenean biderapen arazoak azaltzen dira; memoria galdu egiten da taulek datu gutxi dutenean eta denbora asko behar da datuetara jotzeko. Ikerketa honetan egitura berri bat plazaratzen da, Bineurritako Array Dinamikoa (BAD), zeinek datuak antolatzen dituen geziak erabiliz, eta honek bizitasuna ekarriko lioke prozesuari. Egitura honek Ohizko Array-ak dituen mugak gainditzen ditu eta, aldi berean, datuetara heltzerako denbora murriztu egiten du elementuen guneak berez berritzen direlako. Emaitzik hoberenak, bai denboretan eta baita ere memorietan, lortzen dira datu gutxiko matrizeekin, zeren eta egitura honek ez bait du gordetzen inongo tokirik erabili gabe. Hasteko, BAD sortu da zerrenda ortogonalak erabiliz: elementuak kokatzen dira zerrenda bikoitzaren sare baten bitartez eta ezaugarriak zerrenda bakar batez. Zerrendetan, datu baten bilaketak O(z) neurria eskatzen du, hau da, bilaketa denbora datu kantitatearen arabera izango da. Egitura honetan datu bilaketa oso garrantzitsua denez, BAD hobetu egin da bilaketa orekatutako arbol binarioak (AVL motatakoak) erabiliz elementu eta datu ezaugarriak kokatzeko. Arbol hauek aukera ematen dute datuetara heltzea, gehienez, $O(\log_2 z)$ neurritako denboretan, eta, jakina denez, honek azkartu egiten du bilaketa. Ikerketak hau guzti hau garatu egiten du.

RELACIÓN DE FIGURAS

2.1.	Lista Enlazada Simple.	**8**
2.2.	Otros Tipos de Listas.	**14**
2.3.	Array Bidimensional Dinámico implementado mediante Listas.	**20**
3.1.	Árbol Binario de Búsqueda.	**93**
3.2.	Inserción en AVL. Rotación Simple.	**98**
3.3.	Inserción en AVL. Rotación Doble.	**99**
3.4.	Borrado en AVL. Caso 1.	**102**
3.5.	Borrado en AVL. Caso 2.	**102**
3.6.	Borrado en AVL. Caso 3a.	**102**
3.7.	Borrado en AVL. Caso 3b.	**103**
3.8.	Borrado en AVL. Caso 3c.	**103**
3.9.	Array Bidimensional Dinámico implementado mediante Árboles AVL.	**110**

1. INTRODUCCIÓN

Los datos, eje en torno al cual giran gran cantidad de conceptos y áreas de investigación, pueden organizarse de muy diferentes formas para su almacenamiento y posterior tratamiento.

Una **estructura de datos** es un modelo matemático y lógico, una manera conceptual, de organizar los datos.

Para elegir una estructura de datos en particular, se han de tener en cuenta principalmente dos factores. Primero debe ser lo suficientemente complejo como para mostrar la relación entre los datos y lo que éstos representan. Por el contrario, la estructura debe ser lo suficientemente simple para que los datos puedan ser procesados de forma eficiente [LIPS87].

Wirth resalta la importancia de las estructuras de datos al considerar que "*los programas, después de todo, son formulaciones concretas de algoritmos abstractos basadas en ciertas representaciones y estructuras de datos*" [WIR76].

Hoare en su tratado *'Notes on Data Structuring'* [HOA72] demostró que las decisiones sobre la estructuración de datos no pueden tomarse sin conocer los algoritmos que se aplican a los datos y, viceversa, la estructura y selección de los algoritmos con frecuencia depende mucho de la estructura de los datos subyacentes. En resumen, la composición de programas y las estructuras de datos están interrelacionados y son inseparables [WIR76] [WIR87].

Las estructuras de datos existen con independencia del lenguaje o notación utilizada para expresarlas. Mediante el sistema notacional que se utilice se especifican las estructuras, asociando a cada una un tipo y un conjunto de operaciones de manipulación. Así cada estructura de datos se distingue de las demás no solo por su tipo, sino también por la forma en que se manipula.

Un concepto relacionado con lo anterior es el de abstracción de datos o **tipo abstracto de datos** (***TAD***), que consiste en una forma de definir los datos en la que sólo se consideran un tipo de datos junto con un conjunto de operaciones, que definen cómo se manipulan los datos de ese tipo [BIS89].

En este concepto de abstracción de datos es importante destacar la idea de la separación entre la forma en que está internamente implementado el tipo de datos y la forma en que se maneja externamente dicho tipo. Esto se conoce como encapsulación u ocultación de la información [GAL93].

Mediante la encapsulación, la implementación del tipo abstracto de datos está oculta, permaneciendo sólo visible su especificación. Así para utilizar un tipo abstracto de datos basta con conocer su especificación o interfaz ("qué hace" o su visión externa), sin tener conocimiento de su implementación concreta ("cómo lo hace" o su visión interna). Por lo tanto, cualquier cambio que se realice en su implementación no afectará a los programas que hagan uso del TAD Por el contrario, un cambio en su interfaz sí repercutirá en los programas que lo utilicen, de ahí la importancia que tiene el realizar un diseño correcto del mismo, antes de ponerlo a disposición de los programas de usuario.

Las estructuras de datos se pueden situar en un nivel de abstracción inferior al de las abstracciones de datos, utilizándose para la implementación de éstas, de tal forma que una misma estructura de datos se puede utilizar para implementar varios tipos abstractos de datos distintos.

El **array** es probablemente la estructura de datos más simple y la mejor conocida, siendo en algunos lenguajes la única explícitamente disponible.

Un array, arreglo, tabla o vector es una estructura n-dimensional de elementos homogéneos, cada uno de los cuales se referencia por medio de n índices, uno por cada dimensión. Los elementos se almacenan en memoria en posiciones consecutivas, disposición que permite el acceso al azar a un elemento cualquiera en base a sus índices [DALE86] [WIR76].

Los arrays bidimensionales o matrices (arrays de dos dimensiones) son estructuras destinadas a almacenar información homogénea dispuesta en filas y columnas. Cada dato ocupa una posición referenciada por medio de un índice para filas y otro índice para columnas.

Un array bidimensional se declara en los lenguajes de programación tradicionales mediante tres parámetros [CAIS3]:

- *Nombre* del array.
- *Conjunto de índices* para cada una de las dos dimensiones.
- *Tipo de datos* que va a contener (tipo base).

En lo referente a los conjuntos de índices es donde surgen los inconvenientes y limitaciones de esta estructura, que serán las premisas de partida de esta Tesis para desarrollar una estructura similar conceptualmente, que aporte mayores prestaciones, y supere las desventajas que se citan a continuación:

a) Definición previa.

La gran mayoría de los lenguajes de programación tradicionales exigen que los índices, y por lo tanto las dimensiones del array, se incluyan en la parte declarativa del programa, no pudiéndose crear de forma dinámica en tiempo de ejecución, en base a datos externos aportados al programa durante la ejecución del mismo.

Ciertos lenguajes permiten esto último, pero debiendo quedar definidos los índices antes de la introducción de los elementos, y con la limitación de que una vez fijada la longitud del array, ésta permanece inalterable durante el resto de la ejecución del programa, sin posibilidad de redefinirla.

En muchas situaciones, es difícil poder determinar de antemano, antes de comenzar a incorporar elementos, las dimensiones exactas que va a necesitar el array para almacenar la información. Esto ocurre, sobre todo, cuando se va a cargar sobre el mismo el contenido de otra estructura (soportada en memoria primaria o secundaria) sometida a actualizaciones continuas (altas y bajas), y por tanto, sin tamaño prefijado.

Esta naturaleza estática y rígida de la estructura obliga a la hora de definir el array, a optar por una de las siguientes opciones:

- Intentar ajustar al máximo las dimensiones del array a lo que se estima pueden resultar en la práctica. Esto suele generar situaciones de desbordamiento u overflow que imposibilitan la utilización de la estructura e implican, por lo tanto, la finalización del programa que hace uso de ella.

- Definir unos límites máximos lo suficientemente amplios. Por lo general, con estos límites se desperdicia memoria, debido al elevado número de posiciones que suelen quedar sin ocupar. Además, las magnitudes de estos límites máximos están limitadas por:

    - El lenguaje de programación.
    - El compilador utilizado.
    - El espacio disponible en memoria RAM.

b) Series de elementos ordinales consecutivos.

El segundo problema viene dado por la disposición intrínseca de la estructura en posiciones consecutivas de memoria, lo cual garantiza el inmediato acceso a un elemento concreto de la misma, en base a los dos índices que lo referencian.

Para un array bidimensional de M filas y N columnas de elementos que ocupen B bytes cada uno, será necesario un espacio de M * N * B posiciones consecutivas de memoria. El sistema genera la matriz a partir de una determinada posición de memoria llamada dirección base, utilizando uno de los dos tipos de disposición posibles: en filas o en columnas

La dirección de un determinado elemento de una matriz A, referenciado por los índices [I , J], utilizando una disposición de filas, vendrá dada por la siguiente fórmula [COL78]:

Dirección A [I , J] = Dirección Base + B * [N * (I - 1) + (J - 1)]

y si la disposición es en columnas:

Dirección A [I , J] = Dirección Base + B * [M * (J - 1) + (I - 1)]

Por esto se exige que los conjuntos de índices sean series de elementos ordinales consecutivos. Incluso en muchos lenguajes dichas series únicamente pueden ser de números enteros consecutivos, siendo además necesario en muchos de ellos que el límite inferior sea 0 o 1.

Otros lenguajes como el Pascal dan más facilidades a la hora de definir los conjuntos de índices; permiten no sólo series de números enteros con límite inferior negativo, sino que también admiten como índices datos de tipo carácter, lógico e incluso series de datos enumerados definidos por el propio programador.

Ahora bien, si los elementos a almacenar están referenciados por una serie de valores no consecutivos no será posible declarar el array, salvo que:

- Sea posible reducir la serie de índices a una serie de valores consecutivos, en base a alguna relación matemática que cumplan los datos, casuística que no se da frecuentemente.

  Por ejemplo: la serie 1000, 2000, 3000, 4000, ... sería reducible a la serie 1, 2, 3, 4, ... en base a la división por 1.000 de cada elemento.

  La serie 10, 100, 1000, 10000, ... sería reducible también a la misma serie anterior 1, 2, 3, 4, ... en base al logaritmo en base 10 de cada elemento.

- Se construya el conjunto de índices con toda la serie de elementos comprendida entre el elemento menor (límite inferior) y el mayor (límite superior).

  Esta solución evidentemente implica reservar posiciones en memoria que no van a ser utilizadas nunca. En el caso de un array lineal, la longitud del mismo es igual a la diferencia entre el límite superior y el inferior del conjunto de índices, más 1. El desperdicio de espacio será tanto mayor cuanta más diferencia exista entre la longitud y el número de posiciones ocupadas por elementos, y cuanto mayor sea el tamaño del tipo base.

  Por ejemplo: si se desean almacenar 4 elementos de B bytes, referenciados por la serie 1, 103, 1.097 y 33.045, será necesario declarar 33.045 posiciones (todos los números enteros comprendidos entre el menor 1 y el mayor 33.045), lo cual supone desperdiciar 33.041 * B bytes.

  Si el problema se traslada a un array bidimensional, que es el caso que nos ocupa, la infrautilización del array se dispara.

c) Indices calculados fuera de rango.

El hecho de que los índices de arrays deban ser de un tipo de datos definido (escalar), tiene una consecuencia fundamental: los índices pueden ser calculados. En lugar de una constante, puede utilizarse una variable como índice o una expresión de índice general; esta expresión será evaluada y el resultado identificará el elemento seleccionado. Evidentemente esta propiedad proporcionará generalidad al programa.

Ahora bien, esta generalidad no sólo ofrece un recurso de programación más significativo y poderoso, sino que al mismo tiempo origina uno de los errores de programación más frecuentes: el valor resultante puede estar fuera del intervalo especificado como ámbito de los índices del array [WIR76] [WIR87].

Este acceso equivocado a un componente del array no existente, suele originar la interrupción anormal de la ejecución del programa, o la generación de resultados imprevistos.

En esta Tesis Doctoral se define un nuevo tipo abstracto de datos llamado **Array Bidimensional Dinámico** (en adelante ***ABD***), el cual conserva las prestaciones del array bidimensional tradicional, aporta otras nuevas, y elimina los inconvenientes citados anteriormente. Además de su definición se describen e implementan las operaciones básicas para su manejo.

Su naturaleza será *dinámica*, a diferencia del carácter estático que presentan los arrays clásicos. Esto significa que puede cambiar durante la ejecución del programa. En general, los cambios se pueden dividir en dos categorías: cambios de contenido y cambios de organizacion. Un cambio de contenido es cualquier cambio de los valores almacenados en los elementos de la estructura de datos. Un cambio en la organización es cualquier cambio relativo al tamaño o configuración de una estructura de datos dinámica. Los cambios de contenido (o al menos la posibilidad de cambio) son comunes a todas las variables, estáticas y dinámicas. Es la capacidad de una estructura dinámica de datos de cambiar su organización lo que la distingue de una estructura de datos estática [KIN92].

En lenguajes de programación avanzados es un hecho común el hacer posible la manipulación explícita de referencias de datos además de los propios datos. Esto implica que debe existir una distinción notacional clara entre los datos y la referencia de éstos y que en consecuencia, deben introducirse tipos de datos cuyos valores sean apuntadores (referencias) de otros datos [WIR76] [WIR87].

En base a estos apuntadores se implementará la estructura para conseguir su naturaleza dinámica que le permita establecer y reajustar su configuración en tiempo de ejecución en función de las actualizaciones de sus componentes.

2. IMPLEMENTACIÓN DEL ABD MEDIANTE LISTAS ENLAZADAS SIMPLES Y DOBLES

2.1. LISTAS ENLAZADAS. DEFINICIÓN Y TIPOS

2.1.1. Listas enlazadas simples

Un **lista enlazada simple o lista unidireccional** está constituida por una colección lineal de elementos llamados nodos, donde el orden de los mismos se establece mediante punteros [DALE86].

Cada nodo se divide en dos partes: una que contiene la información asociada al elemento, y otra llamada campo de enlace o puntero, que contiene la dirección del siguiente nodo de la lista.

El último nodo de la lista tendrá en su campo enlace el valor nulo, que es una dirección no válida, indicativa de que no existen más nodos en la estructura.

Además será necesario disponer de una variable puntero adicional que contenga la dirección del primer elemento de la lista. Cuando la lista esté vacía dicha variable contendrá el valor nulo.

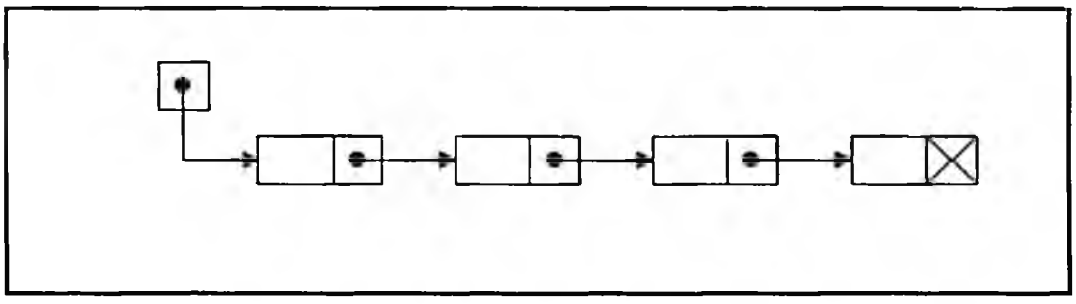


Fig. 2.1.
Lista Enlazada Simple

La implementación de una secuencia lineal de elementos en base a listas enlazadas presenta frente a los arrays las ventajas siguientes [LIPS87] :

a) En los arrays el orden lógico de los elementos viene predeterminado por el orden fisico (el primer elemento es el que ocupa la primera posición del array, el segundo la segunda posición, etc.).

Por el contrario, en las listas enlazadas existe independencia entre el orden lógico y el orden fisico. El orden lógico entre los elementos viene dado por el campo enlace que apunta desde cada elemento al siguiente y no por su ubicación física.

b) Consecuencia de lo anterior, los procesos de inserción y borrado de elementos en listas enlazadas serán mucho más simples que en arrays.

En estos últimos, cuando se quiere insertar un elemento en una determinada posición, es necesario desplazar hacia la derecha todos los elementos situados a la derecha de dicha posición, con el fin de abrir hueco al nuevo elemento.

De la misma forma, cuando se procede a borrar un elemento en una determinada posición, es necesario desplazar hacia la izquierda todos los elementos situados a la derecha de dicha posición con el fin de cubrir el hueco que deja el elemento eliminado.

Esto se debe a que en los arrays, el orden lógico de los elementos depende de la situación fisica de los mismos.

Por el contrario, para insertar un nodo N entre otros dos nodos N1 y N2, en listas enlazadas, bastará con asignar al campo enlace del nodo N la dirección del nodo N2, y asignar al campo enlace del nodo N1 la dirección del nodo N.

Asimismo, para eliminar un nodo N situado entre otros dos N1 y N2, bastará con asignar al campo enlace del nodo N1 la dirección del nodo N2, y destruir el nodo N.

Tanto en inserciones como en borrados en listas enlazadas simples se habrá de tener en cuenta la excepción de que el elemento a insertar o borrar sea el primero de la lista, en cuyo caso no existe nodo predecesor del que modificar su campo enlace; en esta situación habrá que modificar la variable de comienzo de lista.

Existe gran variedad de implementaciones de listas enlazadas, orientadas a facilitar los procesos aplicados sobre las mismas. Entre ellas destacan las que se describen a continuación [KNU86] [TEN86].

2.1.2. Listas con inserción al final

Para insertar un elemento al final de una estructura de lista enlazada simple, es necesario enganchar el nuevo nodo tras el que ocupa la última posición, para lo que previamente hay que conocer la dirección de éste con el fin de asignarle en su campo de enlace la dirección del nuevo nodo.

Para obtener la dirección de ese último nodo es preciso realizar un recorrido completo de la lista desde el comienzo hasta alcanzar el nodo cuyo campo enlace tenga el valor nulo.

Evidentemente este proceso no resulta nada eficiente, siendo tanto peor cuanto mayor es la lista.

Para realizar estas operaciones de forma óptima, será mejor definir otra variable puntero adicional que contenga la dirección del último elemento de la lista. De esta forma se tendrá acceso directo al último nodo de la lista, sin necesidad de realizar un recorrido previo para obtener su dirección.

Este tipo de implementación obliga a redefinir las operaciones de inserción y borrado, puesto que además de controlar la ya citada excepción de que el elemento a insertar o borrar sea el primero de la lista, es necesario controlar ahora si va a ser también el último (en cuyo caso habrá que actualizar la variable que lo apunta).

2.1.3. Listas circulares

Otra forma de resolver el problema de acceso al último elemento es convertir la lista en circular; es decir, el último elemento no tiene el valor nulo en su campo enlace al elemento siguiente, sino que apunta al primer elemento de la lista, cerrando de esta forma el trayecto de punteros de lista entre sí. Por lo tanto, no existirá ningún nodo que tenga el valor nulo en su campo enlace.

El recorrido completo de la lista podrá comenzar en cualquier nodo, finalizando en el anterior, puesto que los nodos están interconectados en anillo. Por esta razón, la variable puntero externa podría estar apuntando en principio a cualquier nodo.

De todas formas, será necesario diferenciar el primer elemento en orden lógico de la lista. Por ello, la variable puntero externa podría apuntar hacia dicho nodo, pero tendríamos el inconveniente anteriormente citado, de que para insertar un nodo al final de la lista, se necesitaría recorrerla al completo para obtener la dirección del último nodo.

Por esta razón, en listas circulares, es preferible asignar a la variable puntero externa la dirección del último nodo de la lista, ya que de esta forma se tiene acceso directo al último y al primer nodo (la dirección de este último se encuentra en el campo enlace del último).

Este tipo de listas obligan a reconsiderar no sólo los algoritmos de inserción y borrado, sino también los que necesiten realizar un recorrido de la lista, ya que ahora éste no finalizará tras procesar el nodo que tiene en su campo enlace el valor nulo (puesto que no existe ninguno), sino tras procesar el nodo apuntado por la variable externa.

2.1.4. Listas con cabecera

Este tipo de listas incluyen al comienzo de las mismas un nodo cabecera que no contiene información como el resto de nodos. Ese elemento existe físicamente, pero no se manipula ni se visualiza en los procesos de acceso a la

lista. En dicho nodo se puede almacenar alguna información especial relativa a la lista, tal como el número de nodos, la fecha de última actualización, etc.

La variable puntero de comienzo de lista apuntará siempre a dicho nodo cabecera. La lista con cabecera vacía será aquella que contenga únicamente dicho nodo cabecera.

La ventaja que presenta este tipo de listas es que simplifican los procesos de inserción y borrado, puesto que todos los nodos tendrán predecesor (incluido el primero, cuyo antecesor es el nodo cabecera), por lo que no existirá ninguna excepción cuando el nodo a insertar o borrar sea el primero.

2.1.5. Listas doblemente enlazadas

En estas listas cada nodo consta de tres partes:

- La información asociada al elemento.
- La dirección del siguiente nodo de la lista.
- La dirección del nodo anterior de la lista.

Desde cada nodo se tiene acceso directo, no sólo al nodo siguiente como ocurre en las listas simples, sino también al nodo que le precede. El último nodo de la lista tendrá el valor nulo en el campo que contiene la dirección del nodo sucesor, y el primer nodo de la lista tendrá el valor nulo en el campo que contiene la dirección del nodo antecesor.

Para eliminar un nodo, proporcionada su dirección, es necesario conocer la dirección del nodo precedente, para asignarle en su campo de enlace al siguiente, la dirección del nodo sucesor al que va a ser eliminado.

Para obtener la dirección del predecesor en listas enlazadas simples, es necesario realizar un recorrido de la lista hasta alcanzar dicho nodo, que será aquel cuyo campo enlace contenga la dirección del nodo a borrar. Evidentemente éste no es un proceso eficaz.

Por el contrario, en listas enlazadas dobles, conocida la dirección del nodo a borrar, se obtiene directamente la dirección del nodo predecesor, la cual estará almacenada en uno de sus campos puntero. Por lo tanto, el borrado de un nodo por dirección será inmediato.

Al contener en cada nodo la dirección de su predecesor, será posible realizar un recorrido de la lista en orden inverso, desde el último hasta el primero. Para comenzar este recorrido habrá que situarse en el último nodo, por lo que se deberá tener acceso directo a él. Por esta razón, suele ser muy frecuente el uso de listas enlazadas dobles con inserción por el final, es decir con un puntero al primer elemento de la lista y otro puntero al último.

Los procedimientos de inserción y borrado en listas enlazadas dobles son algo más complejos que en listas simples, puesto que a la hora de insertar y borrar no sólo habrá que modificar el campo enlace al siguiente del nodo predecesor, como ocurre en las simples, sino que también será necesario modificar el campo enlace al anterior del nodo sucesor.

2.1.6. Listas ordenadas

Son aquellas listas en las cuales los nodos se encuentran ordenados en creciente o decreciente en base a sus campos de información.

La inserción de un nodo debe realizarse siempre en el lugar adecuado, de forma que se conserve el criterio de ordenación tras la incorporación del nuevo nodo.

Las listas enlazadas ordenadas presentan ventaja en los procesos de búsqueda de un elemento. En listas no ordenadas una búsqueda no fructífera obliga a recorrer al completo la lista. En el caso de listas ordenadas en creciente, esa búsqueda puede finalizar en el momento de alcanzar un elemento superior al buscado, indicando esto que el elemento no existe en la lista.

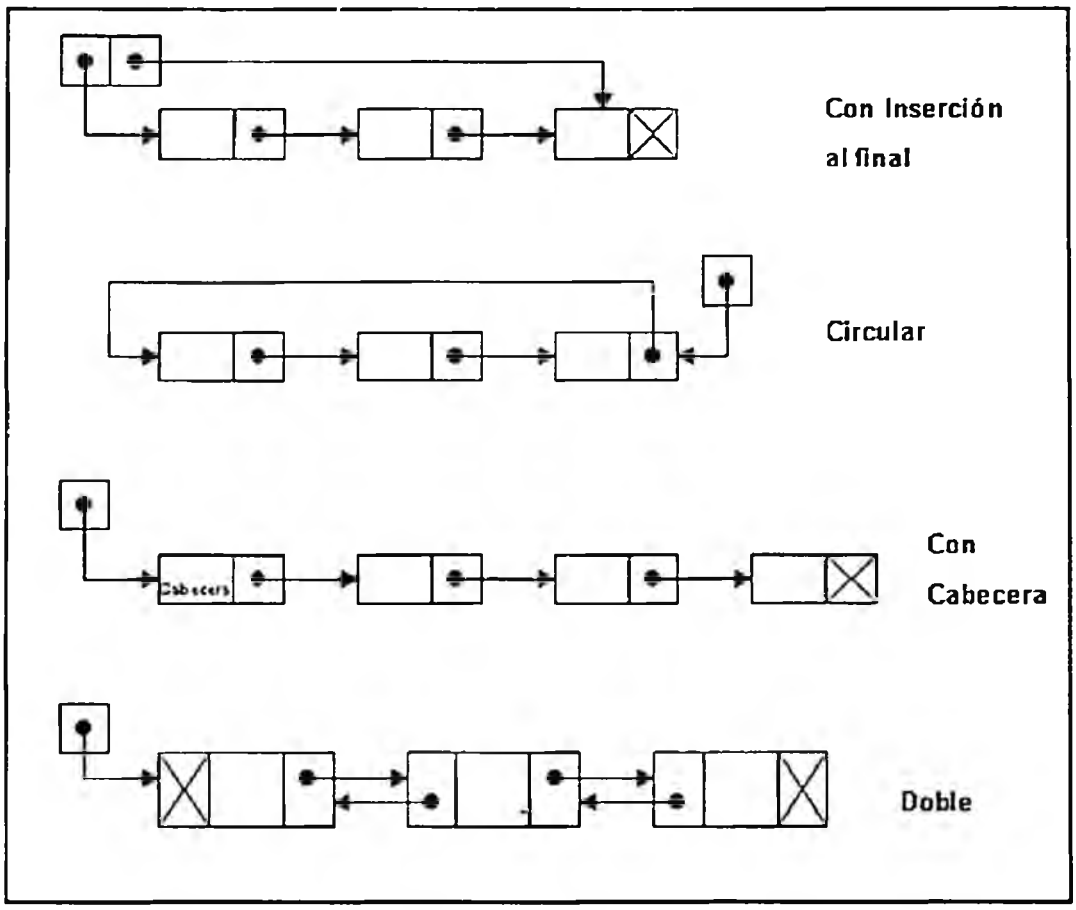


Fig. 2.2.
Otros tipos de Listas

Es frecuente encontrar listas enlazadas que son combinación de las aquí expuestas: dobles con inserción al final, circulares con cabecera, circulares doblemente enlazadas, de cualquier tipo ordenada, etc.

2.2. ESTRUCTURA DEL ABD EN BASE A LISTAS

Para implementar la estructura propuesta de **Array Bidimensional Dinámico** se pueden emplear las siguientes disposiciones de listas:

a) Dos listas enlazadas simples ordenadas para almacenar los índices (una para filas y otra para columnas).

b) Una lista lineal enlazada doble con inserción al final ordenada para cada fila y columna, entrelazadas entre sí, para almacenar los elementos.

a) Listas de Índices

Cada lista enlazada de índices contendrá un nodo por cada índice. El nodo estará formado por:

- Un campo de información (*InfoIndice*) que contendrá el valor del índice que representa.

- Un campo puntero (*SigIndice*) que contendrá la dirección del nodo siguiente, es decir, el siguiente índice del conjunto de índices.

- Un campo puntero (*Primero*) que contendrá la dirección del primer elemento de la lista de elementos de la fila o columna que representa.

- Un campo puntero (*Ultimo*) que contendrá la dirección del último elemento de la lista de elementos de la fila o columna que representa.

El primer nodo de la lista enlazada de índices de fila estará apuntado por una variable inicio de tipo puntero que llamaremos *ListaIF*. Del mismo modo, el primer nodo de la lista enlazada de índices de columna estará apuntado por una variable inicio de tipo puntero denominada *ListaIC*.

Asimismo el campo *SigIndice* del último índice de cada lista tendrá el valor nulo, indicando que es el último del conjunto de índices.

No se podrá dar el caso de que algún nodo de la lista de índices tenga el campo *Primero* o *Ultimo* con el valor nulo, puesto que esto significaría que no existe ningún elemento en la fila o columna que representa, por lo que no tendría sentido la existencia de tal índice.

El razonamiento es evidente: una de las características fundamentales de la estructura que se propone es el ajuste de su dimensionamiento de forma dinámica en tiempo de ejecución a medida que se añadan o eliminen elementos. Por tanto, cuando sea necesario incluir un elemento referenciado por un par de índices y alguno de ellos no esté creado, previamente se añadirá dicho índice a la lista enlazada de índices correspondiente.

De la misma forma, cuando se elimine un elemento de la estructura que era el único de su fila y/o columna, se eliminará el índice correspondiente a la fila y/o columna.

Así, antes de comenzar a incorporar elementos a la estructura, no existirán las listas de índices, puesto que no es necesario predeterminar los conjuntos de índices correspondientes. Por tanto las variables *ListaIF* y *ListaIC* tendrán el valor nulo.

Cada lista de índices estará ordenada en base al campo de información con el objeto de facilitar la búsqueda del índice. La ventaja que presenta la búsqueda de un elemento en una estructura de lista enlazada ordenada frente a una que no lo está es que, en caso de no existir el elemento buscado, en el primer caso el algoritmo puede detenerse cuando se alcanza un elemento de clave superior, mientras que en caso de lista desordenada sería necesario llegar al final de la estructura.

b) Listas de Elementos

Existirá una lista enlazada doble con inserción al final por cada fila y cada columna de la estructura, es decir, si existen M índices en filas y N índices en columnas, en total existirán M+N listas entrelazadas a modo de malla. Cada nodo, representante de un elemento de la matriz, formará parte simultáneamente de una lista de filas y de una lista de columnas (las correspondientes a sus índices).

Por tanto cada nodo estará formado, en principio, por:

- Uno o varios campos de información (*InfoElemento*) propios del elemento. A nivel de contenido, cada elemento estará formado de un dato simple (si es necesario almacenar una única información), o por un dato compuesto (por ej. un registro si es necesario almacenar más de una información por elemento).

- Un campo puntero (*SigFila*) que contendrá la dirección del siguiente elemento de su fila.

- Un campo puntero (*SigColumna*) que contendrá la dirección del siguiente elemento de su columna.

- Un campo puntero (*AntFila*) que contendrá la dirección del elemento anterior de su fila.

- Un campo puntero (*AntColumna*) que contendrá la dirección del elemento anterior de su columna.

- Un campo puntero (*IndFila*) que apuntará al nodo índice de su fila.

- Un campo puntero (*IndColumna*) que apuntará al nodo índice de su columna.

El primer elemento de cada fila y de cada columna estará apuntado por el campo *Primero* del nodo representante del índice correspondiente de las listas de índices. Además el primer elemento de cada fila y de cada columna tendrá en el campo *AntFila* y en el campo *AntColumna* respectivamente el valor nulo.

El último elemento de cada fila y de cada columna tendrá en el campo *SigFila* y en el campo *SigColumna* respectivamente el valor nulo. Dicho elemento último estará apuntado por el campo *Ultimo* del nodo índice correspondiente.

Las listas de elementos se han implementado como dobles, almacenando la dirección del predecesor de cada nodo, puesto que como veremos más adelante, será interesante de cara a determinados procesos. Además son listas con inserción

al final, con apuntadores a su primer y a su último elemento, para facilitar el recorrido de las filas y de las columnas en sentido directo y en sentido inverso.

Las listas de filas estarán ordenadas por los índices de columna de sus elementos, y las listas de columnas por los índices de fila de sus elementos. Esta ordenación es una consecuencia lógica de la disposición matricial.

Los campos puntero *IndFila* y *IndColumna* se han implementado sobre todo de cara al acceso de un determinado elemento en base a sus dos índices. Estos campos punteros podrían ser sustituidos directamente por dos campos de información de los tipos índices de filas y columnas, almacenando en ellos los valores de los índices de fila y columnas respectivamente de cada elemento.

Ahora bien, se opta por el almacenamiento de punteros debido a dos razones fundamentales:

a) Si el tipo de dato base utilizado en índices no es excesivamente simple, el tipo puntero ocupará menos espacio en memoria.

b) Podría ser interesante en algún caso reemplazar el valor de un índice por otro. Ante esta tesitura, almacenando en el elemento los valores de sus índices de fila y columna, además de modificar el nodo índice en la lista de índices correspondiente sustituyendo el valor antiguo por el nuevo, sería necesario modificar todos aquellos elementos referenciados por el citado índice.

   Por el contrario, si en cada elemento se almacenan direcciones a sus respectivos nodos índices en filas y columnas, no será necesario modificar para nada los elementos por el hecho de cambiar el valor del índice.

Por tanto para una adecuada gestión de la estructura es necesario, por cada elemento que se incluya en el array bidimensional dinámico, almacenar adicionalmente a su propia información, seis elementos de tipo puntero.

De entrada esto puede parecer excesivo, pero su efecto negativo de necesidad de memoria adicional, quedará minimizado, frente a un array bidimensional convencional :

a) *Cuanto mayor sea la cantidad de información a almacenar por cada elemento.*

Los seis punteros adicionales supondrán un peso menor en la cantidad total de espacio ocupado por el elemento cuanto mayor sea su información propia.

b) *Cuanto mayores sean las dimensiones del array y más bajo su grado de ocupación.*

En los arrays tradicionales, debido a la forma de acceso, tal y como se ha visto anteriormente, es necesario establecer posiciones para todos los posibles elementos referenciados por todos los índices, aunque parte de ellos no vayan a existir nunca.

Se reserva espacio en memoria para los posibles M*N elementos, se vayan a introducir posteriormente o no. El desperdicio dependerá del tamaño del array y de su grado de ocupación.

En el ***ABD*** existe un nulo desperdicio de espacio en las posiciones no utilizadas, ya que no existe físicamente el nodo correspondiente al elemento que no existe.

Asimismo únicamente estarán presentes los índices utilizados para referenciar los elementos incorporados a la estructura. Como ya se ha comentado no es necesario que los conjuntos de índices sean series de datos consecutivos.

c) *Por las ventajas que ofrece el **ABD**:*

- Mayor flexibilidad al no tener que conocer de entrada ni la cantidad ni los posibles valores de los índices, ni tener que ser éstos series consecutivas.

- Creación y destrucción dinámica de Indices y Elementos.

- Ahorro del espacio de las posiciones no utilizadas.

- Control de errores ante índices fuera de rango.

- Mayor eficacia en determinadas operaciones.

Se adjunta la representación de esta implementación mediante listas de un *ABD* que consta de los 8 elementos siguientes:

[3 , 9] = 22 [3 , 49] = 12 [20 , 25] = 6

[25 , 9] = 11 [25 , 95] = 12 [80 , 25] = 12

[86 , 49] = 77 [86 , 95] = 36

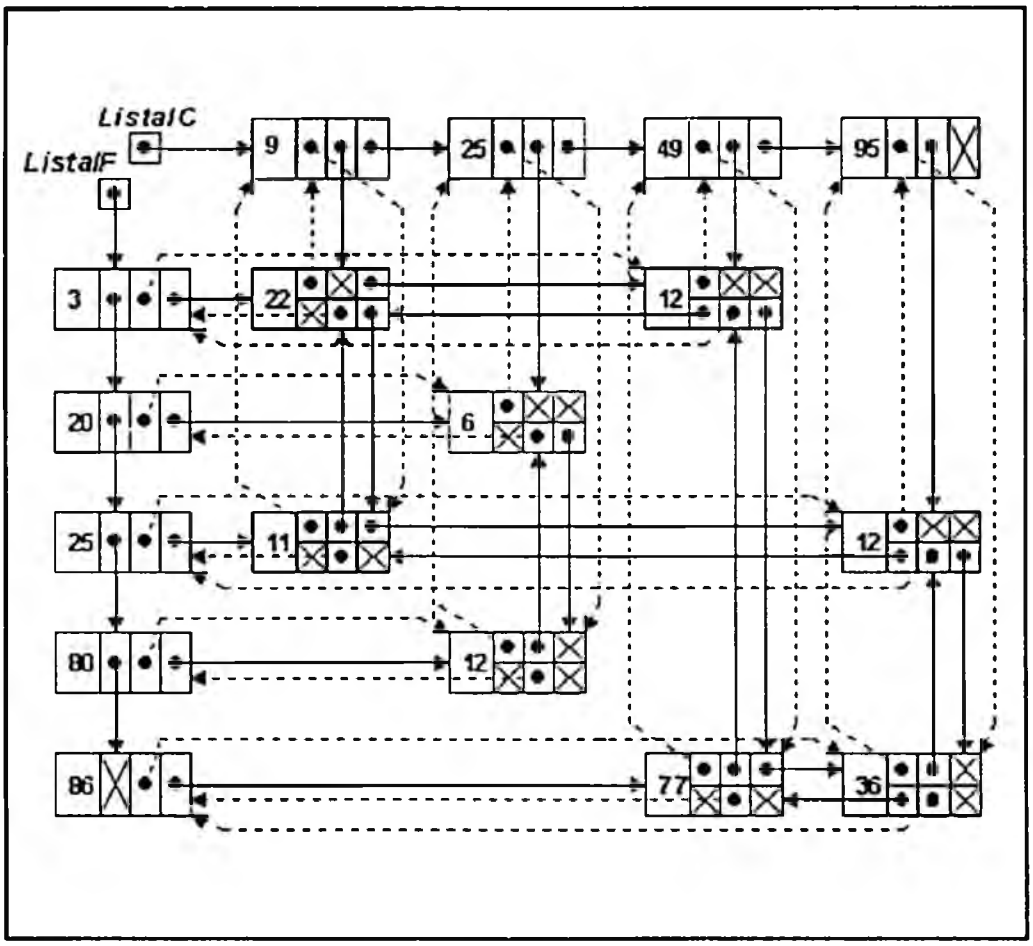


Fig. 2.3.

Array Bidimensional Dinámico implementado mediante Listas

Este tipo de estructura mallada de listas que se propone para representar los elementos del ***ABD*** se conoce con el nombre de ***Listas Ortogonales***, ***Multienlazadas*** o ***Multilistas*** [STAN80] [REIN83] [WOOD93].

Aparecen en la literatura frecuentemente asociadas a lo que se denominan *Sparse Matrices*, matrices numéricas en las que la mayoría de sus componentes son cero [HORO82] [GOT78].

Ejemplos comunes de matrices con un significativo número de elementos a cero, y que son consideradas *sparse* son:

La matriz *diagonal*, matriz cuadrada con ceros en todas las posiciones excepto en la diagonal principal, y que puede ser considerada como un vector lineal. En el caso de una matriz diagonal de orden n, el ratio entre el número de elementos no-cero y el número total de elementos de la matriz es $n / n^2 = 1 / n$.

La matriz *tridiagonal*, matriz cuadrada con elementos no-cero en la diagonal principal y en sus adyacentes inferior y superior, y ceros en todas las demás posiciones. El ratio de ocupación en una matriz de orden n es $(3n - 2) / n^2$.

En ambos casos el ratio es O(1/n), reduciéndose de forma significativa a medida que aumenta el valor de n.

La matriz *triangular* superior o inferior, según sean cero todos los elementos situados por debajo o por encima de la diagonal principal, si bien posee bastantes elementos a cero, no suele ser considerada como sparse. Su ratio de ocupación es $[n * (n + 1) / 2] / n^2 = (1 / 2 + 1 / 2n)$. Orden: O(1).

Cada uno de estos tipos de matrices puede ser considerado como un caso especial de una matriz de bloques más general. Tewarson [TEW73] propone 10 formas canónicas de matrices de bloques a las que pueden ser reducidas las matrices con un significativo número de ceros. Esta reducción supone la ventaja de, además del ahorro de no tener que almacenar los componentes cero, aumentar la velocidad y precisión en determinadas operaciones.

Si bien es razonable y lógica la utilización de las listas ortogonales para la implementación de sparse matrices, con el propósito de economizar espacio, al no almacenar los elementos cero, dicho uso se antoja precario, teniendo en cuenta el rendimiento que se puede obtener de esta estructura basada en apuntadores.

Dedicar esta representación a almacenar únicamente matrices numéricas poco densas, con el objetivo prioritario de reducir la ocupación en memoria de la matriz, es restringir las posibilidades que ofrece la estructura.

Si bien el ***ABD*** propuesto en esta Tesis toma, en una primera versión, como base para su implementación, las listas ortogonales, aporta ventajas adicionales a la economía de espacio, tales como su condición dinámica en el mantenimiento de índices y elementos, la flexibilidad en la definición de los conjuntos de índices y el control de errores.

Asimismo, su aplicación no se circunscribe a las matrices numéricas, sino que se considera estructura de almacenamiento general de cualquier tipo de datos, donde cada elemento se identifica y recupera en base a dos índices.

Además, el ***ABD*** debe ser entendido como un tipo abstracto de datos (*TAD*), puesto que a la estructura de implementación comentada, incorpora las operaciones básicas de manejo de la misma. La definición de estas operaciones constituye el interfaz del TAD que permite a los programas de usuario hacer uso de la estructura. Su implementación interna, en esta primera versión, en base a listas enlazadas, permanece oculta, y cualquier cambio en la misma no deberá afectar a los programas que lo utilizan.

Este es uno de los principios básicos del TAD, diferenciación entre su visión interna y su visión externa. Esta propiedad, quedará corroborada en la segunda parte de la Tesis, en la cual se sustituye la implementación en base a listas enlazadas, por una implementación en base a árboles binarios, y sin embargo, el interfaz de manejo del TAD no se verá alterado.

2.3. OPERACIONES DE MANIPULACIÓN DEL ABD

Una vez definida la estructura se van a implementar y describir de forma detallada las diferentes operaciones básicas que permiten su manipulación.

Estas operaciones las podemos dividir en dos tipos diferenciados:

a) Las comunes a los arrays bidimensionales estáticos.

- ⇨ Recorrido por filas.
- ⇨ Recorrido por columnas.
- ⇨ Acceso a un elemento referenciado por sus índices.
- ⇨ Búsqueda de un elemento (o todos) en base a un campo de información.
- ⇨ Inserción de un elemento.
- ⇨ Borrado de un elemento referenciado por sus índices.
- ⇨ Borrado de un elemento (o todos) en base a un campo de información.
- ⇨ Modificación de un elemento referenciado por sus índices.
- ⇨ Modificación de un elemento (o todos) en base a un campo de información.

b) Las particulares e intrínsecas del ABD, en base a su condición dinámica.

- ⇨ Eliminación de una fila.
- ⇨ Eliminación de una columna.
- ⇨ Modificación de un índice de fila.
- ⇨ Modificación de un índice de columna.

Notación Seudocódigo utilizada en los algoritmos

*** Asignación.**

Variable ← *Dato*

Carga la variable especificada a la izquierda con el dato indicado a la derecha.

*** Estructuras condicionales.**

Si *Condición* **entonces** { Conjunto de Sentencias } **fin-si**	Si la condición es cierta ejecuta el conjunto de sentencias; en caso contrario no hace nada.
Si *Condición* **entonces** { Conjunto 1 de Sentencias } **si no** { Conjunto 2 de Sentencias } **fin-si**	Si la condición es cierta ejecuta el conjunto de sentencias 1; en caso contrario ejecuta el conjunto de sentencias 2.

*** Estructura repetitiva.**

Mientras *Condición* **hacer**
{ Conjunto de Sentencias }
fin-mientras

Repetitiva de 0 - N. Evalúa la condición y si es cierta ejecuta el conjunto de sentencias del ámbito del ciclo. Este proceso se repite hasta que la condición toma valor falso.

*** Operadores.**

Relacionales

=	Igual	< >	Distinto	>	Mayor
<	Menor	>=	Mayor o igual	<=	Menor o igual

Lógicos

o	O Lógico.	Cierto cuando algún operando es cierto. Falso cuando todos los operandos son falsos.
y	Y Lógico.	Cierto cuando todos los operandos son ciertos. Falso cuando algún operando es falso.
no	Negación.	Devuelve el estado contrario del operando.

*** Arrays.**

A [I , J]	Elemento del array bidimensional A referenciado por los índices I , J.

*** Registros.**

Se utilizan en el texto para representar los nodos. Cada campo se referencia como sigue:

Nombre Registro.Nombre Campo

*** Punteros o Apuntadores.**

Variables que contienen la dirección de otras variables (en este caso de nodos).
Valor Nulo es el contenido que indica que la variable no apunta a nada.
Las variables referenciadas por punteros se expresan con el nombre de la variable puntero y el carácter ^ : *Variable Puntero*^

*** Procedimientos.**

Se definen en la cabecera con el nombre del procedimiento, y entre paréntesis los diferentes parámetros separados por punto y coma. Cada parámetro se especifica si es de entrada (e), de salida (s), o de entrada/salida (es).
Tras la cabecera se especifican las sentencias ejecutables de que consta.
Se llaman desde otros procedimientos o programas indicando su nombre y la serie de parámetros entre paréntesis y separados por comas.

*** Pilas.**

Se utilizan como estructuras auxiliares en los recorridos de árboles implementados de forma iterativa.
Los procedimientos que se usan en los algoritmos son:

PilaVacia (*Pila*)	Inicializa la pila.
Push (*Pila* , *X*)	Inserta el elemento X en la cima de la pila.
Pop (*Pila* , *X*)	Extrae el elemento cima de pila y asigna su valor a X.

*** Comentarios.**

(* Texto de comentario *)

2.3.1. Recorrido por filas

La operación de recorrido de una estructura consiste en acceder una única vez a cada componente de la misma, aunque se puedan procesar uno o mas datos de cada elemento.

En un array bidimensional esta operación puede ser realizada de dos formas: por filas o por columnas.

Analicemos el primer caso, que es el que nos ocupa en este apartado. El recorrido por filas consistirá en tratar de forma secuencial los elementos de la primera fila, a continuación los de la segunda, ..., y por último los componentes de la última fila.

En un array bidimensional tradicional estático, el proceso estaría basado en dos ciclos anidados: el externo haría tantas iteraciones como filas tuviese la matriz y el interno haría tantas iteraciones como columnas existiesen en la matriz.

Es decir, se procesarían las filas secuencialmente desde la primera hasta la última, y en cada una de ellas se tratarían todas las columnas. Cada posición correspondiente a una fila y a una columna podrá tener almacenado un elemento o bien estar vacía, en cuyo caso debería existir algún dato en esa posición que indicara tal circunstancia. (por ej. si en el array se almacenan números naturales, incluir el valor -1 en las posiciones vacías).

La operación de recorrido sólo deberá tratar las posiciones válidas, es decir, las que contienen un elemento, pero sin embargo se ve obligado a acceder a todas las posiciones de la matriz. Esto evidentemente encarece la operación de recorrido tanto más cuanto mayor sea el array y cuanto menor sea su grado de ocupación.

En la estructura propuesta el proceso de recorrido se haría de la forma siguiente:

El principio general es el mismo, y se puede implementar también en dos ciclos anidados: el externo haría tantas iteraciones como filas tuviese la matriz y el interno el número de iteraciones variaría por cada fila y sería igual al número de elementos de dicha fila.

Para llevar a cabo este proceso se partiría de la dirección contenida en la variable *ListaIF*, que apuntaría al primer nodo de la lista de índices de fila, es decir, al primer índice de filas. Este nodo tendrá en su campo *Primero* la dirección del primer elemento de la fila que representa. A partir de este nodo todos los demás de dicha fila estarán encadenados por su campo *SigFila*, de tal forma que los iremos procesando hasta llegar a uno que tenga en dicho campo *SigFila* el valor nulo o indicador de elemento último.

Una vez tratados todos los elementos de una fila, se debe pasar al siguiente índice de filas, es decir, se debe avanzar en la lista de índices al nodo siguiente al que acabamos de tratar en base a su campo *SigIndice*. Una vez situados en ese nuevo índice se debe repetir el proceso descrito anteriormente de procesar todos los elementos de esa fila. El recorrido finalizará después de tratar los nodos de la última fila, que será la representada por el nodo índice que tenga en su campo *SigIndice* el valor nulo.

Este recorrido tiene una clara ventaja respecto al anterior sobre array clásico: se accede únicamente a los elementos almacenados en la estructura, puesto que son los únicos que están presentes en la misma, ya que aquí no aparecen nunca posiciones vacías.

Cada acceso nos proporciona siempre un elemento a tratar, y existirán tantos accesos como elementos existan en la estructura. En el array tradicional existen tantos accesos como posiciones reservadas, es decir, como elementos potenciales pueden existir, siendo además el tratamiento más complejo puesto que obliga a evaluar si existe o no información válida en la posición accedida.

Por tanto los tiempos de respuesta del recorrido en la estructura dinámica propuesta serán inferiores a los del array estático clásico.

En el Anexo A.1. se desarrollan en pseudocódigo ambos recorridos.

2.3.2. Recorrido por columnas

El recorrido por columnas consistirá en tratar de forma secuencial los elementos de la primera columna, a continuación los de la segunda, ..., y por último los componentes de la última columna.

En un array bidimensional tradicional estático, el proceso estaría basado en dos ciclos anidados: el externo haría tantas iteraciones como columnas tuviese la matriz y el interno haría tantas iteraciones como filas existiesen en la matriz.

Es decir, se procesarían las columnas secuencialmente desde la primera hasta la última, y por cada una de ellas se tratarían todas las filas. Cada posición correspondiente a una fila y a una columna podrá tener almacenado un elemento o bien estar vacía, en cuyo caso, como ya se ha comentado anteriormente debería existir algún dato en esa posición que indicara tal circunstancia.

La operación de recorrido sólo deberá tratar las posiciones válidas, es decir, las que contienen un elemento, pero sin embargo se ve obligado a acceder a todas las posiciones de la matriz. Esto evidentemente encarece la operación de recorrido tanto más cuanto mayor sea el array y cuanto menor sea su grado de ocupación.

En el array dinámico el proceso de recorrido por columnas estará basado en dos ciclos anidados: el externo iterará tantas veces como columnas tenga la matriz y en el interno el número de iteraciones variará por cada columna y coincidirá con el número de elementos de dicha columna.

Para llevar a cabo este proceso se partiría de la dirección contenida en la variable *ListaIC*, que apuntaría al primer nodo de la lista de índices de columna, es decir al primer índice de columna. Este nodo tendrá en su campo *Primero* la dirección del primer elemento de la columna que representa. A partir de este nodo todos los demás de dicha columna estarán encadenados por su campo *SigColumna*, de tal forma que los iremos procesando hasta llegar a uno que tenga en dicho campo *SigColumna* el valor nulo o indicador de elemento último.

Una vez tratados todos los elementos de una columna, se debe pasar al siguiente índice de columnas, es decir, se debe avanzar en la lista de índices de columna al nodo siguiente al que acabamos de tratar, en base a su campo

SigIndice. Una vez situados en ese nuevo índice se debe repetir el proceso descrito anteriormente de procesar todos los elementos de esa columna. El recorrido finalizará después de tratar los nodos de la última columna, representada por el nodo índice que tenga en su campo *SigIndice* el valor nulo.

Las ventajas del array dinámico frente al array estático, en la operación de recorrido de columnas, son las mismas descritas anteriormente en el recorrido por filas.

En el array dinámico cada avance nos sitúa en un elemento a tratar de la estructura. Por el contrario, en el array clásico cada avance nos sitúa en una posición del mismo que puede estar ocupada por un elemento o no, y que por tanto será necesario examinar,

La diferencia estriba en que en el primer caso no existen posiciones vacías y en el segundo sí. Por tanto el número de accesos en el caso del array dinámico será igual al número de elementos incluidos en el mismo (NUM), mientras que en el caso del array clásico el número de accesos coincidirá con el de potenciales elementos a incluir en el mismo (M * N)

Evidentemente siempre NUM será menor o igual a M * N. Será igual en el caso de que el array esté lleno, es decir, contenga el máximo de elementos, y será menor en caso contrario, acrecentándose la diferencia entre ambos cuanto mayor sea y más vacía esté la estructura.

El tiempo de respuesta estará en relación directa con el número de accesos realizados. Por ello será más óptimo en el array dinámico que en el estático. no sólo por el menor número de accesos, sino también porque estos son más simples (no necesitan evaluación previa como ocurre en el estático).

Cuanto mayor sea el array y menor su porcentaje de ocupación, mayor sera la diferencia de tiempo de ejecución entre el array estático y el dinámico, ya que el recorrido sobre éste último es sensible al número de elementos de la estructura.

En el Anexo A.2. se desarrollan en pseudocódigo ambos recorridos.

2.3.3. Acceso a un elemento referenciado por sus índices

En un array bidimensional tradicional, el acceso a un elemento en base a sus índices [I , J] es inmediato debido a su implementación en memoria, dispuestos todos sus componentes de forma secuencial y consecutiva bien por filas o columnas.

El problema que plantea, es que si alguno de los índices se encuentra fuera de rango se producirá un error en ejecución, interrumpiéndose ésta de forma inmediata, o incluso, si no se detecta el error se obtendrán resultados imprevisibles.

En un array bidimensional dinámico el acceso en base a los índices es un proceso más complejo, si bien están controlados los errores por índices fuera de rango.

El acceso se puede realizar de dos formas:

A) Primer Método

En primer lugar habrá que realizar una búsqueda con el valor del índice de fila *(I)* en la lista ordenada de índices de fila.

Esta búsqueda comenzará a partir del primer nodo, que será el apuntado por la variable *ListaIF*. El avance se realizará en base al campo *SigIndice* que contiene para cada nodo índice la dirección del siguiente índice.

Finalizará cuando se alcance un nodo cuyo campo de información (*InfoIndice*) coincida con el valor del índice de fila *(I)* (búsqueda con éxito) o bien cuando se llegue al final de la lista o cuando se alcance un nodo cuyo campo de información sea mayor al valor del índice de fila (búsqueda fallida).

En el caso de que la búsqueda sea fallida el algoritmo devolverá un código de error indicando que el índice de fila se encuentra fuera de rango del conjunto de índices.

En el caso de que la búsqueda del índice de fila tenga éxito, se podría comenzar a buscar el elemento en la lista de elementos del índice en cuestión. Ahora bien, en el caso de que dicho elemento no existiera, no se podría determinar si existía o no el índice de columna.

En determinadas aplicaciones, puede resultar interesante e incluso necesario el proporcionar un código de error en el caso de que no exista el índice de columna, por lo cual, previamente a la búsqueda del elemento, se realizará la búsqueda del índice de columna.

Esta búsqueda comenzará a partir del primer nodo de la lista de índices de columna, que será el apuntado por la variable *ListaIC*. El avance se realizará en base al campo *SigIndice* que contiene para cada nodo índice la dirección del siguiente índice.

Finalizará cuando se alcance un nodo cuyo campo de información (*InfoIndice*) coincida con el valor del índice de columna *(J)* (búsqueda con éxito) o bien cuando se llegue al final de la lista o cuando se alcance un nodo cuyo campo de información sea mayor al valor del índice de columna (búsqueda fallida).

En el caso de que exista el índice de columna, se puede comenzar a buscar el elemento, en la lista de elementos referenciada por el índice de fila. Para ello se deberá comenzar en el primer nodo, el cual estará apuntado por el campo *Primero* del nodo índice.

Una vez situados en un nodo deberemos comprobar si éste es el elemento buscado. Para ello haremos uso del campo *IndColumna* del nodo, el cual contiene la dirección del nodo índice de la columna. Se pueden dar 3 casos:

a) Si el valor del campo de información del nodo índice de columna (*InfoIndice*) referenciado por *IndColumna* es menor que valor del índice de columna *(J)*, deberemos continuar la búsqueda pasando al siguiente nodo en la lista de elementos del índice de fila, realizando el avance por medio del campo *SigFila* del nodo, y repitiendo el tratamiento anterior.

b) Si el valor del campo de información del nodo índice de columna (*InfoIndice*) referenciado por *IndColumna* coincide con el valor del índice de columna *(J)*, quiere decir que la búsqueda ha sido exitosa y estamos situados en el nodo buscado, con lo cual se da por finalizado el algoritmo.

c) Si el valor del campo de información del nodo índice de columna (*InfoIndice*) referenciado por *IndColumna* es mayor que el valor del índice de columna *(J)*, la búsqueda se considerará fallida, ya que la lista de índices de columna está ordenada y por tanto el resto de índices de columna serán superiores al de referencia. Se dará por fallida la búsqueda también en el caso de llegar al final de la lista. En este tercer caso el algoritmo devolverá un código de error indicando que el elemento no existe.

En este modelo se ha recorrido la fila correspondiente al índice *(I)* comenzando por el primer elemento de dicha fila y continuando en orden directo, de izquierda a derecha, progresando de un nodo al siguiente por medio del campo *SigFila*.

Si por alguna razón se sospechara que el elemento referenciado se encuentra más próximo al final de la fila que al principio, se podría haber comenzado la búsqueda por el final de la fila y haber continuado en sentido inverso de derecha a izquierda, puesto que la lista es doblemente enlazada.

El recorrido inverso de la fila comenzaría en el nodo apuntado por el campo *Ultimo* del nodo índice y el avance de un elemento al anterior se haría en base al campo *AntFila* de cada elemento.

Análogamente se podría actuar en la localización de los índices. Los procesos de localización de los índices de fila *(I)* y de columna *(J)* se realizan en las listas de índices de fila y de columna comenzando desde *ListaIF* y *ListaIC*, respectivamente, y avanzando por medio del campo *SigIndice*.

De forma similar a lo comentado anteriormente, si se pudiera deducir que el índice buscado está más próximo del final de su lista de índices que del comienzo de la misma, sería más interesante comenzar la búsqueda desde el final de la lista, en sentido inverso, en dirección al comienzo de la misma.

Para ello sería necesario que las listas de índices estuviesen implementadas como listas dobles, es decir, que cada nodo tuviera un puntero adicional (*AntIndice*) conteniendo la dirección del índice anterior.

Pero además, para poder comenzar la búsqueda por el índice mayor sería necesario mantener una variable *FinalIF* (para filas) y *FinalIC* (para columnas) conteniendo la dirección del último nodo de la lista de índices de fila y de columna respectivamente, es decir, la dirección del índice superior de filas y columnas.

Estos elementos adicionales (*AntIndice, FinalIF, FinalIC*) no los vamos a implementar en la estructura, debido a que en muy raros casos sería posible determinar si el valor buscado está más próximo del final que del comienzo.

Si el conjunto de índices fuese una serie de valores consecutivos sí sería posible establecer fácilmente el extremo más adecuado para comenzar la búsqueda. Bastaría con calcular el elemento mitad sumando los límites inferior y superior del conjunto de índices y dividiendo entre 2.

Indice Mitad = (LIF + LSF) / 2

Si el índice buscado fuese menor que el índice mitad, estaría en la primera mitad de la lista de índices, por lo que lo más rápido sería comenzar la búsqueda desde el comienzo. Por el contrario, si el índice buscado fuese mayor que el índice mitad, estaría en la segunda mitad de la lista de índices, por lo que sería más adecuado comenzar la búsqueda desde el final de la lista de índices en orden inverso.

En la estructura propuesta de array bidimensional dinámico, una de las ventajas es precisamente que la serie de índices no tiene que estar formada por valores consecutivos, por lo cual no es aplicable el procedimiento anterior.

Una solución a este problema consistiría en implementar las listas como "listas de salto" (skip lists) [PUGH90] [SCH94]. En ellas, por término medio, la mitad de los nodos tienen un único puntero al nodo siguiente. La otra mitad tiene también un puntero al nodo que se encuentra en la lista dos posiciones por delante de él. La mitad de estos nodos tiene también un puntero al nodo situado

cuatro posiciones por delante de él. A su vez la mitad de estos nodos tiene un puntero al nodo situado ocho posiciones por delante, etc.

Evidentemente esta implementación casi convertiría las búsquedas en binarias, pero presenta algunos inconvenientes como el aumento considerable del espacio ocupado por la estructura derivado del aumento de punteros, la diferente configuración de los nodos de la lista, y la mayor complejidad de las inserciones y borrados de nodos.

En el Anexo A.3. se desarrollan en pseudocódigo los algoritmos correspondientes.

B) Segundo Método

El proceso será similar al descrito anteriormente, con la diferencia de que en éste primeramente se busca el índice de columna y a partir de ahí se recorre la columna hasta llegar a la fila del elemento en cuestión. En el primer método se localizaba el índice de fila y se recorría la fila hasta llegar a la columna del elemento al que se pretendía acceder.

En primer lugar habrá que realizar una búsqueda con el valor del índice de columna *(J)* en la lista ordenada de índices de columna.

Esta búsqueda comenzará a partir del primer nodo, que será el apuntado por la variable *ListaIC*. El avance se realizará en base al campo *SigIndice* que contiene para cada nodo índice la dirección del siguiente índice.

Finalizará cuando se alcance un nodo cuyo campo de información (*InfoIndice*) coincida con el valor del índice de columna *(J)* (búsqueda con éxito) o bien cuando se llegue al final de la lista o cuando se alcance un nodo cuyo campo de información sea mayor al valor del índice de columna (búsqueda fallida).

En el caso de que la búsqueda sea fallida el algoritmo devolverá un código de error indicando que el índice de columna se encuentra fuera de rango del conjunto de índices.

En el caso de que la búsqueda del índice de columna sea correcta, se podría comenzar a buscar el elemento en la lista de elementos del índice en cuestión. Ahora bien, en el caso de que dicho elemento no existiera, no se podría determinar si existía o no el índice de fila.

En determinadas aplicaciones, puede resultar interesante e incluso necesario el proporcionar un código de error en el caso de que no exista el índice de fila, por lo cual, previamente a la búsqueda del elemento, se realizará la búsqueda del índice de fila.

Esta búsqueda comenzará a partir del primer nodo de la lista de índices de fila, que será el apuntado por la variable *ListaIF*. El avance se realizará en base al campo *SigIndice* que contiene para cada nodo índice la dirección del siguiente índice.

Finalizará cuando se alcance un nodo cuyo campo de información (*InfoIndice*) coincida con el valor del índice de fila *(I)* (búsqueda con éxito) o bien cuando se llegue al final de la lista o cuando se alcance un nodo cuyo campo de información sea mayor al valor del índice de fila (búsqueda fallida).

En el caso de que exista el índice de fila, se puede comenzar a buscar el elemento, en la lista referenciada por el índice de columna. Para ello se deberá comenzar en el primer nodo, el cual estará apuntado por el campo *Primero* del nodo índice.

Una vez situados en un nodo deberemos comprobar si éste es el elemento buscado. Para ello haremos uso del campo *IndFila* del nodo, el cual contiene la dirección del nodo índice de la fila. Se pueden dar 3 casos:

a) Si el valor del campo de información del nodo índice de fila (*InfoIndice*) referenciado por *IndFila* es menor que valor del índice de fila *(I)*, deberemos continuar la búsqueda pasando al siguiente nodo en la lista de elementos del índice de columna, realizando el avance por medio del campo *SigColumna* del nodo, y repitiendo el tratamiento anterior.

b) Si el valor del campo de información del nodo índice de fila (*InfoIndice*) referenciado por *IndFila* coincide con el valor del índice de fila *(I)*, quiere decir que la búsqueda ha sido exitosa y estamos situados en el nodo buscado, con lo cual se da por finalizado el algoritmo.

c) Si el valor del campo de información del nodo índice de fila (*InfoIndice*) referenciado por *IndFila* es mayor que el valor del índice de fila *(I)*, la búsqueda se considerará fallida, ya que la lista de índices de fila está ordenada y por tanto el resto de índices de fila serán superiores al de referencia. Se dará por fallida la búsqueda también en el caso de llegar al final de la lista. En este tercer caso el algoritmo devolverá un código de error indicando que el elemento en cuestión no existe.

En este modelo se ha recorrido la columna correspondiente al índice *(J)* comenzando por el primer elemento de dicha columna y continuando en orden directo, de arriba a abajo, progresando de un nodo al siguiente por medio del campo *SigColumna*.

Si por alguna razón se sospechara que el elemento referenciado se encuentra más próximo al final de la columna que al principio, se podría haber comenzado la búsqueda por el final de la columna, situándose en el nodo apuntado por el campo *Ultimo* y continuando en sentido inverso de abajo a arriba, puesto que la lista es doblemente enlazada. El avance de un elemento al anterior se habría hecho en base al campo *AntColumna* de cada elemento.

El mismo criterio puede aplicarse a los procesos de localización del índice de columna *(J)* y del índice de fila *(I)*, los cuales se realizan en la lista de índices de columna desde *ListaIC* y en la lista de índices de fila desde *ListaIF*, respectivamente, avanzando en ellas por medio del campo *SigIndice*.

De forma similar a lo comentado anteriormente, si se pudiera deducir que alguno de estos índices está más próximo del final de su lista de índices que del comienzo de la misma, sería más interesante comenzar la búsqueda desde el final de la lista, en sentido inverso, en dirección al comienzo de la misma.

Para ello sería necesario que las listas de índices estuviesen implementadas como listas dobles, es decir, que cada nodo tuviera un puntero adicional (*AntIndice*) conteniendo la dirección del índice anterior.

Pero además, para poder comenzar la búsqueda por el índice mayor sería necesario mantener una variable *FinalIC* para la lista de índices de columna, y una variable *FinalIF* para la lista de índices de fila, conteniendo la dirección del último nodo de la lista de índices de columna y fila respectivamente.

Tal y como hemos comentado anteriormente, estos elementos adicionales (*AntIndice, FinalIF, FinalIC*) no los vamos a implementar en la estructura, debido a que en muy raros casos sería posible determinar si el valor buscado está más próximo del final que del comienzo.

Si el conjunto de índices fuese una serie de valores consecutivos sí sería posible establecer el extremo más adecuado para comenzar la búsqueda. Bastaría con calcular el índice mitad sumando los límites inferior y superior del conjunto de índices y dividiendo entre 2. Índice Mitad = (LIF + LSF) / 2.

Si el índice buscado fuese menor que el índice mitad, estaría en la primera mitad de la lista de índices, por lo que lo más rápido sería comenzar la búsqueda desde el comienzo. Sin embargo, si el índice buscado fuese mayor que el índice mitad, estaría en la segunda mitad de la lista de índices, y por tanto la búsqueda comenzaría desde el final de la lista de índices en orden inverso.

En la estructura propuesta de array bidimensional dinámico, una de las ventajas es precisamente que la serie de índices no tiene que estar formada por valores consecutivos, por lo cual no es aplicable el procedimiento anterior.

En el Anexo A.3. se desarrollan en pseudocódigo los algoritmos correspondientes.

Como se puede comprobar fácilmente, el acceso a un elemento en base a sus índices es mucho más eficaz en el caso del array bidimensional clásico que en el caso del ***ABD***, debido a la implementación física del primero en memoria, que garantiza la inmediatez de la operación. La ventaja del ***ABD*** consiste en el control del acceso incorrecto por índices fuera de rango.

2.3.4. Búsqueda de un elemento (o todos) en base a un campo de información

La operación de búsqueda consiste en localizar uno o varios elementos cuyos campos de información coincidan con un dato proporcionado.

Se irá recorriendo la estructura, comparando el campo de información de cada nodo tratado con el valor buscado. La finalización de dicho proceso vendrá determinada por la localización del elemento (búsqueda exitosa) o bien por la llegada al final de la estructura sin haber encontrado ningún elemento que tenga en su campo de información el valor buscado (búsqueda fallida).

Debemos hacer las siguientes consideraciones:

a) El punto de comienzo y el sentido de la búsqueda dentro de la estructura puede realizarse de muy diferentes maneras; por ejemplo:

- Comenzando en el primer elemento de la primera fila, procesando cada fila de izquierda a derecha y progresando hacia la fila inmediatamente inferior.

- Comenzando en el primer elemento de la primera columna, procesando cada columna de arriba a abajo y progresando hacia la columna inmediatamente adyacente a la derecha.

- Comenzando en el último elemento de la última fila, procesando cada fila de derecha a izquierda, y progresando hacia la fila inmediatamente superior.

- Comenzando en el último elemento de la última columna, procesando cada columna de abajo a arriba y progresando hacia la columna inmediatamente adyacente a la izquierda.

- Por Diagonales, en el sentido de la diagonal principal o en el de la secundaria y comenzando por diferentes vértices.

- Salteando filas o columnas.

Si se conociera a priori la zona del array donde puede encontrarse el elemento, sería más fácil el elegir el o los métodos más adecuados de recorrido. Ahora bien, como esto no suele ocurrir en la realidad más que en muy contadas ocasiones, utilizaremos en principio uno cualquiera de ellos.

b) Como ya se ha comentado en el capítulo de implementación, cada nodo tendrá uno o varios campos de información, en función de los datos que sea necesario almacenar en el mismo. En general nos referiremos a *InfoElemento* tanto si es un dato simple de información como si es un dato compuesto, bien entendido que en este último caso sería necesario explicitar en los algoritmos el dato individual utilizado en la búsqueda.

c) Puede ocurrir que el campo de información en base al cual se realiza la búsqueda no sea campo clave única (es decir, que cada elemento tenga un valor diferente al de los demás que le permita diferenciarse del resto). En ese caso, puede ocurrir que la búsqueda interese realizarla para todos los elementos que tengan su campo de información igual al valor buscado, o bien que sea suficiente con localizar el primero que cumpla dicha condición. En los algoritmos se diferenciarían ambas situaciones.

El proceso de búsqueda sobre array bidimensional clásico del primer elemento cuyo campo de información coincida con el valor buscado, y eligiendo por ejemplo, el criterio de recorrido por filas, comenzando en el primer elemento de la primera fila, y avanzando dentro de cada fila de izquierda a derecha, y pasando al primer elemento de la siguiente fila una vez completada la anterior, estaría implementado por medio de dos ciclos: el externo, que contendrá el tratamiento de las diferentes filas del array y el interno, que realizará el tratamiento de los elementos de cada fila.

En cada momento del recorrido (fila, columna) el proceso consistirá primero en determinar si la posición referenciada contiene un elemento válido o está vacía (lo cual vendrá indicado, tal y como se comentó en el caso de recorrido, por un valor especial).

Si la posición está vacía se deberá pasar al siguiente elemento de la matriz. Por el contrario si tiene un valor válido deberá compararse con el valor buscado. Si no coincide se continuará con el recorrido pasando a evaluar el elemento siguiente,

mientras que si coincide se puede dar por finalizada la búsqueda puesto que se ha hallado el elemento buscado.

Por tanto el proceso de búsqueda en este caso puede finalizar por dos motivos:

a) Se alcanza un elemento con campo de información igual al dato proporcionado (búsqueda con éxito).

b) Se alcanza el final del array sin haber encontrado ningún elemento coincidente con el dato proporcionado (búsqueda fallida).

En el caso en que sea necesario obtener todos los elementos de la estructura que cumplan el criterio de búsqueda, y utilizando el mismo sistema de recorrido, el proceso será similar, con la particularidad de que en este caso será necesario recorrer la matriz completa, a diferencia del caso anterior, que en el momento de encontrar el primer elemento que cumpliera la condición terminaba sin necesidad de completar el recorrido de todo el array.

En este caso, para cada posición referenciada se evaluará si está o no vacía, comparando en éste último caso su contenido con el del dato de búsqueda. Si coincide se aplicará el proceso que corresponda al elemento. En cualquiera de las situaciones anteriores se pasará a continuación a la siguiente posición del array, repitiendo el proceso hasta llegar al final de la estructura.

Al necesitar realizar un recorrido de la estructura para poder localizar un determinado elemento en base a su información, se plantean los mismos inconvenientes vistos en la operación de recorrido. Se accede a las posiciones de la matriz, sean válidas o no, es decir, almacenen la información de un elemento o no contengan ningún elemento.

En la estructura dinámica propuesta se accederá directamente a los elementos del array (no existen posiciones no ocupadas o no válidas), por lo que será tanto más ventajosa cuanto mayor sea el array y cuantas menos posiciones ocupadas tenga.

En el ***ABD*** el proceso de búsqueda del primer elemento que cumpla la condición, realizando el recorrido por filas, de izquierda a derecha, y de arriba a abajo, sería el siguiente:

El principio general es el mismo, y se puede implementar también en dos ciclos anidados: el externo trataría las diferentes filas del array y el interno procesaría los elementos de cada fila.

Para llevar a cabo este proceso se partiría de la dirección contenida en la variable *ListaIF*, que apuntaría al primer nodo de la lista de índices de fila, es decir al primer índice de fila. Este nodo tendrá en su campo *Primero* la dirección del primer elemento de la fila que representa. A partir de este nodo todos los demás de dicha fila estarán encadenados por su campo *SigFila*, de tal forma que se irán procesando en secuencia.

Para cada nodo tratado será necesario comparar su contenido de información *InfoElemento* con el valor de búsqueda. Si ambos datos coinciden se dará por finalizado el algoritmo, mientras que en caso contrario, se irán accediendo y evaluando los elementos de la fila hasta llegar a uno con el valor buscado en su campo de información (búsqueda con éxito) o bien a uno que tenga en su campo *SigFila* el valor nulo (indicador de elemento último de fila).

En este último caso, una vez tratados todos los elementos de una fila, se deberá pasar al siguiente índice de fila, es decir, se debe avanzar en la lista de índices al siguiente nodo al que acabamos de tratar en base a su campo *SigIndice*. Una vez situados en ese nuevo índice se debe repetir el proceso descrito anteriormente de ir accediendo y evaluando los elementos de esa fila.

Si no existe ningún elemento coincidente con el dato buscado, el algoritmo finalizará después de tratar los nodos de la última fila, que será la representada por el nodo índice que tenga en su campo *SigIndice* el valor nulo.

En el caso de que sea necesario tratar todos los elementos coincidentes en su campo de información con el valor buscado y el sistema de recorrido elegido sea el previamente descrito, el proceso únicamente diferirá del anterior en el criterio de terminación, que será siempre al llegar al final de la estructura.

Estos algoritmos de búsqueda sobre ***ABD*** tienen una clara ventaja respecto a los anteriores sobre array clásico, y que es la ya comentada en los algoritmos de recorrido sobre los que se apoyan estos de búsqueda: se accede únicamente a los elementos almacenados en la estructura, puesto que son los únicos que están presentes en la misma, ya que aquí no aparecen nunca posiciones vacías.

Cada acceso nos proporciona siempre un elemento a tratar, y existirán tantos accesos como elementos existan en la estructura. En el array tradicional existen tantos accesos como posiciones totales reservadas, complicándose el tratamiento debido a la necesidad de evaluar si existe o no información válida en la posición accedida, lo cual encarece los tiempos de ejecución frente al ***ABD***.

En el Anexo A.4. se desarrollan en pseudocódigo los algoritmos correspondientes.

2.3.5. Inserción de un elemento

La operación de inserción de un elemento en una estructura consiste en incluirlo en la misma en una determinada posición que se debe indicar, o bien en la posición que le corresponda según criterios de funcionamiento de la propia estructura (por ejemplo, en *pilas* y *colas* se inserta el elemento siempre por un extremo; en *colas de prioridad* se inserta por detrás de los de prioridad superior y misma prioridad, y antes de los de prioridad inferior; en estructuras ordenadas por algún campo de información de los elementos, el elemento a insertar se colocará en una posición que permita seguir conservando el orden de la estructura).

En el caso que nos ocupa, es decir, en una estructura de array, será necesario indicar la fila y la columna donde se desea insertar el elemento.

Para realizar la inserción de un elemento en una determinada fila y columna de un array estático, bastará con realizar un acceso a dicha posición, y a continuación comprobar si está ocupada o está vacía. En el caso de tener ya una información asignada devolver una condición de error, y en caso contrario asignar el elemento en dicha posición.

Puesto que se necesita realizar un acceso en base a los índices, el procedimiento de inserción es inmediato, pero tiene el inconveniente comentado anteriormente de romper la ejecución del programa o producir resultados imprevistos, en caso de que los índices estén fuera de rango.

En el caso del ***ABD*** será necesario realizar los siguientes pasos:

a) Comprobar la existencia del índice de fila, y si no existe crearlo.

b) Comprobar la existencia del índice de columna, y si no existe crearlo.

c) Comprobar la no existencia de un elemento en la fila y columna.

d) Si ya existe un elemento en esa posición, devolver una condición de error.

e) Si no existe, incluir el elemento en la lista de elementos referenciada por el índice de fila y en la lista de elementos referenciada por el índice de columna en la posición adecuada.

El proceso comenzará comprobando la existencia del índice de fila, para lo cual se irá procesando la lista de índices de fila de forma secuencial desde el primer elemento, que será el apuntado por la variable *ListaIF*. Para cada nodo se comprobará si su campo de información (*InfoIndice*) coincide con el índice buscado *(I)*. Para pasar de un índice al siguiente se hará en base al campo *SigIndice*.

Este recorrido finalizará cuando se produzca alguna de las situaciones siguientes:

a) El campo de información del nodo índice tratado coincide con *(I)*. Búsqueda con éxito.

b) El campo de información del nodo índice tratado es superior a *(I)*. Puesto que la lista de índices es una lista enlazada ordenada, se puede determinar que el índice buscado *(I)* no existe en la lista (todos los índices que restan de tratar serán superiores). Búsqueda sin éxito.

c) Se llega al final de la lista de índices. Búsqueda sin éxito.

En los casos b y c será necesario crear el índice de fila *(I)*, es decir, añadir un nodo a la lista de índices de fila con el valor *(I)*. Puesto que la lista deberá seguir conservando su criterio de ordenación, el índice en cuestión se añadirá antes del primer nodo índice mayor que él (punto de parada en el caso b), o al final de la lista (punto de parada en el caso c). Por lo tanto al campo *SigIndice* del nodo índice nuevo se le asignará, en el caso b, la dirección del nodo índice inmediatamente mayor a él, y en el caso c, el valor nulo.

En ambos casos el nodo índice se creará con el valor nulo en sus campos *Primero y Ultimo*, ya que en principio únicamente se crea el índice, sin existir elementos en la fila que representa.

En ambos casos también será necesario modificar el campo *SigIndice* del nodo índice precedente si existe, asignándole la dirección del nuevo nodo índice. En el caso de que el nuevo índice insertado sea el menor de la lista de índices, su dirección se deberá asignar a la variable *ListaIF*.

A continuación se deberá comprobar la existencia del índice de columna, para lo cual se irá procesando la lista de índices de columna de forma secuencial desde el límite inferior, que será el apuntado por la variable *ListaIC*. Para cada nodo se comprobará si su campo de información (*InfoIndice*) coincide con el índice buscado *(J)*. Para pasar de un índice al siguiente se hará en base al campo *SigIndice*.

Este recorrido finalizará cuando se produzca alguna de las situaciones siguientes:

a) El campo de información del nodo índice tratado coincide con *(J)*. Búsqueda con éxito.

b) El campo de información del nodo índice tratado es superior a *(J)*. Puesto que la lista de índices es una lista enlazada ordenada, se puede determinar que el índice buscado *(J)* no existe en la lista (todos los índices que restan de tratar serán superiores). Búsqueda sin éxito.

c) Se llega al final de la lista de índices. Búsqueda sin éxito.

En los casos b y c será necesario crear el índice de columna *(J)*, es decir, añadir un nodo a la lista de índices de columna con el valor *(J)*. Puesto que la lista deberá seguir conservando su criterio de ordenación, el índice en cuestión se añadirá antes del primer nodo índice mayor que él (punto de parada en el caso b), o al final de la lista (punto de parada en el caso c). Por lo tanto al campo *SigIndice* del nodo índice nuevo se le asignará, en el caso b, la dirección del nodo índice inmediatamente mayor a él, y en el caso c, el valor nulo.

En ambos casos el nodo índice se creará con el valor nulo en sus campos *Primero y Ultimo*, ya que en principio únicamente se crea el índice, sin existir elementos en la columna que representa.

En ambos casos también será necesario modificar el campo *SigIndice* del nodo índice precedente si existe, asignándole la dirección del nuevo nodo índice. En el caso de que el nuevo índice insertado sea el menor de la lista de índices, su dirección se deberá asignar a la variable *ListaIC*.

En el caso de que haya sido necesario crear alguno de los dos índices de fila y columna, lógicamente no existirá ningún nodo elemento en la posición referenciada por ambos índices, por lo que bastará con generar el nuevo nodo con la información correspondiente, e insertarlo en la posición correcta en la lista de elementos referenciada por el índice de fila y en la lista referenciada por el índice de columna.

Si por el contrario, no ha sido necesario generar ningún índice por existir ambos previamente, es posible que exista ya un elemento en la posición referenciada por ambos índices. Por lo tanto, habrá que realizar una operación de acceso para poder determinar tal circunstancia. Si existe se devolverá una condición de error, y en caso contrario se deberá generar el nodo e insertarlo en la posición correcta en las listas de elementos correspondientes a los índices de fila y de columna.

Se detallan a continuación los tratamientos a realizar en cada caso:

1) *Insertar1*. Necesario crear el índice de fila y el de columna.

En este caso, bastará con crear el nodo, completando su campo de información (*InfoElemento*) con el dato a insertar, y los campos que enlazan el nodo con el anterior y siguiente de su fila (*AntFila*, *SigFila*) y con el anterior y siguiente de su columna (*AntColumna*, *SigColumna*) a valor nulo, puesto que previamente no existían ni la fila ni la columna, por lo que el nodo generado será el único de su fila y el único de su columna.

Los campos del nodo que apuntan a su índice de fila (*IndFila*) y a su índice de columna (*IndColumna*), se cargarán con la dirección de dichos índices creados previamente.

Por último, en los índices de fila y columna creados, los campos que apuntan al primer nodo y al último de sus listas respectivas (*Primero*, *Ultimo*) se cargarán con la dirección del nodo de información generado, el cual será por la razón antes comentada, el primero y el último de su fila y el primero y último de su columna.

2) *Insertar2*. Necesario crear el índice de fila y no el de columna.

En este caso, al igual que el anterior, tenemos la seguridad de que no existe previamente el elemento a insertar, ya que no existía su índice de fila, y por tanto ha habido que crearlo.

Por ello se puede generar el nodo directamente, y completar su campo de información (*InfoElemento*) con el dato a insertar, y se puede asignar el valor nulo a sus campos de dirección del antecesor y sucesor en su fila (*AntFila*, *SigFila*), puesto que es el único elemento de su fila, ya que ésta no existía previamente.

Por esta razón, los campos del índice de fila que contienen la dirección del primer y del último elemento de dicha fila (*Primero*, *Ultimo*) deben ser inicializados con la dirección del nodo creado.

Asimismo, los campos del nodo que apuntan a su índice de fila y a su índice de columna (*IndFila*, *IndColumna*) se deberán cargar con la dirección de dichos índices, el primero de los cuales ha sido creado previamente, y el segundo localizado.

Por último falta enganchar el elemento en la posición adecuada en la lista que representa su columna. Para ello, se comenzará a recorrer dicha lista desde el principio (el primer nodo será el apuntado por el campo *Primero* del índice de columna), hasta llegar al elemento que se encuentre en la fila inmediatamente superior a la del nodo a insertar, o bien hasta llegar al final de la lista columna si el nodo a insertar es el de mayor fila de esa columna.

En este último recorrido se utilizarán dos punteros, de tal forma que al final del mismo, contengan las direcciones de los elementos predecesor y sucesor al nodo que se va a insertar en dicha columna. Estos valores son los que se asignarán respectivamente a los campos del predecesor y sucesor de columna del nodo (*AntColumna*, *SigColumna*).

Como consecuencia de incorporar el elemento en una determinada posición en la lista columna, en principio, será necesario modificar el campo del sucesor de columna de su nodo predecesor, y el campo del predecesor de columna de su nodo sucesor, asignándoles a ambos la dirección del nuevo nodo. Ahora bien, pueden darse las dos excepciones siguientes:

a) Que no exista el nodo predecesor, lo cual ocurrirá si el elemento a insertar tiene el menor índice de fila de los elementos de su columna

   En este caso, el nuevo nodo pasará a ser el primero de su columna, por lo que habrá que asignar su dirección al campo *Primero* del nodo índice de columna.

b) Que no exista el nodo sucesor, lo cual ocurrirá si el elemento a insertar tiene el mayor índice de fila de los elementos de su columna.

   En este caso, el nuevo nodo pasará a ser el último de su columna, por lo que habrá que asignar su dirección al campo *Ultimo* del nodo índice de columna.

Por supuesto, no pueden darse las dos excepciones simultáneamente, puesto que la columna ya existía previamente, lo cual quiere decir que tenía al menos un elemento. Por tanto el nuevo elemento incluido en dicha columna podrá ser el primero o el último de la misma, pero no ambas condiciones a la vez.

3) ***Insertar3*. Necesario crear el índice de columna y no el de fila.**

En este caso, al igual que en los dos anteriores, tenemos la seguridad de que no existe previamente el elemento a insertar, ya que no existía su índice de columna, y por tanto ha habido que crearlo.

Por ello se puede generar el nodo directamente, y completar su campo de información (*InfoElemento*) con el dato a insertar, y se puede asignar el valor nulo a sus campos de dirección del antecesor y sucesor en su columna (*AntColumna*, *SigColumna*), puesto que es el único elemento de su columna, ya que ésta no existía previamente.

Por esta razón, los campos del índice de columna que contienen la dirección del primer y del último elemento de dicha columna (*Primero*, *Ultimo*) deben ser inicializados con la dirección del nodo creado.

Asimismo, los campos del nodo que apuntan a su índice de fila y a su índice de columna (*IndFila*, *IndColumna*) se deberán cargar con la dirección de dichos índices, el primero de los cuales ha sido localizado, y el segundo creado.

Por último falta enganchar el elemento en la posición adecuada en la lista que representa su fila. Para ello, se comenzará a recorrer dicha lista desde el principio (el primer nodo será el apuntado por el campo *Primero* del índice de fila), hasta llegar al elemento que se encuentre en la columna inmediatamente superior a la del nodo a insertar, o bien hasta llegar al final de la lista de fila si el nodo a insertar es el de mayor columna de esa fila.

En este último recorrido se utilizarán dos punteros, de tal forma que al final del mismo, contengan las direcciones de los elementos predecesor y sucesor al nodo que se va a insertar en dicha fila. Estos valores son los que se asignarán

respectivamente a los campos del predecesor y sucesor de fila del nodo (*AntFila*, *SigFila*).

Como consecuencia de incorporar el elemento en una determinada posición en la lista fila, en principio, será necesario modificar el campo del sucesor de fila de su nodo predecesor, y el campo del predecesor de fila de su nodo sucesor, asignándoles a ambos la dirección del nuevo nodo. Ahora bien, pueden darse las dos excepciones siguientes:

a) Que no exista el nodo predecesor, lo cual ocurrirá si el elemento a insertar tiene el menor índice de columna de los elementos de su fila.

   En este caso, el nuevo nodo pasará a ser el primero de su fila, por lo que habrá que asignar su dirección al campo *Primero* del nodo índice de fila.

b) Que no exista el nodo sucesor, lo cual ocurrirá si el elemento a insertar tiene el mayor índice de columna de los elementos de su fila.

   En este caso, el nuevo nodo pasará a ser el último de su fila, por lo que habrá que asignar su dirección al campo *Ultimo* del nodo índice de fila.

Por supuesto, no pueden darse las dos excepciones simultáneamente, puesto que la fila ya existía previamente, lo cual quiere decir que tenía al menos un elemento. Por tanto el nuevo elemento incluido en dicha fila podrá ser el primero o el último de la misma, pero no ambas condiciones a la vez.

4) *Insertar4*. Existen el índice de fila y el de columna.

En este caso, a diferencia de los anteriores, es posible que exista ya un elemento en la estructura en la posición referenciada, por lo que habrá que comprobar esta condición y generar un código de error en caso de producirse.

Por lo tanto, se realizará una búsqueda en la lista referenciada por el índice de fila para localizar un posible nodo perteneciente a la columna indicada. Este proceso comenzará a partir del primer elemento de la fila (el apuntado por el campo *Primero* del nodo índice de fila) y finalizará cuando se cumpla alguna de estas tres situaciones:

a) Se alcanza el nodo perteneciente a la columna referenciada, es decir, aquel que apunta en su campo *IndColumna* al nodo de la lista de índices de columna cuyo campo de información tiene el valor del índice de columna *(J)*.

b) Se alcanza un nodo cuyo campo *IndColumna* apunta a un índice de columna de valor superior a *(J)*.

c) Se llega al final de la fila.

En el primer caso se proporcionará un código de error por existir ya el elemento referenciado por los índices [I , J], y finalizará el algoritmo.

En los casos b y c, se deduce que no existe elemento en esa posición, y el proceso de búsqueda habrá servido, mediante la ayuda de dos punteros, para establecer el predecesor y el sucesor del nodo a insertar en la fila *(I)*.

A continuación es necesario determinar la posición que deberá ocupar el elemento en su columna, por lo que se aplica el mismo esquema de búsqueda que en la fila, con la seguridad de que ahora no se dará el caso a (alcanzar un nodo perteneciente a la fila *(I)* referenciada), puesto que dicho elemento no ha sido localizado previamente en la búsqueda por fila. Se establece por medio de dos punteros el predecesor y el sucesor en la columna.

Tras generar el nodo, se completa su campo de información con el dato a añadir, y se asigna a los campos de predecesor y sucesor de fila y predecesor y sucesor de columna las direcciones obtenidas en los procesos de búsqueda anteriormente comentados.

Asimismo se asigna a los campos del nodo que apuntan a su índice de fila e índice de columna, las direcciones respectivas de dichos índices.

Por último falta actualizar los siguientes datos:

a) *El campo de sucesor de fila (SigFila) del nodo antecesor de fila.*

A dicho campo se le asignará la dirección del nuevo nodo generado, salvo en el caso excepcional de que no exista el nodo predecesor. Esta circunstancia se producirá cuando el elemento insertado sea el de menor índice de columna de los de su fila, en cuyo caso se asignará la dirección del nuevo nodo al campo *Primero* de su nodo índice de fila, ya que el nuevo elemento pasa a ser el primero de su fila.

b) *El campo de antecesor de fila (AntFila) del nodo sucesor de fila.*

A dicho campo se le asignará la dirección del nuevo nodo generado, salvo en el caso excepcional de que no exista el nodo sucesor. Esta circunstancia se producirá cuando el elemento insertado sea el de mayor índice de columna de los de su fila, en cuyo caso se asignará la dirección del nuevo nodo al campo *Ultimo* de su nodo índice de fila, ya que el nuevo elemento pasa a ser el último de su fila.

c) *El campo de sucesor de columna (SigColumna) del nodo antecesor de columna.*

A dicho campo se le asignará la dirección del nuevo nodo generado, salvo en el caso excepcional de que no exista el nodo predecesor. Esta circunstancia se producirá cuando el elemento insertado sea el de menor índice de fila de los de su columna, en cuyo caso se asignará la dirección del nuevo nodo al campo *Primero* de su nodo índice de columna, ya que el nuevo elemento pasa a ser el primero de su columna.

d) *El campo de antecesor de columna (AntColumna) del nodo sucesor de columna.*

A dicho campo se le asignará la dirección del nuevo nodo generado, salvo en el caso excepcional de que no exista el nodo sucesor. Esta circunstancia se producirá cuando el elemento insertado sea el de mayor índice de fila de los de su columna, en cuyo caso se asignará la dirección del nuevo nodo al campo *Ultimo* de su nodo índice de columna, ya que el nuevo elemento pasa a ser el último de su columna.

Las ventajas e inconvenientes del ***ABD*** frente al array estático en el algoritmo de inserción, dependen de la existencia o no de la posición referenciada:

- En el caso de que exista la posición referenciada (independientemente de que esté ocupada o no en el caso del array estático, ya que en el caso del ***ABD*** si existe estará ocupada con un elemento), será más eficiente el array estático que el ***ABD***, puesto que el acceso previo en el primer caso es inmediato, mientras que en el segundo es necesario un recorrido secuencial primero por las listas de indices, y a continuación por la lista de elementos de fila y/o por la lista de elementos de columna.

- En el caso de no existir la posición referenciada, el ***ABD*** ofrece mayores prestaciones que el array estático.

  En el caso de este último, si la posición no existe, quiere decir que al menos uno de los índices se encuentra fuera de rango del conjunto de índices correspondiente. Esto puede generar un error en ejecución si no está previamente controlado, y en cualquier caso se imposibilita la operación de inserción del nuevo elemento.

  Por el contrario en el ***ABD***, si alguno de los índices no existe, se crea automáticamente en tiempo de ejecución, y a continuación se inserta el elemento en la posición correspondiente, no imposibilitándose nunca la operación. La posibilidad de generar índices en ejecución de forma dinámica es una de las ventajas importantes de esta estructura.

En el Anexo A.5. se desarrollan en pseudocódigo los algoritmos correspondientes.

2.3.6. Borrado de un elemento referenciado por sus índices

La operación de borrado de un elemento en una estructura, referenciado por su posición física en la misma, consiste en acceder a dicha posición, y si existe proceder a eliminarlo de la misma.

En el caso de un array estático, bastará con realizar un acceso a dicha posición, y a continuación comprobar si está ocupada o está vacía. En el caso de no existir información asignada se devolverá una condición de error, y en caso contrario se procederá a eliminar el elemento de dicha posición, volviéndola a dejar vacia.

Puesto que se necesita realizar un acceso en base a los índices, el procedimiento de borrado es prácticamente inmediato, pero tiene el inconveniente ya comentado de romper la ejecución del programa o producir resultados imprevistos, en caso de que los índices estén fuera de rango.

En el caso de un array dinámico será necesario realizar los siguientes pasos:

a) Comprobar la existencia del índice de fila. Si no existe, devolver condición de error y fin del algoritmo.

b) Comprobar la existencia del índice de columna. Si no existe, devolver condición de error y fin del algoritmo.

c) Comprobar la existencia de un elemento en la fila y columna. Si no existe, devolver condición de error y fin del algoritmo.

d) Si existe, proceder a eliminarlo reajustando las listas de elementos de su índice de fila y de su índice de columna.

e) Reajustar, si procede, las listas de índices de fila y columna.

El proceso comenzará comprobando la existencia del índice de fila, para lo cual se procesará la lista de índices de fila de forma secuencial desde el primer elemento, que será el apuntado por la variable *ListaIF*. Para cada nodo se

comprobará si su campo de información (*InfoIndice*) coincide con el índice buscado *(I)*. Para pasar de un índice al siguiente se hará en base al campo *SigIndice*.

Este recorrido finalizará cuando se produzca alguna de las situaciones siguientes:

a) El campo de información del nodo índice tratado coincide con *(I)*. Búsqueda con éxito.

b) El campo de información del nodo índice tratado es superior a *(I)*. Puesto que la lista de índices es una lista enlazada ordenada, se puede determinar que el índice buscado *(I)* no existe en la lista (todos los índices que restan de tratar serán superiores). Búsqueda sin éxito.

c) Se llega al final de la lista de índices. Búsqueda sin éxito.

En los casos b y c, se indicará mediante un código de error la no existencia del índice de fila, dando por finalizado el algoritmo.

A continuación se deberá comprobar la existencia del índice de columna, para lo cual se procesará la lista de índices de columna de forma secuencial desde el límite inferior, que será el apuntado por la variable *ListaIC*. Para cada nodo se comprobará si su campo de información (*InfoIndice*) coincide con el índice buscado *(J)*. Para pasar de un índice al siguiente se hará en base al campo *SigIndice*.

Este recorrido finalizará cuando se produzca alguna de las situaciones siguientes:

a) El campo de información del nodo índice tratado coincide con *(J)*. Búsqueda con éxito.

b) El campo de información del nodo índice tratado es superior a *(J)*. Puesto que la lista de índices es una lista enlazada ordenada, se puede determinar que el índice buscado *(J)* no existe en la lista (todos los índices que restan de tratar serán superiores). Búsqueda sin éxito.

c) Se llega al final de la lista de índices. Búsqueda sin éxito.

En los casos b y c, se indicará mediante un código de error la no existencia del índice de columna, dando por finalizado el algoritmo.

Si los dos índices de fila y columna existen, será necesario realizar una operación de acceso para poder determinar si existe un elemento referenciado por dichos índices. Si no existe se devolverá una condición de error, y en caso contrario se deberá proceder a eliminar el elemento reajustando las listas de elementos referenciadas por los índices de fila y columna.

Puede darse el caso de que como consecuencia de eliminar el elemento, se queden vacías la fila y/o la columna en las cuales estaba incluido. Esta situación se produciría en el caso de que el elemento fuese el único de su fila y/o el único de su columna. En tal situación, se procederá a eliminar de la lista de índices de fila el nodo correspondiente al índice *(I)* (si el elemento era el único de su fila), y/o a eliminar de la lista de índices de columna el nodo correspondiente al índice *(J)* (si el elemento era el único de su columna).

El acceso al elemento se puede realizar recorriendo su fila, o bien recorriendo su columna. Utilizando el primer método, se comenzará a recorrer la lista de elementos de la fila *(I)* desde el principio (el primer nodo será el apuntado por el campo *Primero* del índice de fila *(I)*), y se finalizará cuando se cumpla alguna de las condiciones siguientes:

a) Al llegar a un nodo que se encuentre en la columna *(J)* (nodo a borrar) (búsqueda con éxito).

b) Al llegar a un nodo de columna superior a *(J)*. (búsqueda fallida)

c) Al llegar al final de la lista de fila. (*(J)* > Índices de columna de los nodos de esa fila) (búsqueda fallida)

En el segundo método, se comenzará a recorrer la lista de elementos de la columna *(J)* desde el principio (el primer nodo será el apuntado por el campo *Primero* del índice de columna *(J)*), y se finalizará cuando se cumpla alguna de las condiciones siguientes:

a) Al llegar a un nodo que se encuentre en la fila *(l)* (nodo a borrar) (búsqueda con éxito).

b) Al llegar a un nodo de fila superior a *(l)*. (búsqueda fallida)

c) Al llegar al final de la lista de columna. (*(l)* > índices de fila de los nodos de esa columna) (búsqueda fallida)

En cualquiera de ambos métodos, la finalización del proceso por los casos b y c implica que el elemento a borrar no existe, por lo que se devolverá un código de error indicando tal circunstancia.

Utilizando el método primero, se podía haber omitido en el procedimiento general de borrado, el paso b (comprobación del índice de columna), ya que no es necesario para la localización del elemento en el paso c. Sin embargo, es conveniente incluirlo, para en el caso de no existir el elemento, diferenciar entre la no existencia del índice de columna, y en caso de existir éste, la no existencia del elemento.

De la misma forma, utilizando el segundo método, se podía haber omitido en el procedimiento general de borrado, el paso a (comprobación del índice de fila), ya que no es necesario para la localización del elemento en el paso c. Sin embargo, es conveniente incluirlo, para en el caso de no existir el elemento, diferenciar entre la no existencia del índice de fila, y en caso de existir éste, la no existencia del elemento.

En el caso de existir el elemento a borrar (caso a), se procederá a su eliminación, y al reajuste de punteros en sus nodos predecesor y sucesor de fila y predecesor y sucesor de columna, tal y como se indica:

a) *El campo de sucesor de fila del nodo antecesor de fila (AntFila^.SigFila).*

A dicho campo se le asignará la dirección del nodo sucesor de fila al que se va a eliminar, salvo en el caso excepcional de que no exista el nodo antecesor. Esta circunstancia se producirá cuando el elemento borrado sea el de menor índice de columna de los de su fila, en cuyo caso se asignará la dirección del

nodo sucesor al campo *Primero* del nodo índice de fila, ya que dicho nodo sucesor pasa a ser el primero de su fila.

b) *El campo de antecesor de fila del nodo sucesor de fila (SigFila^.AntFila).*

A dicho campo se le asignará la dirección del nodo predecesor de fila al que se va a eliminar, salvo en el caso excepcional de que no exista el nodo sucesor. Esta circunstancia se producirá cuando el elemento borrado sea el de mayor índice de columna de los de su fila, en cuyo caso se asignará la dirección del nodo antecesor al campo *Ultimo* del nodo índice de fila, ya que dicho nodo antecesor pasa a ser el último de su fila.

c) *El campo de sucesor de columna del nodo antecesor de columna (AntColumna^.SigColumna).*

A dicho campo se le asignará la dirección del nodo sucesor de columna al que se va a eliminar, salvo en el caso excepcional de que no exista el nodo antecesor. Esta circunstancia se producirá cuando el elemento borrado sea el de menor índice de fila de los de su columna, en cuyo caso se asignará la dirección del nodo sucesor al campo *Primero* del nodo índice de columna, ya que dicho nodo sucesor pasa a ser el primero de su columna.

d) *El campo de antecesor de columna del nodo sucesor de columna (SigColumna^.AntColumna).*

A dicho campo se le asignará la dirección del nodo antecesor de columna al que se va a eliminar, salvo en el caso excepcional de que no exista el nodo sucesor. Esta circunstancia se producirá cuando el elemento borrado sea el de mayor índice de fila de los de su columna, en cuyo caso se asignará la dirección del nodo antecesor al campo *Ultimo* del nodo índice de columna, ya que el nodo antecesor pasa a ser el último de su columna.

Una vez eliminado el elemento, y reajustado los punteros de sus nodos adyacentes, será necesario comprobar si la fila y/o la columna donde se encontraba el elemento han quedado vacías, en cuyo caso se deberá proceder a eliminar los índices de fila y/o de columna.

Si tras el borrado del elemento, el nodo índice de fila ha quedado con su campo *Primero* (o bien su campo *Ultimo)* a valor nulo, quiere decir que el elemento eliminado era el único de su fila. En tal situación se debe proceder a eliminar el nodo índice de fila, puesto que la fila que representa ha quedado vacía.

El procedimiento de búsqueda del índice de fila, aplicado al comienzo del algoritmo de borrado, devuelve la dirección del índice anterior en la lista de índices. En este nodo será necesario modificar el campo *SigIndice*, al cual se le asignará la dirección del nodo índice siguiente al que se elimina.

Si el índice de fila que se borra es el menor de su conjunto de índices, no tendrá nodo predecesor, por lo que en ese caso se deberá modificar la variable *ListaIF*, a la cual se asignará la dirección del nodo índice sucesor al que se elimina.

De la misma forma, si tras el borrado del elemento, el nodo índice de columna ha quedado con su campo *Primero* (o bien su campo *Ultimo)* a valor nulo, quiere decir que el elemento eliminado era el único de su columna. En tal situación se debe proceder a eliminar el nodo índice de columna, puesto que la columna que representa ha quedado vacía.

El procedimiento de búsqueda del índice de columna, aplicado al comienzo del algoritmo de borrado, devuelve la dirección del índice anterior en la lista de índices. En este nodo será necesario modificar el campo *SigIndice*, al cual se le asignará la dirección del nodo índice siguiente al que se elimina.

Si el índice de columna que se borra es el menor de su conjunto de índices, no tendrá nodo predecesor, por lo que en ese caso se deberá modificar la variable *ListaIC*, a la cual se asignará la dirección del nodo índice sucesor al que se elimina.

Comparando la operación de borrado de un elemento referenciado por sus índices en un array estático y en el ***ABD***, la ventaja que presenta el primero sobre el segundo viene dada por el acceso previo, que en el array estático es inmediato y en el dinámico exige un recorrido secuencial primero en las listas de índices, y a continuación en la de elementos de su fila o columna.

Esta ventaja del acceso por índices se convierte en inconveniente cuando la posición referenciada no existe debido a que los índices se encuentran fuera de

rango, lo cual provocará la interrupción de la ejecución del programa o la obtención de resultados incorrectos. En el ***ABD***, por el contrario, tal error se encuentra controlado.

Por otra parte, en el array estático el borrado de un elemento no implica la recuperación del espacio que ocupaba, mientras que en el ***ABD*** el elemento se destruye y desaparece físicamente de memoria liberando el espacio que ocupaba en la misma, pudiendo desaparecer también los índices de fila y columna si sólo representaban a ese elemento.

En el Anexo A.6. se desarrolla en pseudocódigo el algoritmo correspondiente.

2.3.7. Borrado de un elemento (o todos) en base a un campo de información

La operación de borrado de un elemento en base a un campo de información consiste en realizar una búsqueda del elemento, y a continuación si existe proceder a eliminarlo de la estructura.

Puesto que en principio es necesario buscar el elemento, son válidas las consideraciones hechas en el algoritmo de búsqueda: si el campo de información utilizado en la búsqueda no es campo clave, pueden existir varios elementos que contengan el dato a borrar, por lo cual si se opta por eliminar el primero que se encuentre en el proceso de búsqueda, el elemento seleccionado será diferente según la forma de recorrido que se aplique.

En el diseño de los algoritmos consideraremos también el caso de borrar todos los elementos cuyo campo de información tenga el dato proporcionado.

El proceso de borrado sobre array bidimensional clásico del primer elemento cuyo campo de información coincida con el valor proporcionado, aplicará el algoritmo de búsqueda utilizando un determinado criterio de recorrido (por ej.

criterio de recorrido por filas y de izquierda a derecha, o por columnas y de arriba a abajo, etc.).

Para cada posición referenciada será necesario comprobar si existe un elemento en dicha posición (es decir, si contiene un dato diferente al valor especial de posición vacía), y en caso afirmativo, comprobar si coincide con el dato proporcionado. Si coincide, se procederá a eliminar el elemento sustituyendo su información por el valor especial indicador de posición vacía, dando por finalizado el algoritmo.

Si la posición no contiene información válida, o bien ésta no coincide con el dato suministrado se pasará a evaluar el elemento siguiente.

Si en el array no existe ninguna posición que contenga el dato de borrado, el algoritmo finalizará cuando se llegue en el recorrido al final de la estructura.

Por tanto el proceso de borrado puede finalizar por dos motivos:

a) Se alcanza un elemento con campo de información igual al dato proporcionado, y se procede a su borrado.

b) Se alcanza el final del array sin haber encontrado ningún elemento coincidente con el dato proporcionado.

En el caso en que sea necesario eliminar todos los elementos de la estructura que cumplan el criterio de borrado, el proceso será similar, con la particularidad de que en este caso será necesario recorrer la matriz completa, a diferencia del caso anterior, que en el momento de encontrar el primer elemento que cumpliera la condición terminaba sin necesidad de recorrer todo el array.

En este caso, para cada posición referenciada se evaluará si está o no vacía, comparando en éste último caso su contenido con el del dato a borrar. Si coincide se procederá a eliminar el elemento, asignando a dicha posición el indicador de posición vacía.

En cualquiera de las situaciones anteriores se pasará a continuación a la siguiente posición del array, repitiendo el proceso hasta llegar al final de la estructura.

Al necesitar realizar un recorrido de la estructura para poder localizar un determinado elemento en base a su información, se plantean los mismos inconvenientes vistos en la operación de recorrido. Se accede a las posiciones de la matriz, sean válidas o no, es decir, almacenen la información de un elemento o no contengan ningún elemento.

En el *ABD* se accederá directamente a los elementos del array (no existen posiciones no ocupadas o no válidas), por lo que será tanto más ventajoso frente al estático cuanto mayores sean las dimensiones del array y cuantos menos elementos tenga.

En el *ABD* el proceso de borrado del primer elemento que contenga la información proporcionada, se basará también en el algoritmo de búsqueda sobre array dinámico.

Aplicando un sentido de recorrido determinado, se comparará el campo de información de cada elemento con el dato de borrado. Dicho proceso finalizará cuando se cumpla alguna de las dos condiciones siguientes:

a) Se accede a un elemento cuyo campo de información coincide con el dato a borrar. Se procede a eliminar dicho elemento, y se reorganizan las listas de elementos de su fila y de su columna, y las de índices si fuese necesario.

b) Se alcanza el final del array sin encontrar ningún elemento coincidente en su campo de información con el dato suministrado.

Los reajustes de punteros de los nodos adyacentes en las listas de elementos de su fila y de su columna, así como la eliminación de los índices de fila y/o columna si el elemento borrado era el único de su fila y/o columna respectivamente, se realizarán de la misma forma descrita en el algoritmo de borrado anterior.

En el caso de que sea necesario borrar todos los elementos coincidentes en su campo de información con el valor dado, el proceso únicamente diferirá del anterior en el criterio de terminación, que será siempre al llegar al final de la estructura.

Además, en este caso, se deberá proceder a revisar las listas de índices de fila y columna, y se eliminarán de las mismas todos los nodos índices, que como consecuencia del borrado de elementos de la estructura, se hayan quedado con su lista de elementos vacía (campo *Primero* o *Ultimo* con valor nulo).

El algoritmo de borrado de un elemento (o todos) en base a un campo de información sobre ***ABD*** tiene una clara ventaja respecto al realizado sobre array clásico, y es la ya comentada en los algoritmos de búsqueda sobre los que se apoyan estos de borrado: se accede únicamente a los elementos almacenados en la estructura, puesto que son los únicos que están presentes en la misma, ya que aquí no aparecen nunca posiciones vacías.

Cada acceso proporciona siempre un elemento a tratar, y existirán tantos accesos como elementos existan en la estructura. En el array tradicional existen tantos accesos como posiciones totales reservadas, complicándose el tratamiento debido a la necesidad de evaluar si existe o no información válida en la posición accedida, lo cual encarece los tiempos de ejecución frente al array dinámico.

Ligado a lo anterior, el array estático presenta el inconveniente de que el borrado de un elemento no implica la recuperación del espacio que ocupaba (siempre ocupa el mismo espacio), mientras que en el ***ABD*** el elemento se destruye y desaparece físicamente de memoria liberando el espacio que ocupaba en la misma, pudiendo desaparecer también los índices de fila y columna que, como consecuencia de la eliminación de los elementos que referenciaban, dejan de tener sentido.

En el Anexo A.7. se desarrollan en pseudocódigo los algoritmos correspondientes.

2.3.8. Modificación de un elemento referenciado por sus índices

La operación de modificación de un elemento en una estructura, referenciado por su posición física en la misma, consiste en acceder a dicha posición, y si existe, proceder a sustituir su campo o campos de información con un nuevo dato proporcionado.

En el caso de un array estático, bastará con realizar un acceso a dicha posición referenciada por los índices de fila y columna, y a continuación comprobar si está ocupada o está vacía. En el caso de no existir información asignada se devolverá una condición de error, y en caso contrario se procederá a sustituir la información existente por la nueva información suministrada.

Puesto que se necesita realizar un acceso en base a los índices, el procedimiento de modificación es prácticamente inmediato, pero tiene el inconveniente ya comentado anteriormente de romper la ejecución del programa o producir resultados imprevistos, en caso de que los índices estén fuera de rango.

En el caso de un *ABD* será necesario realizar los siguientes pasos:

a) Comprobar la existencia del índice de fila. Si no existe, devolver condición de error y fin del algoritmo.

b) Comprobar la existencia del índice de columna. Si no existe, devolver condición de error y fin del algoritmo.

c) Comprobar la existencia del elemento en la fila y columna. Si no existe, devolver condición de error y fin del algoritmo.

d) Si existe, proceder a reemplazar su campo o campos de información por el nuevo o los nuevos datos.

El proceso comenzará comprobando la existencia del índice de fila, para lo cual se procesará la lista de índices de fila de forma secuencial desde el primer elemento, que será el apuntado por la variable *ListaIF*. Para cada nodo se

comprobará si su campo de información (*InfoIndice*) coincide con el Indice buscado *(I)*. Para pasar de un indice al siguiente se hará en base al campo *SigIndice*.

Este recorrido finalizará cuando se produzca alguna de las situaciones siguientes:

a) El campo de información del nodo Indice tratado coincide con *(I)*. Búsqueda con éxito.

b) El campo de información del nodo índice tratado es superior a *(I)*. Puesto que la lista de índices es una lista enlazada ordenada, se puede determinar que el índice buscado *(I)* no existe en la lista (todos los índices que restan de tratar serán superiores). Búsqueda sin éxito.

c) Se llega al final de la lista de índices. Búsqueda sin éxito.

En los casos b y c, se indicará mediante un código de error la no existencia del índice de fila, dando por finalizado el algoritmo.

A continuación se deberá comprobar la existencia del índice de columna, para lo cual se procesará la lista de índices de columna de forma secuencial desde el límite inferior, que será el apuntado por la variable *ListaIC*. Para cada nodo se comprobará si su campo de información (*InfoIndice*) coincide con el índice buscado *(J)*. Para pasar de un indice al siguiente se hará en base al campo *SigIndice*.

Este recorrido finalizará cuando se produzca alguna de las situaciones siguientes:

a) El campo de información del nodo índice tratado coincide con *(J)*. Búsqueda con éxito.

b) El campo de información del nodo índice tratado es superior a *(J)*. Puesto que la lista de índices es una lista enlazada ordenada, se puede determinar que el índice buscado *(J)* no existe en la lista (todos los índices que restan de tratar serán superiores). Búsqueda sin éxito.

c) Se llega al final de la lista de índices. Búsqueda sin éxito.

En los casos b y c, se indicará mediante un código de error la no existencia del índice de columna, dando por finalizado el algoritmo.

Si los dos índices de fila y columna existen, será necesario realizar una operación de acceso para poder determinar si existe un elemento referenciado por dichos índices. Si no existe se devolverá una condición de error, y en caso contrario se deberá proceder a modificar el dato o datos de información del elemento.

El acceso al elemento se puede realizar recorriendo su fila, o bien recorriendo su columna. En el primer método, se comenzará a recorrer la lista de elementos de la fila *(I)* desde el principio (el primer nodo será el apuntado por el campo *Primero* del índice de fila *(I)*), y se finalizará cuando se cumpla alguna de las condiciones siguientes:

a) Al llegar a un nodo que se encuentre en la columna *(J)* (búsqueda con éxito).

b) Al llegar a un nodo de columna superior a *(J)*. (búsqueda fallida)

c) Al llegar al final de la lista de fila. (*(J)* > índices de columna de los nodos de esa fila) (búsqueda fallida)

En el segundo método, se comenzará a recorrer la lista de elementos de la columna *(J)* desde el principio (el primer nodo será el apuntado por el campo *Primero* del índice de columna *(J)*), y se finalizará cuando se cumpla alguna de las condiciones siguientes:

a) Al llegar a un nodo que se encuentre en la fila *(I)* (búsqueda con éxito).

b) Al llegar a un nodo de fila superior a *(I)*. (búsqueda fallida)

c) Al llegar al final de la lista de columna. (*(I)* > índices de fila de los nodos de esa columna) (búsqueda fallida)

En cualquiera de ambos métodos, la finalización del proceso por los casos b y c implica que el elemento a modificar no existe, por lo que se devolverá un código de error indicando tal circunstancia.

Utilizando el método primero, se podía haber omitido en el procedimiento general de modificación, el paso b (comprobación del índice de columna), ya que no es necesario para la localización del elemento en el paso c. Sin embargo, es conveniente incluirlo, para en el caso de no existir el elemento, diferenciar entre la no existencia del índice de columna, y en caso de existir éste, la no existencia del elemento.

De la misma forma, utilizando el segundo método, se podía haber omitido en el procedimiento general de modificación, el paso a (comprobación del índice de fila), ya que no es necesario para la localización del elemento en el paso c. Sin embargo, es conveniente incluirlo, para en el caso de no existir el elemento, diferenciar entre la no existencia del índice de fila, y en caso de existir éste, la no existencia del elemento.

En el caso de existir el nodo a modificar (caso a), se procederá a sustituir su información actual por la nueva.

Comparando la operación de modificación de un elemento referenciado por sus índices en el array estático y en el ***ABD***, la ventaja que presenta el primero sobre el segundo viene dada por el acceso previo, que en el array estático es inmediato y en el dinámico exige un recorrido secuencial primero en las listas de índices, y a continuación en la de elementos de su fila o columna.

Sin embargo, y como ya se ha comentado con anterioridad, ese acceso inmediato es también un inconveniente si los índices están fuera de rango, ya que provoca la interrupción del programa durante su ejecución o la obtención de resultados erróneos.

En el Anexo A.8. se desarrolla en pseudocódigo el algoritmo correspondiente.

2.3.9. Modificación de un elemento (o todos) en base a un campo de información

La operación de modificación de un elemento localizado en base a un campo de información, consiste en realizar una búsqueda del elemento, y a continuación si existe, proceder a sustituir su información por la nueva suministrada.

Por lo tanto el algoritmo necesitará como parámetros de entrada, el dato en base al cual se ha de realizar la búsqueda y que va a ser modificado, y el dato nuevo que sustituirá al anterior.

Puesto que en principio es necesario buscar el elemento, son válidas las consideraciones hechas en el algoritmo de búsqueda: si el campo de información utilizado en la búsqueda no es campo clave, pueden existir varios elementos que contengan el dato a modificar, por lo cual si se opta por modificar el primero que se encuentre en el proceso de búsqueda, el elemento seleccionado será diferente según la forma de recorrido que se aplique.

En el diseño de los algoritmos consideraremos también el caso de modificar todos los elementos cuyo campo de información tenga el dato de búsqueda.

El proceso de modificación sobre array bidimensional clásico del primer elemento cuyo campo de información coincida con el valor proporcionado, aplicará el algoritmo de búsqueda utilizando un determinado criterio de recorrido (por ej. criterio de recorrido por filas y de izquierda a derecha, o por columnas y de arriba a abajo, etc.).

Para cada posición referenciada será necesario comprobar si existe un elemento en dicha posición (es decir, si contiene un dato diferente al valor especial de posición vacía), y en caso afirmativo, comprobar si coincide con el dato proporcionado. Si coincide, se procederá a sustituir su información por la nueva, dando por finalizado el algoritmo. Si la posición no contiene información válida, o bien ésta no coincide con el dato suministrado, se pasará a evaluar el elemento siguiente.

Si en el array no existe ninguna posición que contenga el dato de búsqueda, el algoritmo finalizará cuando se llegue en el recorrido al final de la estructura.

Por tanto el proceso de modificación puede finalizar por dos motivos:

a) Se alcanza un elemento con campo de información igual al dato proporcionado, y se procede a sustituir su información por la nueva.

b) Se alcanza el final del array sin haber encontrado ningún elemento coincidente con el dato proporcionado.

En el caso en que sea necesario modificar todos los elementos de la estructura cuya información coincida con el dato de búsqueda, el proceso será similar, con la particularidad de que en este caso será necesario recorrer la matriz completa, a diferencia del caso anterior que, en el momento de encontrar el primer elemento que cumpliera la condición, terminaba sin necesidad de recorrer todo el array.

En este caso, para cada posición referenciada se evaluará si está o no vacía, comparando en éste último caso su contenido con el del dato a modificar. Si coincide se procederá a sustituir su información por la nueva.

En cualquiera de las situaciones anteriores se pasará a continuación a la siguiente posición del array, repitiendo el proceso hasta llegar al final de la estructura.

Al necesitar realizar un recorrido de la estructura para poder localizar un determinado elemento en base a su información, se plantean los mismos inconvenientes vistos en la operación de recorrido. Se accede a las posiciones de la matriz, sean válidas o no, es decir, almacenen la información de un elemento o no contengan ningún elemento.

En el ***ABD*** se accederá directamente a los elementos del array (no existen posiciones no ocupadas o no válidas), por lo que será tanto más ventajoso frente al estático cuanto mayores sean las dimensiones del array y cuantos menos elementos tenga.

En el *ABD* el proceso de modificación del primer elemento que contenga la información proporcionada, se basará también en el algoritmo de búsqueda sobre array dinámico. Aplicando un sentido de recorrido determinado, se comparará el campo de información de cada elemento con el dato a modificar. Dicho proceso finalizará cuando se cumpla alguna de las dos condiciones siguientes:

a) Se accede a un elemento cuyo campo de información coincide con el dato a modificar. Se procede a reemplazar su información por la nueva.

b) Se alcanza el final del array sin encontrar ningún elemento coincidente en su campo de información con el dato suministrado de búsqueda.

En el caso de que sea necesario modificar todos los elementos coincidentes en su campo de información con el valor dado, el proceso únicamente diferirá del anterior en el criterio de terminación, que será siempre al llegar al final de la estructura.

El algoritmo de modificación de un elemento (o todos) buscado en base a un campo de información sobre *ABD* tienen una clara ventaja respecto al realizado sobre array clásico, y que es la ya comentada en los algoritmos de búsqueda sobre los que se apoyan estos de modificación: se accede únicamente a los elementos almacenados en la estructura, puesto que son los únicos que están presentes en la misma, ya que aquí no aparecen nunca posiciones vacias.

Cada acceso proporciona siempre un elemento a tratar, y existirán tantos accesos como elementos existan en la estructura. En el array tradicional existen tantos accesos como posiciones totales reservadas, complicándose el tratamiento debido a la necesidad de evaluar si existe o no información válida en la posición accedida, lo cual encarece los tiempos de ejecución frente al array dinámico.

La desventaja del array estático frente al *ABD* en este proceso será, al igual que en todos los que se basan en búsquedas, y por tanto en recorridos, tanto mayor cuantas más posiciones vacias existan en el array.

En el Anexo A.9. se desarrollan en pseudocódigo los algoritmos correspondientes.

2.3.10. Eliminación de una fila

Este proceso consiste en localizar el índice a eliminar en la lista de índices de fila y proceder a su borrado, suprimiendo también todos los elementos que se encuentran en dicha fila, reorganizando las listas de las columnas de dichos elementos, y reajustando si procede la lista de índices de columna.

La eliminación de un índice es un proceso no implementable en un array estático, puesto que en éste el tamaño es siempre fijo en tiempo de ejecución, al igual que sus conjuntos de índices. No es posible ni añadir, ni modificar, ni borrar un índice.

Seria posible borrar todos los elementos de una fila simplemente sustituyendo su información por la marca de posición vacía, pero no se podría recuperar el espacio físico que ocupaban en memoria, debido a su naturaleza estática.

El ***ABD*** posibilita el reajuste de su dimensionamiento; los conjuntos de índices pueden modificarse de forma dinámica en tiempo de ejecución, por lo que será posible no sólo borrar, sino también insertar (visto en el algoritmo de inserción de un elemento) y modificar los índices. Además el borrado de los elementos incluidos en la fila a eliminar implica la recuperación del espacio físico que ocupaban.

El algoritmo comenzará realizando una búsqueda en la lista de índices de fila del índice a borrar *(I)*, para lo cual se procesará dicha lista de forma secuencial desde el primer elemento, que será el apuntado por la variable *ListaIF*. Para cada nodo se comprobará si su campo de información (*InfoIndice*) coincide con el índice buscado *(I)*. Para pasar de un índice al siguiente se hará en base al campo *SigIndice*.

Este recorrido finalizará cuando se produzca alguna de las situaciones siguientes:

a) El campo de información del nodo índice tratado coincide con *(I)*. Búsqueda con éxito.

b) El campo de información del nodo índice tratado es superior a *(I)*. Puesto que la lista de índices es una lista enlazada ordenada, se puede determinar que el índice buscado *(I)* no existe en la lista (todos los índices que restan de tratar serán superiores). Búsqueda sin éxito.

c) Se llega al final de la lista de índices. Búsqueda sin éxito.

En los casos b y c, se indicará mediante un código de error la no existencia del índice de fila a borrar, dando por finalizado el algoritmo.

Si el índice existe, se puede proceder a eliminarlo, salvando previamente la dirección del primer nodo de la lista de elementos de dicha fila. Dicha dirección estará almacenada en el campo *Primero* del nodo índice.

Para eliminar el nodo índice, además de destruirlo de memoria, será necesario asignar la dirección de su índice sucesor a la variable *ListaIF* si el índice borrado era el menor de la lista de índices, o en caso contrario, al campo *SigIndice* del índice predecesor al borrado.

A continuación será necesario eliminar todos los elementos de la fila en cuestión, reajustando para cada uno de ellos la lista de elementos de la columna en la que están situados.

Este proceso comenzará a partir del primer elemento de la fila, cuya dirección se ha salvado previamente. Se avanzará al siguiente elemento en base al campo puntero *SigFila* y finalizará el recorrido tras procesar el último elemento de la fila, que será aquel que tenga el valor nulo en su campo *SigFila*.

Cada elemento tratado se destruirá de memoria, y se actualizarán punteros en sus nodos predecesor y sucesor de columna, de la forma siguiente:

a) *El campo de sucesor de columna del nodo antecesor de columna (AntColumna^.SigColumna).*

A dicho campo se le asignará la dirección del nodo sucesor de columna al que se va a eliminar, salvo en el caso excepcional de que no exista el nodo antecesor. Esta circunstancia se producirá cuando el elemento borrado sea el de menor índice de fila de los de su columna, en cuyo caso se asignará la dirección del nodo sucesor al campo *Primero* del nodo índice de columna, ya que dicho nodo sucesor pasa a ser el primero de su columna.

b) *El campo de antecesor de columna del nodo sucesor de columna (SigColumna^.AntColumna).*

A dicho campo se le asignará la dirección del nodo antecesor de columna al que se va a eliminar, salvo en el caso excepcional de que no exista el nodo sucesor. Esta circunstancia se producirá cuando el elemento borrado sea el de mayor índice de fila de los de su columna, en cuyo caso se asignará la dirección del nodo antecesor al campo *Ultimo* del nodo índice de columna, ya que el nodo antecesor pasa a ser el último de su columna.

Tras eliminar todos los elementos de la fila, puede ocurrir que queden columnas vacías, debido a que sólo tenían un elemento, y éste estaba situado en la fila eliminada. Por lo tanto, habrá que realizar un recorrido en la lista de índices de columna, procediendo a borrar de la misma todos aquellos nodos índices que representan columnas vacías, es decir, aquellos cuyos campos *Primero* o *Ultimo* contengan el valor nulo.

En el Anexo A.10. se desarrolla en pseudocódigo el algoritmo correspondiente.

2.3.11. Eliminación de una columna

Este proceso consiste en localizar el índice a eliminar en la lista de índices de columna y proceder a su borrado, suprimiendo también todos los elementos que se encuentran en dicha columna, reorganizando las listas de las filas de dichos elementos, y reajustando si procede la lista de índices de fila.

Como ya se ha comentado en el algoritmo anterior, la eliminación de un índice es un proceso no implementable en un array estático, puesto que en éste el tamaño es siempre fijo en tiempo de ejecución, al igual que sus conjuntos de índices. No es posible ni añadir, ni modificar, ni borrar un índice.

Sería posible borrar todos los elementos de una columna simplemente sustituyendo su información por la marca de posición vacía, pero no se podría recuperar el espacio físico que ocupaban en memoria, debido a su naturaleza estática.

El ***ABD*** posibilita el reajuste de su dimensionamiento; los conjuntos de índices pueden modificarse de forma dinámica en tiempo de ejecución, por lo que será posible no sólo borrar, sino también insertar (visto en el algoritmo de inserción de un elemento) y modificar los índices. Además el borrado de los elementos incluidos en la columna a eliminar implica la recuperación del espacio físico que ocupaban.

El algoritmo comenzará realizando una búsqueda en la lista de índices de columna del índice a borrar *(J)*, para lo cual se procesará dicha lista de forma secuencial desde el primer elemento, que será el apuntado por la variable *ListaIC*. Para cada nodo se comprobará si su campo de información (*InfoIndice*) coincide con el índice buscado *(J)*. Para pasar de un índice al siguiente se hará en base al campo *SigIndice*.

Este recorrido finalizará cuando se produzca alguna de las situaciones siguientes:

a) El campo de información del nodo índice tratado coincide con *(J)*. Búsqueda con éxito.

b) El campo de información del nodo índice tratado es superior a *(J)*. Puesto que la lista de índices es una lista enlazada ordenada, se puede determinar que el índice buscado *(J)* no existe en la lista (todos los índices que restan de tratar serán superiores). Búsqueda sin éxito.

c) Se llega al final de la lista de índices. Búsqueda sin éxito.

En los casos b y c, se indicará mediante un código de error la no existencia del índice de columna a borrar, dando por finalizado el algoritmo.

Si el índice existe, se puede proceder a eliminarlo, salvando previamente la dirección del primer nodo de la lista de elementos de dicha columna. Dicha dirección estará almacenada en el campo *Primero*.

Para eliminar el nodo índice, además de destruirlo de memoria, será necesario asignar la dirección de su índice sucesor a la variable *ListaIC* si el índice borrado era el menor de la lista de índices, o en caso contrario, al campo *SigIndice* del índice predecesor al borrado.

A continuación será necesario eliminar todos los elementos de la columna en cuestión, reajustando para cada uno de ellos la lista de elementos de la fila en la que están situados.

Este proceso comenzará a partir del primer elemento de la columna, cuya dirección se ha salvado previamente. Se avanzará al siguiente elemento en base al campo puntero *SigColumna* y finalizará el recorrido tras procesar el último elemento de la columna, que será aquel que tenga el valor nulo en su campo *SigColumna*.

Cada elemento tratado se destruirá de memoria, y se actualizarán punteros en sus nodos predecesor y sucesor de fila, de la forma siguiente:

a) *El campo de sucesor de fila del nodo antecesor de fila (AntFila^.SigFila).*

A dicho campo se le asignará la dirección del nodo sucesor de fila al que se va a eliminar, salvo en el caso excepcional de que no exista el nodo antecesor. Esta circunstancia se producirá cuando el elemento borrado sea el de menor índice de columna de los de su fila, en cuyo caso se asignará la dirección del nodo sucesor al campo *Primero* del nodo índice de fila, ya que dicho nodo sucesor pasa a ser el primero de su fila.

b) *El campo de antecesor de fila del nodo sucesor de fila (SigFila^.AntFila).*

A dicho campo se le asignará la dirección del nodo predecesor de fila al que se va a eliminar, salvo en el caso excepcional de que no exista el nodo sucesor. Esta circunstancia se producirá cuando el elemento borrado sea el de mayor índice de columna de los de su fila, en cuyo caso se asignará la dirección del nodo antecesor al campo *Ultimo* del nodo índice de fila, ya que dicho nodo antecesor pasa a ser el último de su fila.

Tras eliminar todos los elementos de la columna, puede ocurrir que queden filas vacías, debido a que sólo tenían un elemento, y éste estaba situado en la columna eliminada. Por lo tanto, habrá que realizar un recorrido en la lista de índices de fila, procediendo a borrar de la misma todos aquellos nodos índices que representan filas vacías, es decir, aquellos cuyos campos *Primero* o *Ultimo* contengan el valor nulo.

En el Anexo A.11. se desarrolla en pseudocódigo el algoritmo correspondiente.

2.3.12. Modificación de un índice de fila

Este proceso consiste en reemplazar el valor de un índice (IA) del conjunto de índices de fila por otro valor proporcionado (IN).

Evidentemente, para poder aplicar este tratamiento será necesario que en principio se cumplan dos premisas:

1) Que exista el índice a reemplazar (IA).
2) Que no exista el nuevo índice (IN).

La modificación del valor de un índice no se puede implementar en arrays estáticos, en los cuales no puede alterarse en ejecución la composición de los conjuntos de índices. Es por tanto, una operación propia y exclusiva del ***ABD*** propuesto.

El algoritmo comenzará comprobando las dos premisas expuestas.

Para la comprobación de la primera (existencia del índice IA) se debe realizar un recorrido de la lista de índices de fila desde el comienzo (*ListaIF*), hasta que se cumpla alguna de las condiciones siguientes:

a) El valor del índice procesado coincide con IA. Se cumple la premisa 1.

b) El valor del índice procesado es superior a IA. Puesto que la lista de índices de fila es una lista enlazada ordenada en creciente, significa que no existe ningún índice de valor IA. No se cumple la premisa 1.

c) Se alcanza el final de la lista de índices de fila. Significa que el índice buscado IA es superior a todos los índices del conjunto de índices de fila, y que por tanto, no existe. No se cumple la premisa 1.

Para la comprobación de la segunda premisa (no existencia del índice IN) se debe realizar también un recorrido de la lista de índices de fila desde el comienzo (*ListaIF*), hasta que se cumpla alguna de las condiciones siguientes:

a) El valor del índice procesado coincide con IN. No se cumple la premisa 2.

b) El valor del índice procesado es superior a IN. Puesto que la lista de índices de fila es una lista enlazada ordenada en creciente, significa que no existe ningún índice de valor IN. Se cumple la premisa 2.

c) Se alcanza el final de la lista de índices de fila. Significa que el índice buscado IN es superior a todos los índices del conjunto de índices de fila, y que por tanto, no existe. Se cumple la premisa 2.

En el caso de que no se cumpla la premisa 1 se devolverá un código de error indicativo de que no existe el índice a modificar, y en caso de que no se cumpla la premisa 2 se devolverá también un código de error indicativo de que ya existe un índice con el nuevo valor. En ambos casos se dará por finalizado el algoritmo.

Si las dos premisas se cumplen el procedimiento a seguir dependerá de la posición que debe ocupar el nuevo índice en la lista de índices de fila:

Caso A. *La posición del índice nuevo es la misma que la que ocupaba el índice antiguo.*

Esta situación se producirá cuando se cumpla alguna de las siguientes condiciones:

a) IA es el único índice de la lista de índices de fila.

b) IA es el primer índice de la lista de índices de fila y IN es menor que el índice sucesor a IA.

c) IA es el último índice de la lista de índices de fila y IN es mayor que el índice predecesor a IA.

d) IA es un índice intermedio (ni primero ni último) en la lista de índices de fila, y IN es mayor que el índice predecesor a IA y menor que el índice sucesor a IA.

En este caso bastará con reemplazar en el campo de información (*InfoIndice*) el valor IA por IN, en el nodo Indice cuyo campo de información tiene el valor IA.

A partir de ese momento todos los elementos de la fila referenciada anteriormente por IA, pasan a estar referenciados ahora por IN, sin necesidad de realizar ningún cambio en dichos elementos.

El hecho de no tener que modificar la lista de elementos de la fila es debido a que en dichos nodos el dato que se almacena en su campo *IndFila* para registrar el índice de fila a la que pertenecen, es la dirección del nodo Indice y no su campo de información.

Al no modificarse la dirección del nodo índice no es necesario modificar los nodos elementos de dicha fila. Por el contrario, si se hubiese almacenado en el campo *IndFila* el dato de información del índice, al haberse modificado éste, hubiese sido necesario modificar dicho dato en todos los nodos de la fila.

Caso B. *La posición del índice nuevo es distinta a la que ocupaba el índice antiguo.*

En el caso de que no se cumpla ninguna de las situaciones expuestas en el punto anterior, la posición que debe ocupar el índice nuevo (IN) en la lista de índices de fila es distinta a la que ocupaba el índice antiguo (IA), por lo que además de reemplazar el valor IA por IN, será necesario reajustar los enlaces entre los nodos Indices para que la lista de índices siga manteniendo el orden creciente entre sus componentes.

Además como consecuencia del cambio de valor y posición del nodo índice de fila, los elementos pertenecientes a dicha fila pueden sufrir cambios en la columna que ocupan, por lo cual será necesario reajustar las columnas en las que se encuentran los elementos de la fila.

Para el proceso de reajuste de la lista de índices de fila se deben realizar las siguientes tareas:

a) Modificar el campo de información del nodo índice que tiene en dicho campo el valor IA por el valor IN.

b) Desenganchar el nodo de la posición que ocupa en la lista de índices de fila.

Para ello habrá que comprobar si tiene nodo predecesor. En caso afirmativo se deberá cargar en el campo *SigIndice* de dicho nodo predecesor, la dirección del nodo sucesor al que vamos a desenganchar. Si por el contrario, no existe tal nodo predecesor (caso de que el índice antiguo fuese el menor del conjunto de índices), se cargará la variable *ListaIF* con la dirección del nodo sucesor al que se va a desenganchar.

c) Enganchar el nodo en la posición correcta de la lista de índices de fila.

El proceso previo realizado para la búsqueda del índice IN devuelve la dirección del nodo índice que debe preceder al IN.

Si dicha dirección contiene el valor nulo significa que el nuevo índice va a ser el primero de la lista de índices de fila. Por lo tanto se cargará en su campo *SigIndice* la dirección del que va a ser su sucesor, que será *ListaIF* (la dirección del que hasta estos momentos es el primero). La variable *ListaIF* se cargará con la dirección del nuevo índice.

Por el contrario, si el nuevo índice tiene nodo predecesor, se cargará su campo *SigIndice* con el campo *SigIndice* de dicho nodo predecesor (dirección del sucesor), y a dicho campo se le asignará la dirección del nuevo índice.

Una vez restablecida la lista de índices de fila, será necesario recolocar los elementos de dicha fila, para situarlos en la posición adecuada en sus respectivas columnas.

Puede ocurrir, para cada elemento de la fila, que como consecuencia de haber variado la posición de su nodo índice de fila en la lista de índices de fila (debido a su cambio de valor), la posición que ocupaba en su columna no sea ahora la correcta y haya que proceder a rectificarla, reajustando la lista de columna.

Puesto que es necesario tratar todos los elementos de la fila, se realizará un recorrido por la misma, comenzando por el primer elemento (cuya dirección se encuentra en el campo *Primero* del nodo índice de la fila), y avanzando desde cada elemento al siguiente por medio del campo *SigFila*. El proceso de recorrido finalizará tras tratar el elemento cuyo campo *SigFila* contiene el valor nulo (último elemento de la fila).

Para cada elemento de la fila el tratamiento a realizar será el siguiente:

a) Desenganchar el elemento de su posición actual en su columna.

Puesto que las listas de elementos son dobles, habrá que modificar los campos *SigColumna* y *AntColumna* de sus nodos predecesor y sucesor de columna respectivamente, o bien los campos *Primero* y *Ultimo* del nodo índice de columna si dichos nodos no existen.

Por tanto, primero se habrá de comprobar si existe o no el nodo antecesor de columna, para lo cual bastará con evaluar el campo *AntColumna* del nodo.

Si tiene el valor nulo significa que no existe tal nodo predecesor (el elemento a desenganchar es el primero de su columna) y en este caso al campo *Primero* del nodo índice de columna se le asignará la dirección del sucesor al nodo a desenganchar, cuya dirección se encuentra en el campo *SigColumna* de dicho nodo.

Si el campo *AntColumna* posee un valor distinto al nulo, significa que existe el nodo antecesor, por lo que al campo *SigColumna* de dicho nodo antecesor se asignará la dirección del sucesor al nodo a desenganchar, cuya dirección se encuentra en el campo *SigColumna* de dicho nodo.

De la misma forma se procede para comprobar si existe o no el nodo sucesor de columna, para lo cual bastará con evaluar el campo *SigColumna* del nodo.

Si tiene el valor nulo significa que no existe tal nodo sucesor (el elemento a desenganchar es el último de su columna) y en este caso al campo *Ultimo* del nodo índice de columna se le asignará la dirección del antecesor al nodo a desenganchar, cuya dirección se encuentra en el campo *AntColumna* de dicho nodo.

Si el campo *SigCclumna* posee un valor distinto al nulo, significa que existe el nodo sucesor, por lo que al campo *AntColumna* de dicho nodo sucesor se asignará la dirección del antecesor al nodo a desenganchar, cuya dirección se encuentra en el campo *AntColumna* de dicho nodo.

b) Localizar la nueva posición que debe ocupar el elemento en su columna.

Para ello se realizará un recorrido desde el primer nodo de la columna (cuya dirección se encuentra en el campo *Primero* del nodo índice de columna), avanzando desde cada nodo al siguiente por medio del campo *SigColumna*.

Este proceso finalizará cuando se cumpla alguna de las dos condiciones siguientes:

- Se alcanza un nodo cuyo campo *IndFila* apunta a un nodo índice cuyo campo de información sea superior a IN.

- Se alcanza el final de la columna. Esto significa que el nodo pasa a ser el último de su columna.

El proceso proporcionará la dirección del nodo predecesor, la cual se utilizará en el paso siguiente.

c) Enganchar el elemento en la nueva posición de su columna.

Para situar el elemento en su nueva posición se deben actualizar sus campos *AntColumna* y *SigColumna* con las direcciones de sus nuevos nodos predecesor y sucesor de columna respectivamente.

Asimismo se deberá asignar la dirección del nodo al campo *SigColumna* de su nodo antecesor (si existe) o al campo *Primero* del nodo índice de columna (si no existe el nodo antecesor, es decir, si el elemento queda como primero de su columna).

La dirección del nodo deberá ser asignada también al campo *AntColumna* de su nodo sucesor (si existe) o al campo *Ultimo* del nodo índice de columna (si no existe el nodo sucesor, lo cual ocurrirá si el nodo queda como último de su columna).

También puede darse el caso de que aún variando su nodo índice de fila, el elemento no varíe su posición en su columna. Esto ocurrirá cuando no existan elementos en la columna para los índices comprendidos entre IA y IN. Sin embargo es preferible desenganchar y enganchar el elemento en la misma posición, que evaluar la situación anterior para los diferentes elementos de la fila.

Como se ha podido comprobar sólo se modifican los campos enlace de columna del elemento y no los de fila, puesto que la posición de cada elemento en la fila no se altera por el hecho de modificarse el valor y posición del índice de fila.

Asimismo tampoco se modifican los campos *IndFila* y *IndColumna* del elemento, puesto que en ellos se almacenan las direcciones de sus nodos índices de fila y columna respectivamente, las cuales no han variado. Si en dichos campos se almacenaran los valores de los índices en vez de sus direcciones, hubiese sido necesario modificar el campo *IndFila* de todos los elementos de la fila, ya que ha variado el valor del índice de fila.

En el Anexo A.12. se desarrolla en pseudocódigo el algoritmo correspondiente.

2.3.13. Modificación de un índice de columna

Este proceso consiste en reemplazar el valor de un índice (JA) del conjunto de índices de columna por otro valor proporcionado (JN).

Evidentemente, para poder aplicar este tratamiento será necesario que en principio se cumplan dos premisas:

1) Que exista el índice a reemplazar (JA).
2) Que no exista el nuevo índice (JN).

La modificación del valor de un índice no se puede implementar en arrays estáticos, en los cuales no es posible alterar en ejecución la composición de los conjuntos de índices. Es por tanto, una operación propia y exclusiva del ***ABD*** propuesto.

El algoritmo comenzará comprobando las dos premisas expuestas.

Para la comprobación de la primera (existencia del índice JA) se debe realizar un recorrido de la lista de índices de columna desde el comienzo (*ListaIC*), hasta que se cumpla alguna de las condiciones siguientes:

a) El valor del índice procesado coincide con JA Se cumple la premisa 1.

b) El valor del índice procesado es superior a JA. Puesto que la lista de índices de columna es una lista enlazada ordenada en creciente, significa que no existe ningún índice de valor JA. No se cumple la premisa 1.

c) Se alcanza el final de la lista de índices de columna. Significa que el índice buscado JA es superior a todos los índices del conjunto de índices de columna, y que por tanto, no existe. No se cumple la premisa 1.

Para la comprobación de la segunda premisa (no existencia del índice JN) se debe realizar también un recorrido de la lista de índices de columna desde el comienzo (*ListaIC*), hasta que se cumpla alguna de las condiciones siguientes:

a) El valor del índice procesado coincide con JN. No se cumple la premisa 2.

b) El valor del índice procesado es superior a JN. Puesto que la lista de índices de columna es una lista enlazada ordenada en creciente, significa que no existe ningún índice de valor JN. Se cumple la premisa 2.

c) Se alcanza el final de la lista de índices de columna. Significa que el índice buscado JN es superior a todos los índices del conjunto de índices de columna, y que por tanto, no existe. Se cumple la premisa 2.

En el caso de que no se cumpla la premisa 1 se devolverá un código de error indicativo de que no existe el índice a modificar, y en caso de que no se cumpla la premisa 2 se devolverá también un código de error indicativo de que ya existe un índice con el nuevo valor. En ambos casos se dará por finalizado el algoritmo.

Si las dos premisas se cumplen el procedimiento a seguir dependerá de la posición que debe ocupar el nuevo índice en la lista de índices de columna:

Caso A. *La posición del índice nuevo es la misma que la que ocupaba el índice antiguo.*

Esta situación se producirá cuando se cumpla alguna de las siguientes condiciones:

a) JA es el único índice de la lista de índices de columna.

b) JA es el primer índice de la lista de índices de columna y JN es menor que el índice sucesor a JA.

c) JA es el último índice de la lista de índices de columna y JN es mayor que el índice predecesor a JA.

d) JA es un índice intermedio (ni primero ni último) en la lista de índices de columna, y JN es mayor que el índice predecesor a JA y menor que el índice sucesor a JA.

En este caso bastará con reemplazar en el campo de información (*InfoIndice*) el valor JA por JN, en el nodo índice cuyo campo de información tiene el valor JA.

A partir de ese momento todos los elementos de la columna referenciada anteriormente por JA, pasan a estar referenciados ahora por JN, sin necesidad de realizar ningún cambio en dichos elementos.

El hecho de no tener que modificar la lista de elementos de la columna es debido a que en dichos nodos el dato que se almacena en su campo *IndColumna* para registrar el índice de columna a la que pertenecen, es la dirección del nodo índice y no su campo de información.

Al no modificarse la dirección del nodo índice no es necesario modificar los nodos elementos de dicha columna. Por el contrario, si se hubiese almacenado en el campo *IndColumna* el dato de información del índice, al haberse modificado éste, hubiese sido necesario modificar dicho dato en todos los nodos de la columna.

Caso B. *La posición del índice nuevo es distinta a la que ocupaba el índice antiguo.*

En el caso de que no se cumpla ninguna de las situaciones expuestas en el punto anterior, la posición que debe ocupar el índice nuevo (JN) en la lista de índices de columna es distinta a la que ocupaba el índice antiguo (JA), por lo que además de reemplazar el valor JA por JN, será necesario reajustar los enlaces entre los nodos índices para que la lista de índices siga manteniendo el orden creciente entre sus componentes.

Además como consecuencia del cambio de valor y posición del nodo índice de columna, los elementos pertenecientes a dicha columna pueden sufrir cambios en la fila que ocupan, por lo cual será necesario reajustar las filas en las que se encuentran los elementos de la columna.

Para el proceso de reajuste de la lista de índices de columna se deben realizar las siguientes tareas:

a) Modificar el campo de información del nodo Índice que tiene en dicho campo el valor JA por el valor JN.

b) Desenganchar el nodo de la posición que ocupa en la lista de Índices de columna.

Para ello habrá que comprobar si tiene nodo predecesor. En caso afirmativo se deberá cargar en el campo *SigIndice* de dicho nodo predecesor, la dirección del nodo sucesor al que vamos a desenganchar. Si por el contrario, no existe tal nodo predecesor (caso de que el Índice antiguo fuese el menor del conjunto de índices), se cargará la variable *ListaIC* con la dirección del nodo sucesor al que se va a desenganchar.

c) Enganchar el nodo en la posición correcta de la lista de índices de columna.

El proceso previo realizado para la búsqueda del Índice JN devuelve la dirección del nodo índice que debe preceder al JN.

Si dicha dirección contiene el valor nulo significa que el nuevo Índice va a ser el primero de la lista de índices de columna. Por lo tanto se cargará en su campo *SigIndice* la dirección del que va a ser su sucesor, que será *ListaIC* (la dirección del que hasta estos momentos es el primero). La variable *ListaIC* se cargará con la dirección del nuevo Índice.

Por el contrario, si el nuevo índice tiene nodo predecesor, se cargará su campo *SigIndice* con el campo *SigIndice* de dicho nodo predecesor (dirección del sucesor), y a dicho campo se le asignará la dirección del nuevo índice.

Una vez restablecida la lista de índices de columna, será necesario recolocar los elementos de dicha columna, para situarlos en la posición adecuada en sus respectivas filas.

Puede ocurrir, para cada elemento de la columna, que como consecuencia de haber variado la posición de su nodo índice de columna en la lista de Índices de columna (debido a su cambio de valor), la posición que ocupaba en su fila no sea ahora la correcta y haya que proceder a rectificarla, reajustando la lista de fila.

Puesto que es necesario tratar todos los elementos de la columna, se realizará un recorrido por la misma, comenzando por el primer elemento (cuya dirección se encuentra en el campo *Primero* del nodo índice de la columna), y avanzando desde cada elemento al siguiente por medio del campo *SigColumna*. El proceso de recorrido finalizará tras tratar el elemento cuyo campo *SigColumna* contiene el valor nulo (último elemento de la columna).

Para cada elemento de la columna el tratamiento a realizar será el siguiente:

a) Desenganchar el elemento de su posición actual en su fila.

Puesto que las listas de elementos son dobles, habrá que modificar los campos *SigFila* y *AntFila* de sus nodos predecesor y sucesor de fila respectivamente, o bien los campos *Primero* y *Ultimo* del nodo índice de fila, si dichos nodos no existen.

Por tanto, primero se habrá de comprobar si existe o no el nodo antecesor de fila, para lo cual bastará con evaluar el campo *AntFila* del nodo.

Si tiene el valor nulo significa que no existe tal nodo predecesor (el elemento a desenganchar es el primero de su fila) y en este caso al campo *Primero* del nodo índice de fila se le asignará la dirección del sucesor al nodo a desenganchar, cuya dirección se encuentra en el campo *SigFila* de dicho nodo.

Si el campo *AntFila* posee un valor distinto al nulo, significa que existe el nodo antecesor, por lo que al campo *SigFila* de dicho nodo antecesor se asignará la dirección del sucesor al nodo a desenganchar, cuya dirección se encuentra en el campo *SigFila* de dicho nodo.

De la misma forma se procede para comprobar si existe o no el nodo sucesor de fila, para lo cual bastará con evaluar el campo *SigFila* del nodo.

Si tiene el valor nulo significa que no existe tal nodo sucesor (el elemento a desenganchar es el último de su fila) y en este caso al campo *Ultimo* del nodo índice de fila se le asignará la dirección del antecesor al nodo a desenganchar, cuya dirección se encuentra en el campo *AntFila* de dicho nodo.

Si el campo *SigFila* posee un valor distinto al nulo, significa que existe el nodo sucesor, por lo que al campo *AntFila* de dicho nodo sucesor se asignará la dirección del antecesor al nodo a desenganchar, cuya dirección se encuentra en el campo *AntFila* de dicho nodo.

b) Localizar la nueva posición que debe ocupar el elemento en su fila.

Para ello se realizará un recorrido desde el primer nodo de la fila (cuya dirección se encuentra en el campo *Primero* del nodo Indice de fila), avanzando desde cada nodo al siguiente por medio del campo *SigFila*.

Este proceso finalizará cuando se cumpla alguna de las dos condiciones siguientes:

- Se alcanza un nodo cuyo campo *IndColumna* apunta a un nodo índice cuyo campo de información sea superior a JN.

- Se alcanza el final de la fila. Esto significa que el nodo pasa a ser el último de su fila.

El proceso proporcionará la dirección del nodo predecesor, la cual se utilizará en el paso siguiente.

c) Enganchar el elemento en la nueva posición de su fila.

Para situar el elemento en su nueva posición se deben actualizar sus campos *AntFila* y *SigFila* con las direcciones de sus nuevos nodos predecesor y sucesor de fila respectivamente.

Asimismo se deberá asignar la dirección del nodo al campo *SigFila* de su nodo antecesor (si existe) o al campo *Primero* del nodo Indice de fila (si no existe el nodo antecesor, es decir, si el elemento queda como primero de su fila).

La dirección del nodo deberá ser asignada también al campo *AntFila* de su nodo sucesor (si existe) o al campo *Ultimo* del nodo índice de fila (si no existe el nodo sucesor, lo cual ocurrirá si el elemento queda como último de su fila).

También puede darse el caso de que aún variando su nodo índice de columna, el elemento no varíe su posición en su fila. Esto ocurrirá cuando no existan nodos en la fila para los índices comprendidos entre JA y JN Sin embargo es preferible desenganchar y enganchar el elemento en la misma posición, que evaluar la situación anterior para los diferentes elementos de la columna.

Como se ha podido comprobar sólo se modifican los campos enlace de fila del elemento y no los de columna, puesto que la posición de cada elemento en la columna no se altera por el hecho de modificarse el valor y posición del índice de columna.

Asimismo tampoco se modifican los campos *IndFila* y *IndColumna* del elemento, puesto que en ellos se almacenan las direcciones de sus nodos índices de fila y columna respectivamente, las cuales no han variado. Si en dichos campos se almacenaran los valores de los índices en vez de sus direcciones, hubiese sido necesario modificar el campo *IndColumna* de todos los elementos de la columna, ya que ha variado el valor del índice de columna.

En el Anexo A.13. se desarrolla en pseudocódigo el algoritmo correspondiente.

3. IMPLEMENTACIÓN DEL ABD MEDIANTE ÁRBOLES BINARIOS DE BÚSQUEDA EQUILIBRADOS AVL

3.1. ÁRBOLES BINARIOS. DEFINICIÓN Y TERMINOLOGÍA.

Un **árbol binario** T se define como un conjunto finito de elementos, llamados nodos, de forma que T es vacío (en cuyo caso se llama árbol nulo o *árbol vacío*), o bien, T contiene un nodo diferenciado R, llamado *raíz* de T, y los restantes nodos de T forman un par ordenado de árboles binarios disjuntos T1 y T2 [LIPS87] [WIR76] [WIR87].

Si T contiene una raíz R, los dos árboles T1 y T2 se llaman, respectivamente, *subárboles izquierdo* y *derecho* de la raíz R. Si T1 no es vacío, entonces su raíz se llama *sucesor izquierdo* de R; y análogamente, si T2 no es vacío, su raíz se llama *sucesor derecho* de R.

Si N es un nodo de T con un sucesor izquierdo S1 y un sucesor derecho S2, se dice que N es *padre* de S1 y S2. Asimismo S1 se dice que es el *hijo izquierdo* de N y S2 el *hijo derecho* de N. Cualquier nodo N de un árbol binario T tiene 0, 1 o 2 sucesores. A los nodos sin sucesores se les denomina *nodos terminales* u *hojas*. Cada nodo N tiene un único padre, excepto la raíz que carece de él.

Los árboles binarios se suelen representar de forma jerárquica arborescente, situando en la cima el nodo raíz, y conectando cada nodo con sus sucesores. Al camino que termina en una hoja se le denomina *rama*.

A cada nodo de un árbol binario se le asigna un *número de nivel*, de la forma siguiente. La raíz R tiene asignado el número de nivel 0, y al resto de nodos se les asigna un número de nivel que es mayor en 1 al número de nivel de su padre. Aquellos nodos con el mismo número de nivel se dice que pertenecen a la misma *generación*.

La *profundidad* o *altura* de un árbol T es el número de nodos de la rama más larga de T. Equivale a 1 más que el mayor número de nivel de T.

Un árbol binario puede recorrerse de dos formas: en *anchura* o en *profundidad*.

El recorrido en *anchura* consiste en tratar primero el nodo de nivel 0 (nodo raiz), a continuación los nodos de nivel 1, después los de nivel 2, etc., hasta tratar por último los nodos del último nivel del árbol. Dentro de cada nivel los nodos se procesan de izquierda a derecha.

Existen tres métodos básicos de recorrer en *profundidad* un árbol binario T de raiz R:

a) *Preorden*.

1) Procesar la raiz R.
2) Recorrer el subárbol izquierdo de R en preorden.
3) Recorrer el subárbol derecho de R en preorden.

b) *Inorden*.

1) Recorrer el subárbol izquierdo de R en inorden.
2) Procesar la raiz R.
3) Recorrer el subárbol derecho de R en inorden.

c) *Postorden*.

1) Recorrer el subárbol izquierdo en postorden.
2) Recorrer el subárbol derecho en postorden.
3) Procesar la raiz R.

Cada algoritmo está definido recursivamente, ya que involucra el recorrido de los subárboles en el orden dado.

Cada uno de ellos contiene los mismos tres pasos y siempre el subárbol izquierdo se recorre antes que el subárbol derecho. La diferencia entre los algoritmos es el momento en que se procesa la raiz R. En preorden se procesa antes de recorrer los subárboles; en inorden se procesa entre los recorridos de los subárboles; y en postorden se procesa tras recorrer los subárboles.

Un **árbol binario de búsqueda o árbol binario ordenado** es aquel en el cual cada nodo N de T tiene la siguiente propiedad: *El valor de N es mayor que cualquier valor del subárbol izquierdo de N y es menor que cualquier valor del subárbol derecho de N.*

Esta propiedad garantiza que el recorrido inorden de T proporciona una lista ordenada de los elementos de T.

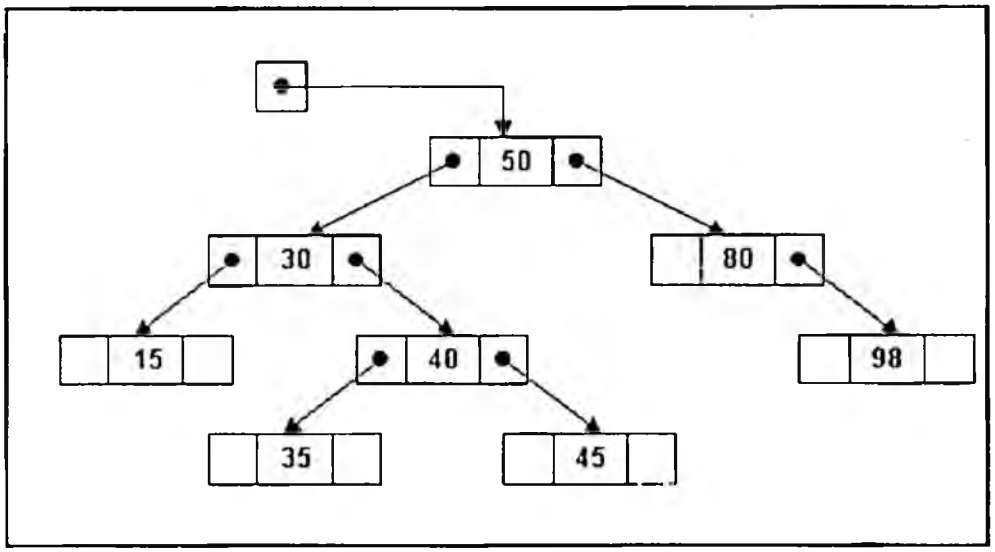


Fig. 3.1.
Árbol Binario de Búsqueda

3.2. ÁRBOL EQUILIBRADO. AVL.

Un árbol está **perfectamente equilibrado** si, para cada nodo, el número de nodos en el subárbol izquierdo, y el número de nodos en el subárbol derecho, difieren como máximo en una unidad.

El número de comparaciones necesarias para encontrar un elemento en un árbol binario de búsqueda perfectamente equilibrado de n nodos es aproximadamente igual a $\log_2 n$ [WIR76] [WIR87].

Ahora bien, cuando se genera un árbol binario de búsqueda, no se conoce a priori cómo va a crecer; no se tiene idea de la forma que va a tomar. Probablemente no será un árbol perfectamente equilibrado; por lo tanto, el número de comparaciones será mayor que $\log_2 n$. La pregunta que surge es ¿en qué medida mayor? ¿merece la pena conservar balanceado un árbol binario de búsqueda tras cada inserción y borrado para optimizar la búsqueda?.

Dicho de otra forma, y suponiendo que los elementos se van a introducir en orden aleatorio, ¿cuántas comparaciones más se necesitarán en promedio en una búsqueda del árbol resultante de las que se requerirían en un árbol totalmente balanceado?.

El caso más desfavorable es fácil de encontrar. Supóngase que todas las claves llegan ya en orden estrictamente ascendente (o descendente); entonces cada clave se coloca inmediatamente a la derecha (o izquierda) de su predecesora, y el árbol resultante degenera completamente, es decir, se convierte en una lista lineal.

En ese caso, el número medio de comparaciones necesarias para localizar un elemento será n/2, siempre y cuando todos los elementos tengan la misma probabilidad de ser buscados. En efecto, el promedio de comparaciones será igual al sumatorio de los productos del número de comparaciones necesarias para localizar cada elemento multiplicado por su probabilidad de ser buscado. Para localizar el primer elemento se necesitará una comparación, 2 para el segundo, 3 para el tercero,..., y n para el n-simo elemento. Al ser equiprobables, la probabilidad de cada elemento será 1/n. Por lo tanto:

Número Medio Comparaciones = $1 * 1/n + 2 * 1/n + ... + n * 1/n =$

$(1 + 2 + ... + n) * 1/n = n / 2 + 1 / 2$

Evidentemente, este caso más desfavorable conduce a un resultado del algoritmo de búsqueda muy pobre, y parece justificar el mantenimiento del equilibrio en el árbol binario de búsqueda. Ahora bien, la comparación será necesario realizarla con el caso medio.

Suponiendo que las n! posibles ordenaciones de los elementos tienen iguales probabilidades de construir el árbol, se demuestra que :

El número promedio de comparaciones que se necesitan en el árbol de búsqueda binaria promedio con n nodos es aproximadamente 2 Ln = (2 L2)(log$_2$n) (siendo L el logaritmo neperiano) [KRU88].

Por lo tanto, el árbol de búsqueda binaria promedio requiere aproximadamente 2 L2 = 1.386 veces más comparaciones que un árbol perfectamente equilibrado. Por lo tanto, el costo ordinario de no balancear un árbol de búsqueda es de aproximadamente 39% más comparaciones [COL78].

El cociente entre las frecuencias de acceso o recuperación de los nodos y de inserción de los mismos, afectará significativamente al umbral de beneficio obtenido con este trabajo de balanceo. Cuanto mayor es este cociente, mayor es el rendimiento del procedimiento reorganizador. Por tanto convendrá evaluar con cierto rigor ese cociente para tomar la decisión de mantener o no el equilibrio del árbol en las inserciones. Se optará por realizarlo si dicho cociente es lo suficientemente grande, o si se teme que se pueda presentar con frecuencia el caso más desfavorable.

Los árboles AVL, definidos en 1962 por G.M. Adelson-Velskii y Landis, cumplen la condición de equilibrio antes propuesta [ADVL62].

Un **árbol AVL** es un árbol de búsqueda binaria en el cual las alturas de los subárboles derecho e izquierdo de la raiz difieren a lo sumo en 1 y en el cual esos subárboles son también árboles AVL. A cada nodo de un árbol AVL se asocia un *factor de balance*, el cual es izquierdo alto (/), igual (=) o derecho alto (\ ),

respectivamente, según que el subárbol izquierdo tenga una altura mayor, igual o menor que la del subárbol derecho [KRU88] [MEHL84].

Los árboles AVL consiguen que las búsquedas, las inserciones y las eliminaciones en un árbol con n nodos se pueden conseguir en un tiempo que es $O(\log_2 n)$ incluso en el peor caso.

Estos resultados son consecuencia directa de un teorema demostrado por Adelson-VelsKii y Landis, según el cual la altura de un árbol AVL con n nodos, nunca excederá de $1.44\log_2 n$, y por ello, aún en el peor caso, su comportamiento no puede estar muy por debajo del de un árbol de búsqueda binaria aleatorio. Con todo, en la generalidad de los casos la longitud concreta de una búsqueda casi es $\log_2 n$, de manera que el comportamiento de los árboles AVL se aproxima mucho al del árbol de búsqueda binaria ideal totalmente balanceado.

Los procesos de inserción y borrado de un nodo en un árbol AVL analizan si tras la actualización se ha vulnerado el criterio de equilibrio, procediendo en tal caso a reestructurar el árbol.

Estudios empíricos realizados por VanDoren y Gray [VAND74], entre otros, han demostrado que tales reorganizaciones locales se requieren para aproximadamente una de cada dos inserciones en el árbol y para cada cuarta eliminación. Los procedimientos AVL garantizan que una sola reorganización requiere como máximo cinco reasignaciones de apuntadores [FOLK92].

Así, el balance en altura usando los métodos AVL garantiza que se obtendrá una aproximación razonable a un desempeño optimo del árbol binario, a un costo que es aceptable en la mayoría de las aplicaciones que usan la memoria primaria de acceso aleatorio (memoria RAM).

A continuación se describen los algoritmos de inserción y borrado de un nodo en un árbol AVL, detallando las rotaciones a aplicar en cada caso para restablecer el equilibrio [BANA91] [KNU87] [KRU88] [WIR76] [WIR87].

3.2.1. Algoritmo de Inserción en AVL

Se puede insertar un nuevo nodo en un árbol AVL utilizando el algoritmo usual del árbol binario de búsqueda, comparando en principio el dato nuevo a insertar con el dato de la raíz. Si coincide se devuelve un mensaje de error por existir ya el elemento; si es menor se desciende al hijo izquierdo de la raíz y si es mayor se desciende a su hijo derecho, realizando la comparación del nuevo dato con el dato del hijo correspondiente.

Se procede de esta forma repetidamente hasta producirse una de las siguientes circunstancias:

a) Se alcanza un nodo cuya información coincide con el nuevo dato a insertar, en cuyo caso se devuelve un código de error indicando la imposibilidad de realizar la inserción por no admitirse elementos repetidos.

b) Se alcanza el final de una rama, siendo el padre del nuevo elemento a insertar el extremo de dicha rama. El nuevo dato se colocará como hijo izquierdo del padre o como hijo derecho, según sea menor o mayor que él respectivamente.

Tras realizar la inserción en el árbol de raiz R y con subárboles izquierdo y derecho I y D, hay que distinguir tres casos. Supóngase que el nuevo nodo se inserta en el subárbol derecho D haciendo que su altura aumente en 1:

a) $H_I = H_D$: I y D tendrán distinta altura, pero el criterio de equilibrio no se vulnera.

b) $H_I > H_D$: I y D igualarán sus alturas, es decir, incluso se mejora el equilibrio.

c) $H_I < H_D$: Se vulnera el criterio de equilibrio y hay que reestructurar el árbol.

Un análisis de la situación revela que sólo hay dos organizaciones del árbol que tienen que considerarse individualmente. El resto de ellas pueden deducirse por simetría a partir de esas dos.

Supongamos, como hemos adelantado, que se ha insertado el nodo en el subárbol derecho del árbol, que su altura ha aumentado, y que el árbol original era derecho alto. Sea r la raíz del subárbol y x la raiz de su subárbol derecho.

a) *Restablecimiento del equilibrio mediante una rotación simple. (x es derecho alto)*

La acción que se necesita en este caso recibe el nombre de *rotación izquierda*. Se gira el nodo x hacia arriba a la raiz, dejando caer r en el subárbol izquierdo de x. El subárbol T_2 de nodos con claves entre las de r y x se convierte ahora en el subárbol derecho de r y no en el subárbol izquierdo de x.

Tras la rotación, la altura del árbol rotado ha disminuido en 1; antes habia aumentado a causa de la inserción; de ahí que al final la altura del árbol permanezca inalterada.

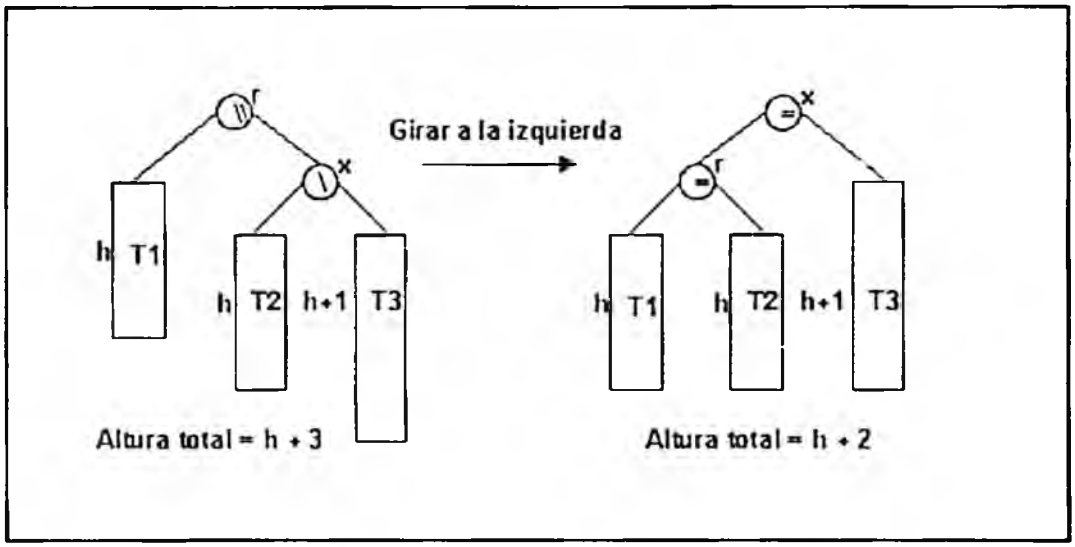


Fig. 3.2.
Inserción en AVL. Rotación Simple

b) *Restablecimiento del equilibrio mediante una doble rotación. (x es izquierdo alto)*

En este caso es preciso mover dos niveles, hacia el nodo w que implanta el subárbol izquierdo de x para encontrar la nueva raiz. Este proceso se denomina rotación doble, porque la transformación se logra en dos pasos. Primero girando

el subárbol de raiz x hacia la derecha (de modo que w se convierta en su raiz) y luego girando el árbol de raiz r hacia la izquierda (moviendo w hacia arriba para que se transforme en la nueva raiz).

Ahora los nuevos factores de balance de r y x dependen del factor anterior de balance de w. El diagrama muestra que los subárboles de w tienen igual altura, pero es posible que w sea izquierdo o derecho alto. Los factores de balance resultantes serán:

w anterior	r nueva	x nueva	
=	=	=	(T2 y T3 altura h)
/	=	\	(T2 h y T3 h-1)
\	/	=	(T2 h-1 y T3 h)

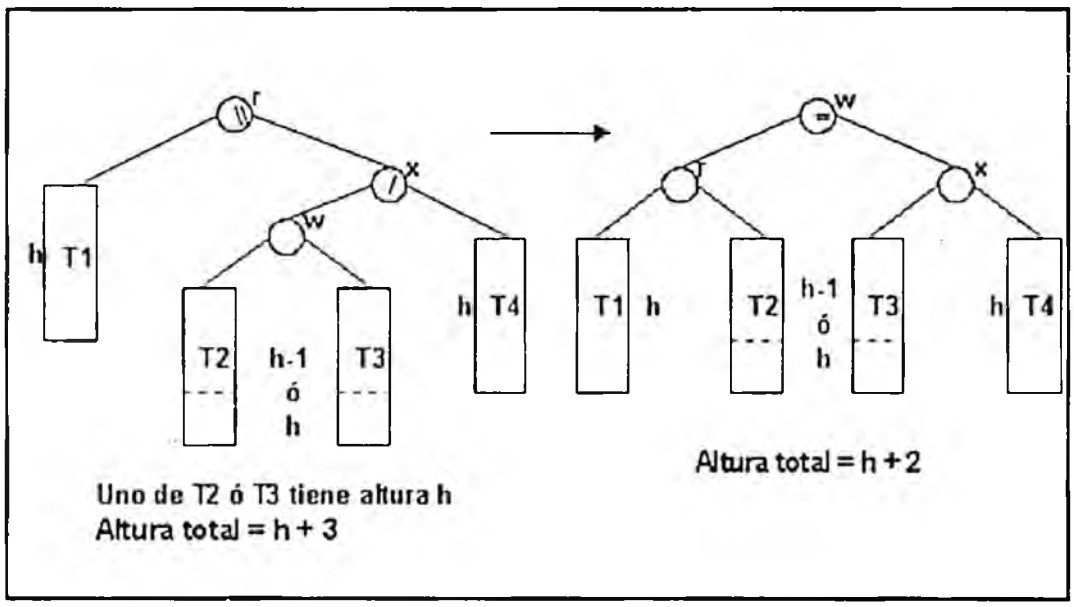


Fig. 3.3.
Inserción en AVL. Rotación Doble

En el Anexo B.14. se desarrolla en pseudocódigo el algoritmo de inserción de un nodo en un AVL, siendo los campos de cada nodo los siguientes:

Info: Información del nodo
Izq, Der: Direcciones de los hijos izquierdo y derecho respectivamente.
FB: Factor de Balance del nodo (-1: izquierdo alto; 0: Igual; 1: derecho alto)

3.2.2. Algoritmo de Borrado en AVL

El proceso de borrado seguirá la siguiente secuencia de pasos:

a) Se reduce el problema al caso en que el nodo x que ha de eliminarse tiene a lo sumo un hijo.

En el caso de que tenga dos, se localiza el predecesor inmediato en inorden z, posicionándose primero en el hijo izquierdo de x y avanzando después por la rama de la derecha hasta llegar a un nodo que no tenga hijo derecho. Este nodo z será el predecesor inmediato de x, puesto que es el mayor de los menores a él.

Se coloca z (o una copia de z) en la posición del árbol ocupada por x (con el mismo padre, los hijos derecho e izquierdo y el factor de balance que tuvo x). A continuación se elimina z de su posición anterior, procediendo según se indica a continuación y utilizando z en lugar de x en cada uno de los pasos siguientes.

b) Se elimina el nodo x en el árbol. Como se sabe (por el paso a) que x tiene a lo sumo un hijo, bastará para suprimir x con sólo ligar el padre de x al único hijo de x (o a nada si x no tiene hijo).

La altura del subárbol antes implantado en x se ha reducido en 1, por lo que será necesario investigar los efectos de este cambio de altura en todos los nodos de la rama que lleva desde x hasta la raiz. Se utilizará para ello una variable booleana. La acción que se tomará en cada nodo dependerá del valor de dicha variable booleana, del factor balance en el nodo y, algunas veces, del de un hijo del nodo.

c) La variable booleana es verdadera al inicio. Los siguientes pasos deben hacerse para cada nodo P que se encuentre en la trayectoria del padre de x hacia la raiz del árbol, siempre y cuando la variable booleana se mantenga con valor verdadero. Cuando se vuelve falsa, no se requerirán ya más cambios y termina el algoritmo.

d) Caso1: el nodo actual P tiene un factor de balance igual. El factor de balance de P ha cambiado según que su subárbol izquierdo o derecho haya sido abreviado y entonces la variable booleana se pone a falso. (altura sin cambio)

e) Caso 2: el factor de balance de P no es igual y el subárbol más alto ha sido acortado. Se cambia el factor de balance de P a igual y se deja como verdadera la variable booleana. (altura reducida)

f) Caso 3: el factor de balance de P no es igual, y el subárbol más corto ha sido acortado. El requerimiento de altura para un árbol AVL deja de cumplirse ahora en P, por lo cual para restaurar el balance se aplica una rotación en la forma mostrada a continuación. Sea Q la raiz del subárbol más alto de P (el que no se ha acortado). Existen tres casos según el factor de balance de Q.

g) Caso 3a: el factor de balance de Q es igual. Una rotación simple (que transforma los factores de balance de P y Q) restablece el balance y la variable booleana se pone a falso. (altura sin cambio)

h) Caso 3b: el factor de balance de Q es el mismo que el de P: Se aplica una rotación simple, se igualan los factores de balance de P y Q y se deja como verdadera la variable booleana. (altura reducida)

i) Caso 3c: los factores de balance de P y Q son contrarios. Se aplica una doble rotación (primero alrededor de Q y luego alrededor de P), se igualan los factores de balance de la nueva raiz y otros según se juzgue apropiado, y se deja como verdadera la variable booleana. (altura reducida)

En el Anexo B.15. se desarrolla en pseudocódigo el algoritmo correspondiente.

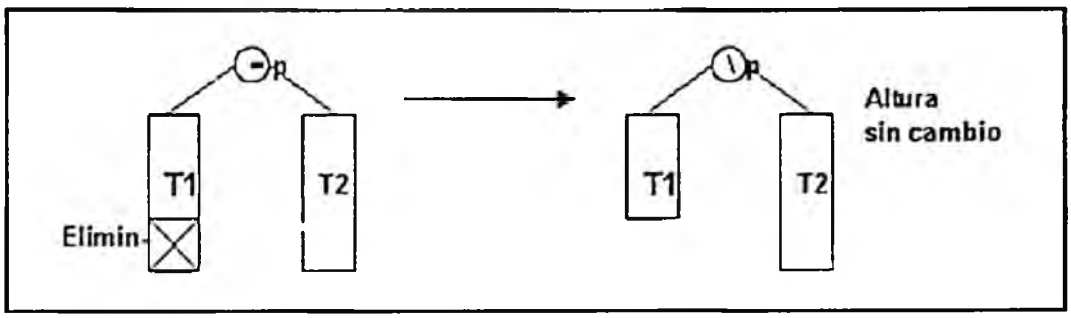


Fig. 3.4.
Borrado en AVL. Caso 1

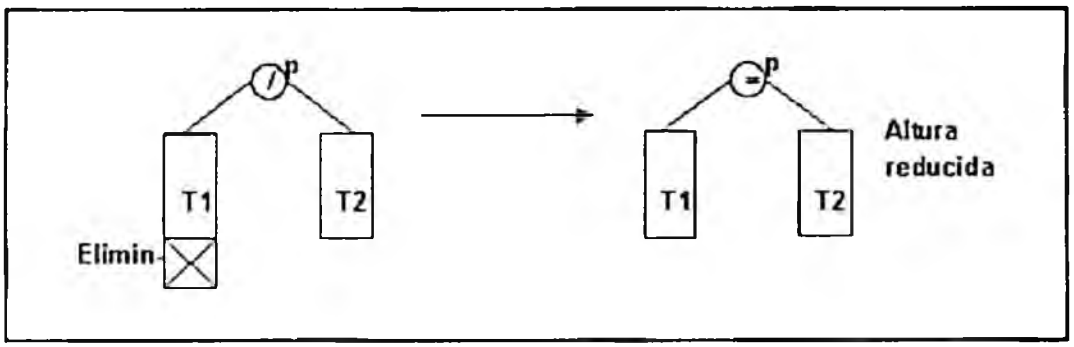


Fig. 3.5.
Borrado en AVL. Caso 2

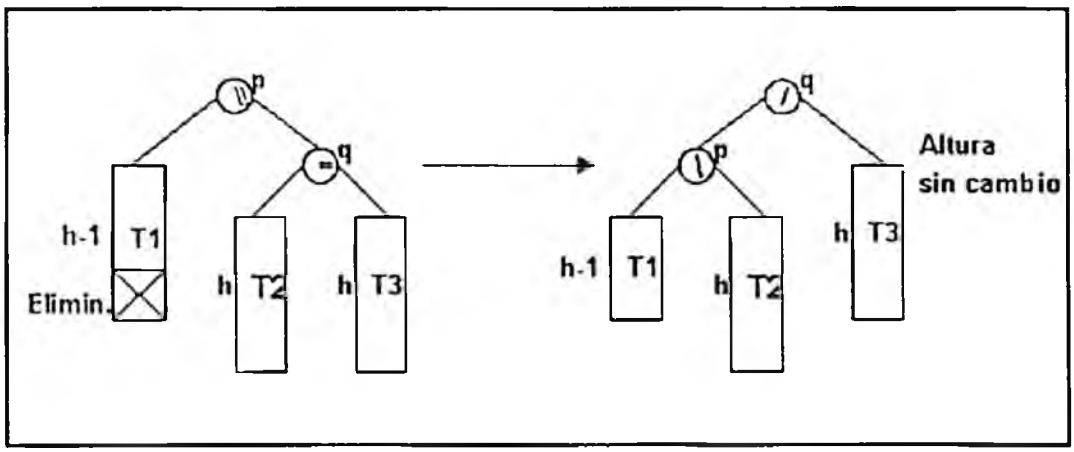


Fig. 3.6.
Borrado en AVL. Caso 3a

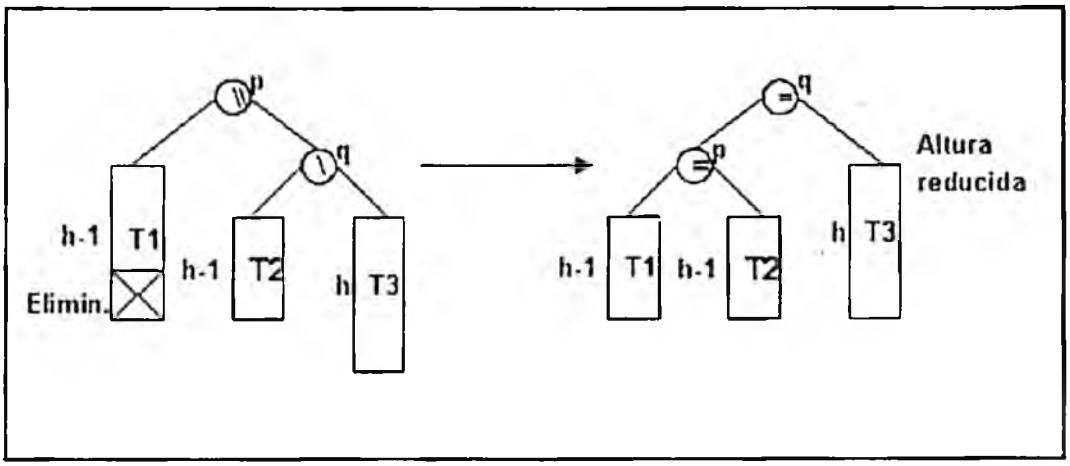


Fig. 3.7.
Borrado en AVL. Caso 3b

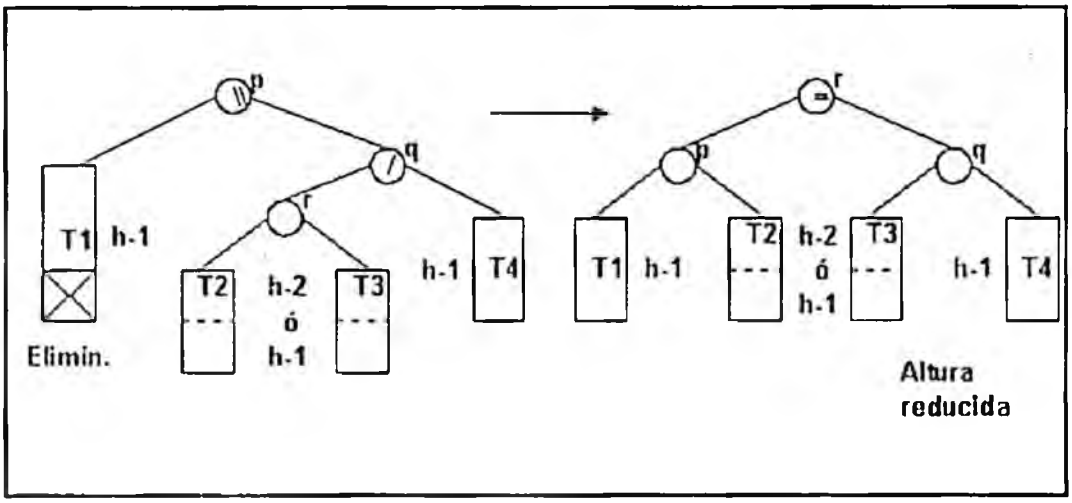


Fig. 3.8.
Borrado en AVL. Caso 3c

3.3. REPLANTEAMIENTO DE LA IMPLEMENTACIÓN DEL ABD

En gran parte de las operaciones de proceso del ***ABD*** descritas anteriormente, aparecen tratamientos de búsqueda secuencial. Estos se aplican tanto en las listas de índices para localizar un determinado índice, como en las listas de elementos para acceder a una determinada posición en base a los índices de fila y columna.

El hecho de que las búsquedas sean secuenciales viene condicionado por la implementación de los índices y de los elementos en base a listas enlazadas.

Con el fin de mejorar los tiempos de búsqueda, y en concordancia con la estructura de los arrays estáticos, las listas enlazadas se construyeron ordenadas:

- Las listas de índices, en base a los propios índices.

- Las listas de elementos que representaban filas, en base a los índices de columna de sus elementos

- Las listas de elementos que representaban columnas, en base a los índices de fila de sus elementos.

La ordenación de las listas de elementos viene impuesta por la ordenación de las listas de índices, y por la estructura mallada elegida. A pesar de la ordenación de estas listas, no es posible aplicar el algoritmo de búsqueda binaria como en arrays estáticos.

La búsqueda binaria en un array lineal ordenado realiza el siguiente tratamiento: accede al elemento mitad del array y compara su contenido con el dato buscado. Si no coinciden, decide en cual de las dos mitades debe hallarse el elemento buscado en caso de existir (izquierda si es menor al elemento mitad y derecha si es mayor), y obtiene el elemento mitad de esa parte del array, repitiendo el proceso de forma iterativa y finalizando bien cuando se encuentra el elemento (búsqueda con éxito) o bien cuando no existe más zona a buscar (búsqueda fallida).

Al realizar cada comparación, elige la mitad de la zona que tiene acotada en ese momento, prescindiendo de la otra mitad. Por ello la complejidad del algoritmo de búsqueda binaria es $O(\log_2 n)$ [BRAS90].

En las listas enlazadas no se tiene acceso inmediato al elemento que ocupa la mitad de la estructura, por lo tanto no es factible aplicar el tratamiento descrito anteriormente. Además la estructura propuesta posee el agravante (ventaja desde el punto de vista de su implementación) de que los conjuntos de índices no tienen que ser series de elementos consecutivos, lo cual invalida la búsqueda binaria. Queda por tanto como único recurso la búsqueda secuencial.

Esta búsqueda secuencial se ve favorecida por el hecho de que las listas enlazadas están ordenadas, y por tanto, en caso de no existir el elemento buscado, no hay necesidad de alcanzar el final de la estructura, sino que se puede detener el proceso en el momento de alcanzar un elemento superior al buscado. En cualquier caso la función de complejidad será O(n).

La solución que se plantea a continuación es implementar las series de índices de filas y columnas, así como las series de elementos de cada fila y columna, en base a árboles binarios de búsqueda equilibrados AVL.

Con esta implementación se conseguirá mejorar los procesos ya citados de búsqueda de índices y elementos, y con ello las operaciones de manejo del ***ABD***, puesto que en los árboles AVL las búsquedas como máximo necesitan $\log_2 n$ comparaciones, siendo n el número de elementos del árbol.

La búsqueda en un árbol binario de búsqueda en general, se basa en el criterio de ordenación del árbol, según el cual cada nodo es mayor que los nodos de su subárbol izquierdo y menor que los nodos de su subárbol derecho. El tratamiento es el siguiente: compara el dato buscado con la raiz; si no coinciden decide en cual de los dos subárboles (izquierdo si es menor o derecho si es mayor) debe encontrarse el elemento en caso de existir, procediendo a comparar el elemento buscado con la raiz del subárbol elegido. El proceso se repite iterativamente hasta encontrar el elemento (búsqueda con éxito) o hasta llegar al final de una rama (búsqueda fallida).

Si el árbol se encuentra equilibrado, el número de comparaciones para localizar un elemento no superará $\log_2 n$, puesto que tras cada comparación se prescinde de una mitad del árbol en que está situado. Se prefiere árboles AVL porque en el caso de no estar balanceado el árbol binario de búsqueda, el número medio de comparaciones aumenta un 39% como ya se ha mencionado anteriormente.

3.4. ESTRUCTURA DEL ABD EN BASE A ÁRBOLES AVL

La implementación del array bidimensional dinámico mediante AVLs estará basada en dos estructuras diferenciadas:

a) Los árboles de índices de fila y columna.

b) Los árboles de elementos de cada fila y columna.

a) Árboles de Índices

Existirán dos árboles, uno para representar el conjunto de índices de fila y otro para representar el conjunto de índices de columna. En cada uno de dichos árboles existirá un nodo por índice con la siguiente composición:

- Un campo de información (*InfoIndice*) que contendrá el valor del índice que representa. Este será el campo clave en base al cual se ordenará el árbol.

- Un campo (*FBIndice*) que contendrá el factor de balance del nodo índice en el árbol AVL.

- Un campo puntero (*HijoIzqIndice*) que contendrá la dirección en el árbol del índice situado como su hijo izquierdo.

- Un campo puntero (*HijoDerIndice*) que contendrá la dirección en el árbol del índice situado como su hijo derecho.

- Un campo puntero (*ArbolElem*) que contendrá la dirección de la raíz del árbol formado por los elementos de la fila o columna que representa el índice.

El campo factor de balance (*FBIndice*) se interpretará como la altura del subárbol derecho menos la altura del subárbol izquierdo, y podrá contener, por consiguiente, los siguientes valores:

-1 *Izquierdo Alto*. El subárbol izquierdo tiene mayor altura que el subárbol derecho.
0 *Igual*. El subárbol izquierdo tiene la misma altura que el subárbol derecho.
1 *Derecho Alto*. El subárbol derecho tiene mayor altura que el subárbol izquierdo.

Cuando de un determinado índice no cuelgue hijo izquierdo o derecho, el campo puntero correspondiente tendrá el valor nulo.

No podrá ocurrir en ningún caso que el campo *ArbolElem* tenga el valor nulo, puesto que eso significaría que la fila o columna referenciada por el índice no contiene ningún elemento, lo cual es imposible debido al carácter dinámico del array, según el cual un índice se crea, si no existe, sólo cuando va a referenciar a un elemento y se elimina cuando se borran todos los elementos de su fila o columna.

Además, existirán dos variables puntero *ArbolIF* y *ArbolIC* que contendrán las direcciones de las raíces de los árboles de índices de fila y de columna respectivamente. Un ***ABD*** vacío se definirá asignando el valor nulo a dichas variables.

b) Árboles de Elementos

Existirá un árbol de elementos por cada fila y por cada columna del ***ABD***. Cada elemento del ***ABD*** vendrá representado por un nodo, el cual estará formando parte simultáneamente de dos árboles: el correspondiente a su fila y el correspondiente a su columna. De esta forma los árboles de filas y columnas se encontrarán entrelazados entre sí. Cada nodo presentará la siguiente estructura:

- Un campo de información (*InfoElemento*) representativo del elemento. Este campo podrá ser un dato de tipo simple o compuesto.

- Un campo (*ClaveArbolF*) que contendrá la dirección del índice de columna en el árbol de índices de columna. El valor de este índice de columna es la clave con la que entra a formar parte el elemento en el árbol de su fila.

- Un campo (*FBArbolF*) que contendrá el factor de balance del nodo en el árbol AVL de su fila.

- Un campo puntero (*HijoIzqArbolF*) que contendrá la dirección de su nodo hijo izquierdo en el árbol de su fila.

- Un campo puntero (*HijoDerArbolF*) que contendrá la dirección de su nodo hijo derecho en el árbol de su fila.

- Un campo (*ClaveArbolC*) que contendrá la dirección del índice de fila en el árbol de índices de fila. El valor de este índice de fila es la clave con la que entra a formar parte el elemento en el árbol de su columna.

- Un campo (*FBArbolC*) que contendrá el factor de balance del nodo en el árbol AVL de su columna.

- Un campo puntero (*HijoIzqArbolC*) que contendrá la dirección de su nodo hijo izquierdo en el árbol de su columna.

- Un campo puntero (*HijoDerArbolC*) que contendrá la dirección de su nodo hijo derecho en el árbol de su columna.

Los árboles representativos de filas estarán ordenados en base a los índices de columna de sus elementos, mientras que los árboles de columnas estarán ordenados en base a los índices de fila de sus elementos.

Las direcciones de las raíces de estos árboles de filas y columnas están registradas, como se ha comentado anteriormente, en los campos *ArbolElem* de los índices correspondientes.

Los campos factor de balance de cada elemento (*FBArbolF)(FBArbolC*) se interpretarán como la diferencia de altura entre el subárbol derecho y el subárbol izquierdo en los árboles de su fila y columna respectivamente, y podrán contener, por consiguiente, los siguientes valores:

-1 *Izquierdo Alto.* El subárbol izquierdo tiene mayor altura que el subárbol derecho.

0 *Igual.* El subárbol izquierdo tiene la misma altura que el subárbol derecho.

1 *Derecho Alto.* El subárbol derecho tiene mayor altura que el subárbol izquierdo.

Los campos puntero de hijos de un nodo contendrán el valor nulo cuando no existan los correspondientes hijos izquierdo o derecho en sus árboles de fila y columna.

Debido a la complejidad de esta implementación, se adjunta la representación de un array sencillo, que conste de los siguientes elementos:

[5, 10] = 80 [23, 20] = 90 [40, 10] = 38 [40, 20] = 25

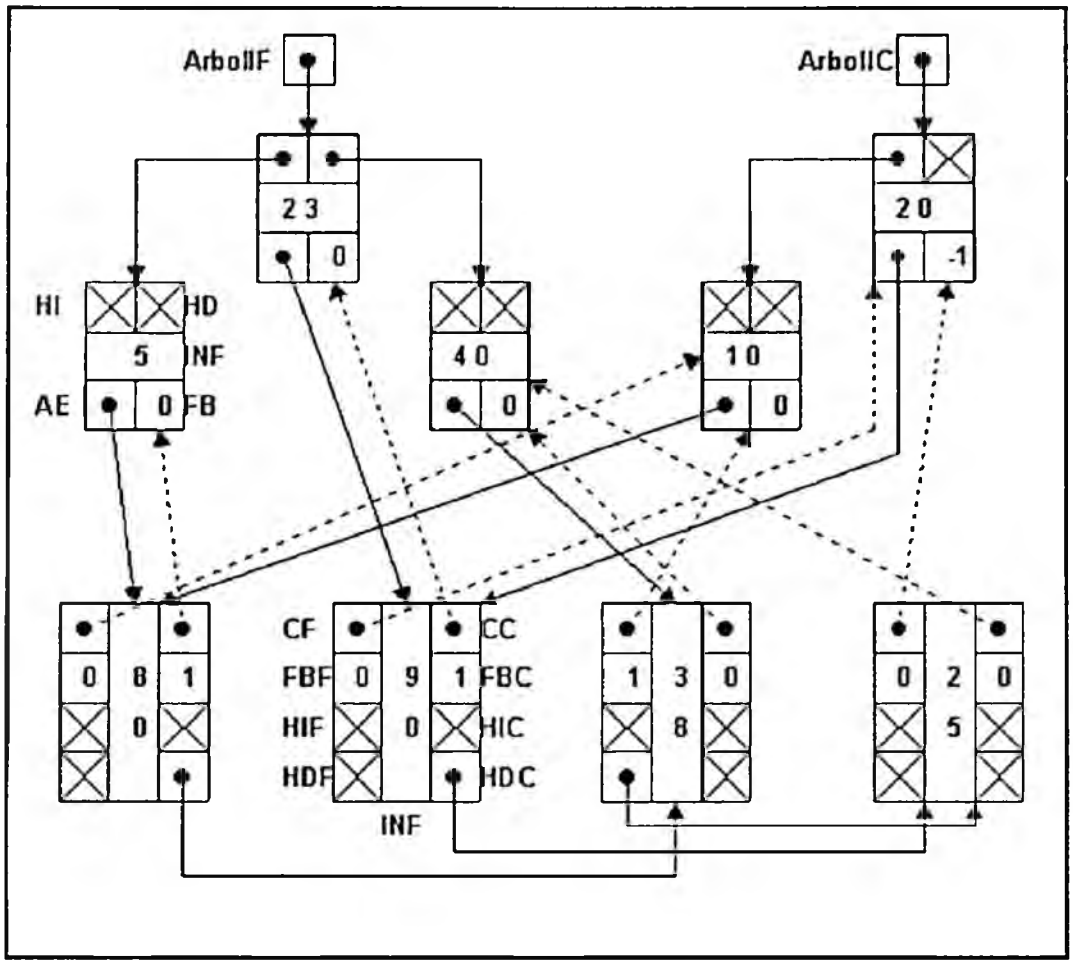


Fig. 3.9.

Array Bidimensional Dinámico implementado mediante Árboles AVL

INF = *InfoIndice*
FB = *FBIndice*
HI = *HijoIzqIndice*
HD = *HijoDerIndice*
AE = *ArbolElem*

INF = *InfoElemento*
CF = *ClaveArbolF*
FBF = *FBArbolF*
HIF = *HijoIzqArbolF*
HDF = *HijoDerArbolF*
CC = *ClaveArbolC*
FBC = *FBArbolC*
HIC = *HijoIzqArbolC*
HDC = *HijoDerArbolC*

3.5. OPERACIONES DE MANIPULACIÓN DEL ABD

Antes de proceder a describir las operaciones en base a la nueva implementación, conviene realizar las siguientes puntualizaciones:

Las operaciones de manipulación, al ser inherentes a la estructura, independientemente de la implementación de ésta, serán las mismas que en el caso de listas. Cambiarán internamente debido a que las estructuras de datos dinámicas sobre las que operan son diferentes, pero esta adaptación no afectará a los programas que hagan uso del TAD, puesto que el interfaz permanece inalterable. Esta propiedad se conoce como encapsulación del tipo abstracto de datos.

En la implementación de las operaciones que se describen a continuación, no se hará referencia en los algoritmos a los factores de balance de los nodos, ni en los árboles de índices ni en los árboles de elementos. Se ha preferido prescindir de dichos datos con el fin de no restar claridad a los procedimientos de inserción y borrado con las rotaciones a aplicar en los casos en que se pierde la condición de equilibrio del árbol AVL.

Dichos tratamientos ya han quedado suficientemente explicados con anterioridad, y en la práctica sería cuestión de incluirlos en los algoritmos de inserción y borrado que se desarrollarán a continuación. Se hará, no obstante, referencia a dicha circunstancia, en todos los casos en que como consecuencia de incluir o eliminar un índice o un elemento, fuera necesario restituir el estado de equilibrio del árbol AVL correspondiente.

Los árboles binarios, al ser estructuras claramente recursivas, se prestan a ser tratados bajo dicho prisma en los algoritmos. Por ello en la implementación de las operaciones se aplicará en algunos casos la recursividad. En otros casos, sin embargo, se preferirán soluciones iterativas. Por último, en otras situaciones se aplican indistintamente técnicas recursivas y/o iterativas.

3.5.1. Recorrido por filas

Al igual que en la implementación mediante listas, el proceso de recorrido por filas del *ABD* implementado mediante árboles binarios de búsqueda AVL, puede construirse en base a dos ciclos anidados: el externo recorrerá el árbol de indices de fila, y realizará, por tanto, tantas iteraciones como filas existan en el array y el interno recorrerá los árboles de elementos de cada fila, siendo el número de iteraciones por fila variable e igual al número de elementos de dicha fila.

La dirección de la raiz del árbol de indices de fila está contenida en la variable *ArbolIF*. El recorrido de este árbol habrá que realizarlo en inorden, puesto que dicho criterio proporciona en un árbol binario de búsqueda, una secuencia ordenada de sus elementos.

Comenzará dicho recorrido inorden con el indice inferior, que estará situado en el extremo de la rama más izquierda del árbol, y terminará con el mayor indice de fila, el cual estará situado en el extremo de la rama más derecha del árbol.

Por cada nodo que se trate del árbol de indices de fila, habrá que recorrer al completo el árbol de elementos de la fila referenciada por dicho indice. La dirección de la raiz de dicho árbol de elementos se encuentra en el campo *ArbolElem* del nodo indice.

El recorrido de cada uno de los árboles de elementos de fila se realizará también inorden, para que los elementos se procesen en el mismo orden en que se encuentran en la fila, es decir, ordenados por columna. Este recorrido inorden en los árboles de elementos se hará en base a los campos *HijoIzqArbolF* y *HijoDerArbolF* de cada nodo.

Los recorridos inorden, tanto en el árbol de indices como en los árboles de elementos, se pueden implementar en base a dos soluciones diferentes:

A) *Solución Recursiva.*

Puesto que un árbol binario es una estructura claramente recursiva, los recorridos se pueden diseñar fácilmente según este principio. En concreto, el recorrido inorden de forma recursiva constará de tres pasos:

a) Llamada recursiva al procedimiento, pasándole la raiz del subárbol izquierdo.

b) Tratamiento de la raiz.

c) Llamada recursiva al procedimiento, pasándole la raiz del subárbol derecho.

Criterio Base: Árbol Vacio (dirección de la raiz igual a valor nulo).

B) ***Solución Iterativa.***

Seria la solución aplicable en el caso de que el lenguaje sobre el que se fuera a implementar la estructura y sus algoritmos, no soportase la recursividad.

El diseño utiliza una estructura de pila, por la cual van pasando todos los nodos antes de ser procesados. En principio, se almacenan las direcciones de los nodos que componen la rama más izquierda del árbol, comenzándose a tratar el extremo de dicha rama. Tras procesar un nodo, se examina si tiene hijo derecho, y en caso afirmativo, se almacenan en pila las direcciones de todos los nodos de la rama más izquierda del subárbol del cual dicho hijo derecho es raiz. Siempre el nodo a tratar es el que ocupa la cima de la pila. El proceso finaliza cuando la pila queda vacia.

En un algoritmo de recorrido, la implementación mediante árboles del ***ABD***, no aporta ninguna ventaja frente a la implementación por medio de listas, puesto que hay que procesar de forma secuencial todos los elementos de la estructura. Las ventajas de la implementación mediante árbol se manifestarán en las operaciones que sea necesario localizar un índice o un elemento.

La ventaja frente al array tradicional estático, es que en el ***ABD*** únicamente se recorren los elementos existentes en la estructura, puesto que no existen "posiciones vacías". Por tanto, como ya se comentó en la implementación por listas, será tanto más ventajoso cuanto mayor sea el número de posiciones no ocupadas del array clásico.

En el Anexo B.1. se desarrollan en pseudocódigo ambos recorridos.

3.5.2. Recorrido por columnas

El recorrido por columnas puede construirse en base a dos ciclos anidados: el externo recorrerá el árbol de índices de columna, y realizará, por tanto, tantas iteraciones como columnas existan en el array y el interno recorrerá los árboles de elementos de cada columna, siendo el número de iteraciones por columna variable e igual al número de elementos de dicha columna.

La dirección de la raíz del árbol de índices de columna está contenida en la variable *ArbolIC*. El recorrido de este árbol habrá que realizarlo en inorden, puesto que dicho criterio proporciona en un árbol binario de búsqueda, una secuencia ordenada de sus elementos.

Comenzará dicho recorrido inorden con el índice inferior de columna, que estará situado en el extremo de la rama más izquierda del árbol, y terminará con el mayor índice de columna, el cual estará situado en el extremo de la rama más derecha del árbol.

Por cada nodo que se trate del árbol de índices de columna, habrá que recorrer al completo el árbol de elementos de la columna referenciada por dicho índice. La dirección de la raíz de dicho árbol de elementos se encuentra en el campo *ArbolElem* del nodo índice.

El recorrido de los árboles de elementos de cada columna se realizará también inorden, para que los elementos se procesen en el mismo orden en que están dispuestos en la columna, es decir, ordenados por fila. Este recorrido inorden en los árboles de elementos se hará en base a los campos *HijoIzqArbolC* y *HijoDerArbolC* de cada nodo.

Los recorridos inorden, tanto en el árbol de índices como en los árboles de elementos, se pueden implementar en base a dos soluciones diferentes:

A) ***Solución Recursiva.***

Puesto que un árbol binario es una estructura claramente recursiva, los recorridos se pueden diseñar fácilmente según este principio. En concreto, el recorrido inorden de forma recursiva constará de tres pasos:

a) Llamada recursiva al procedimiento, pasándole la raiz del subárbol izquierdo.

b) Tratamiento de la raiz.

c) Llamada recursiva al procedimiento, pasándole la raiz del subárbol derecho.

Criterio Base: Árbol Vacio (dirección de la raiz igual a valor nulo).

B) ***Solución Iterativa.***

Seria la solución aplicable en el caso de que el lenguaje sobre el que se fuera a implementar la estructura y sus algoritmos, no soportase la recursividad.

El diseño utiliza una estructura de pila, por la cual van pasando todos los nodos antes de ser procesados. En principio, se almacenan las direcciones de los nodos que componen la rama más izquierda del árbol, comenzándose a tratar el extremo de dicha rama. Tras procesar un nodo, se examina si tiene hijo derecho, y en caso afirmativo, se almacenan en pila las direcciones de todos los nodos de la rama más izquierda del subárbol del cual dicho hijo derecho es raiz. Siempre el nodo a tratar es el que ocupa la cima de la pila. El proceso finaliza cuando la pila queda vacia.

Como ya se ha comentado en el algoritmo de recorrido por filas, en este caso también, la implementación mediante árboles del ***ABD***, no aporta ninguna ventaja frente a la implementación por medio de listas, puesto que hay que procesar de forma secuencial todos los elementos de la estructura, y en principio, no existe apenas diferencia entre recorrer los nodos de una lista o los de un árbol.

La ventaja frente al array tradicional estático, al igual que ocurre en el recorrido por filas, es que en el ***ABD*** únicamente se recorren los elementos existentes en la estructura, puesto que no existen "posiciones vacias". Por tanto, es tanto más ventajoso cuanto mayor es el dimensionamiento del array y menor su número de elementos.

En el Anexo B.2. se desarrollan en pseudocódigo ambos recorridos.

3.5.3. Acceso a un elemento referenciado por sus índices

En la implementación del ***ABD*** en base a listas enlazadas, el proceso de acceso a un elemento por sus índices [I , J] se basaba en tres búsquedas:

a) Búsqueda del índice de fila *(I)*, en la lista de índices de fila.

b) Búsqueda del índice de columna *(J)*, en la lista de índices de columna.

c) Búsqueda del elemento [I , J] en la lista de elementos de la fila *(I)* (primer método), o bien, en la lista de elementos de la columna *(J)* (segundo método).

Las correspondientes listas enlazadas de índices y de elementos están ordenadas según los criterios de búsqueda, con lo cual se facilita este proceso para las búsquedas fallidas, ya que en estos casos no es necesario alcanzar el final de la lista, sino que se puede detener el algoritmo en el momento de tratar un elemento con clave superior a la buscada.

En cualquier caso, la función de complejidad del algoritmo de búsqueda será O(n), es decir, directamente proporcional al número de elementos que intervienen en la búsqueda.

En la implementación del ***ABD*** por medio de AVLs, el acceso a un elemento en base a sus índices, también constará de los tres procesos de búsqueda anteriores:

a) Búsqueda del índice de fila *(I)*, en el árbol de índices de fila.

b) Búsqueda del índice de columna *(J)*, en el árbol de índices de columna.

c) Búsqueda del elemento [I , J] en el árbol de elementos de la fila *(I)* (primer método) , o bien, en el árbol de elementos de la columna *(J)* (segundo método).

Ahora bien, el proceso de búsqueda de un elemento en un árbol binario de búsqueda, tiene una función de complejidad $O(\log_2 n)$. Por este motivo, en la implementación mediante árboles, se apreciará una diferencia en los tiempos de acceso por índices a un elemento, tanto más significativa cuanto mayor sea el número de índices y de elementos del array, frente a la implementación por listas.

El acceso se puede realizar de dos formas:

A) Primer Método

En principio, habrá que localizar el índice de fila *(I)* en el árbol de índices de fila. Para ello se parte del nodo raiz, cuya dirección se encuentra en la variable *ArbolIF*. Se compara su dato de información *InfoIndice* con el índice buscado *(I)*. Si son iguales finaliza el proceso (búsqueda con éxito) y si son distintos se decide en cual de los dos subárboles izquierdo o derecho debe encontrarse el índice *(I)* en caso de existir.

Esta decisión se toma en base a la propiedad de los árboles binarios de búsqueda, según la cual cada nodo es mayor que los de su subárbol izquierdo y menor que los de su subárbol derecho. Por tanto, si el índice buscado *(I)* es menor que el dato del nodo tratado, se avanza hacia el subárbol izquierdo cuya raíz se encuentra en el campo *HijoIzqIndice* del nodo. Por el contrario, si el índice *(I)* es mayor que el dato del nodo evaluado, se avanza hacia la raiz del subárbol derecho mediante la dirección contenida en el campo *HijoDerIndice* del nodo.

Este proceso se repetirá de forma iterativa hasta encontrar el índice (búsqueda con éxito), o bien hasta llegar al final de una rama del árbol sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de fila se encuentra fuera de rango del conjunto de índices.

En el caso de existir el índice de fila, se podría comenzar a buscar el elemento referenciado por [I , J] en el árbol de elementos del índice de fila *(I)*. Sin embargo, si no existiera el elemento, no se podría determinar si existia o no el índice de columna. El conocer este dato puede resultar interesante en muchas

situaciones, por lo que previamente a la búsqueda del elemento, se realizará la búsqueda del índice de columna *(J)*.

Esta búsqueda se llevará a cabo en el árbol de índices de columna, comenzando por la raíz del mismo, cuya dirección se encuentra en la variable *ArbolIC*. El procedimiento será el mismo comentado para el árbol de índices de fila, buscando en este caso un nodo con valor *(J)*.

Este proceso de búsqueda del índice de columna finalizará bien cuando se encuentre un nodo índice con el valor *(J)* (búsqueda con éxito), o bien al llegar al final de una rama del árbol sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de columna se encuentra fuera de rango del conjunto de índices.

Una vez determinada la existencia de los índices de fila y columna, para acceder al elemento referenciado por dichos índices, se procederá, en este primer método, a buscar en el árbol de elementos de la fila *(I)*, un nodo con clave igual al índice de columna *(J)*. Para ello, y teniendo en cuenta que los árboles de elementos son también AVLs, la localización del elemento necesitará como máximo $\log_2 n$ comparaciones.

La dirección del nodo raíz del árbol de elementos de la fila *(I)* se encuentra en el campo *ArbolElem* del nodo índice cuyo campo de información *InfoIndice* tiene el valor *(I)*, localizado previamente.

El proceso comenzará comparando el campo *InfoIndice* del nodo índice referenciado por el campo *ClaveArbolF* (que contiene para cada nodo elemento el valor de su índice de columna, clave de ordenación en su árbol de fila) de la raíz con el índice *(J)*. Si son iguales se da por finalizado el algoritmo puesto que se ha localizado el elemento (su dato de información estará contenido en el campo *InfoElemento*), mientras que si son distintos se ha de decidir en cual de los dos subárboles izquierdo o derecho se encuentra el elemento en caso de existir.

Si *(J)* es menor que el valor *ClaveArbolF^.InfoIndice* del nodo, se habrá de continuar la búsqueda en el subárbol izquierdo, por lo que se avanza hacia la

raíz de dicho subárbol en base al campo *HijoIzqArbolF* del nodo en que estamos posicionados.

Por el contrario, si *(J)* es mayor que *ClaveArbolF^.InfoIndice* del nodo, se deberá continuar la búsqueda en el subárbol derecho, por lo que se avanza hacia la raíz de dicho subárbol en base al campo *HijoDerArbolF* del nodo en que nos encontramos posicionados.

El proceso se repetirá iterativamente hasta encontrar un elemento cuyo nodo índice referenciado por *ClaveArbolF* tenga el valor *(J)* en su campo de información *InfoIndice* (búsqueda con éxito), o bien hasta llegar al final de la rama del árbol por la cual se ha realizado la búsqueda sin encontrar el nodo (búsqueda fallida).

En el Anexo B.3. se desarrolla en pseudocódigo el algoritmo correspondiente.

B) Segundo Método

En este método se procede a buscar el elemento en la columna referenciada por el índice *(J)*, a diferencia del método anterior, en el cual, la búsqueda se realizaba en la fila referenciada por el índice *(I)*.

En principio, habrá que localizar el índice de columna *(J)* en el árbol de índices de columna. Para ello se parte del nodo raíz, cuya dirección se encuentra en la variable *ArbolIC*. Se compara su dato de información *InfoIndice* con el índice buscado *(J)*. Si son iguales finaliza el proceso (búsqueda con éxito) y si son distintos se decide en cual de los dos subárboles izquierdo o derecho debe encontrarse el índice *(J)* en caso de existir.

Si el índice buscado *(J)* es menor que el dato del nodo tratado, se avanza hacia el subárbol izquierdo cuya raíz se encuentra en el campo *HijoIzqIndice* del nodo. Por el contrario, si el índice *(J)* es mayor que el dato del nodo evaluado, se avanza hacia la raíz del subárbol derecho mediante la dirección contenida en el campo *HijoDerIndice* del nodo.

Este proceso se repetirá de forma iterativa hasta encontrar el índice (búsqueda con éxito), o bien hasta llegar al final de una rama del árbol sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de columna se encuentra fuera de rango del conjunto de índices.

En el caso de existir el índice de columna, se podría comenzar a buscar el elemento referenciado por [I , J] en el árbol de elementos del índice de columna *(J)*. Sin embargo, si no existiera el elemento, no se podría determinar si existía o no el índice de fila. El conocer este dato puede resultar interesante en muchas situaciones, por lo que previamente a la búsqueda del elemento, se realizará la búsqueda del índice de fila *(I)*.

Esta búsqueda se llevará a cabo en el árbol de índices de fila, comenzando por la raíz del mismo, cuya dirección se encuentra en la variable *ArbolIF*. El procedimiento será el mismo comentado para el árbol de índices de columna, buscando en este caso un nodo con valor *(I)*.

Este proceso de búsqueda del índice de fila finalizará bien cuando se encuentre un nodo índice con el valor *(I)* (búsqueda con éxito), o bien al llegar al final de una rama del árbol sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de fila se encuentra fuera de rango del conjunto de índices.

Una vez determinada la existencia de los índices de fila y columna, para acceder al elemento referenciado por dichos índices, se procederá, en este segundo método, a buscar en el árbol de elementos de la columna *(J)*, un nodo con clave igual al índice de fila *(I)*. Para ello, y teniendo en cuenta que los árboles de elementos son también AVLs, la localización del elemento necesitará como máximo $\log_2 n$ comparaciones.

La dirección del nodo raíz del árbol de elementos de la columna *(J)* se encuentra en el campo *ArbolElem* del nodo índice cuyo campo de información *InfoIndice* tiene el valor *(J)*, localizado previamente.

El proceso comenzará comparando el campo *InfoIndice* del nodo índice referenciado por el campo *ClaveArbolC* (que contiene para cada nodo elemento el valor de su índice de fila, clave de ordenación en su árbol de columna) de la raiz con el índice *(I)*. Si son iguales se da por finalizado el algoritmo puesto que se ha localizado el elemento (su dato de información estará contenido en el campo *InfoElemento*), mientras que si son distintos se ha de decidir en cual de los dos subárboles izquierdo o derecho se encuentra el elemento en caso de existir.

Si *(I)* es menor que el valor de *ClaveArbolC^.InfoIndice* del nodo, se habrá de continuar la búsqueda en el subárbol izquierdo, por lo que se avanza hacia la raiz de dicho subárbol en base al campo *HijoIzqArbolC* del nodo en que estamos posicionados.

Por el contrario, si *(I)* es mayor que *ClaveArbolC^.InfoIndice* del nodo, se deberá continuar la búsqueda en el subárbol derecho, por lo que se avanza hacia la raiz de dicho subárbol en base al campo *HijoDerArbolC* del nodo en que nos encontramos posicionados.

El proceso se repetirá iterativamente hasta encontrar un elemento cuyo nodo indice referenciado por el campo *ClaveArbolC* tenga el valor *(I)* en su campo de información *InfoIndice* (búsqueda con éxito), o bien hasta llegar al final de la rama del árbol por la cual se ha realizado la búsqueda sin encontrar el nodo (búsqueda fallida).

En el Anexo B.3. se desarrolla en pseudocódigo el algoritmo correspondiente.

La ventaja de trabajar con árboles AVL frente a listas enlazadas, en el acceso de un elemento en base a sus índices, es que en éstas últimas las búsquedas de índices y elementos son secuenciales O(n), mientras que en los árboles las búsquedas son binarias O(log$_2$n).

3.5.4. Búsqueda de un elemento (o todos) en base a un campo de información

Esta operación consiste en localizar un elemento o todos los elementos de la estructura cuyo dato de información coincida con un dato proporcionado. Precisa de un recorrido por la estructura, para evaluar el contenido del campo de información de cada nodo. Como ya se comentó en la implementación por listas, este recorrido puede realizarse de muy diferentes formas en base al punto de comienzo y al sentido de la búsqueda.

Si se trata de localizar todos los elementos de la estructura que contienen el dato de información proporcionado, la forma de recorrido utilizada no tendrá ninguna importancia, puesto que será necesario procesar todos los elementos del array. Por el contrario, si se trata de localizar sólo un elemento que contenga el dato suministrado de búsqueda, y se pudiera deducir la zona del array donde puede encontrarse, se aplicaría en tal caso, el recorrido que antes tratase dicha zona, con el fin de reducir el tiempo del algoritmo de búsqueda.

Asimismo conviene precisar que el algoritmo va a considerar el campo *InfoElemento* como dato simple. En el caso de que fuese compuesto, debido a que en cada elemento se almacena más de una información, se habría de especificar el dato individual en base al cual se haría la búsqueda.

El algoritmo de búsqueda de un solo elemento coincidente en información con el valor proporcionado, y utilizando un criterio de recorrido del array por filas, se implementaría en base a dos ciclos: el externo, que realizará el tratamiento de las diferentes filas del array y el interno, que procesará los elementos de cada fila.

Puesto que se va a realizar el recorrido por filas, comenzando en la primera y avanzando a la siguiente tras haber evaluado todos sus elementos, será necesario obtener la secuencia ordenada de los índices de fila. Para ello habrá que recorrer en inorden el árbol de índices de fila, ya que al ser binario de búsqueda este tipo de recorrido garantiza la serie ordenada de sus elementos. La dirección de la raíz de dicho árbol de índices de fila está contenida en la variable *ArbolIF*.

El recorrido inorden comenzará lógicamente tratando el índice de fila inferior, que estará representado en el árbol por el nodo extremo de la rama más izquierda. Por cada nodo índice de fila que se obtiene en el recorrido, se deberán evaluar los elementos de dicha fila, los cuales se encuentran también formando un árbol AVL cuya raíz está apuntada por el campo *ArbolElem* del nodo índice. Este árbol de elementos podrá recorrerse en principio de cualquier forma, tanto en profundidad (preorden, inorden, postorden) como en anchura.

Ahora bien, por seguir el esquema lógico de un array, donde el recorrido de los elementos de una fila se hace en base a su disposición física, es decir, en orden de columna, en este caso aplicaremos también un recorrido inorden al árbol de elementos de la fila, puesto que en éste los nodos se encuentran ordenados en base al contenido de *ClaveArbolF^.InfoIndice*, que contiene el valor de la columna.

Para cada nodo de dicho árbol se comparará su contenido de información *InfoElemento* con el valor de búsqueda. Si ambos datos coinciden se dará por finalizado el algoritmo, mientras que en caso contrario, se obtendrá el siguiente nodo del árbol de elementos realizando la misma comparación, hasta alcanzar un elemento coincidente con el dato buscado, o bien hasta llegar al final del árbol de elementos de dicha fila.

En este último caso, una vez tratados todos los elementos de una fila, se deberá avanzar a la fila siguiente, para lo cual se obtiene el siguiente nodo índice en el recorrido del árbol de índices de fila y se repite el tratamiento anterior para los elementos de dicha fila.

El proceso finalizará al encontrar un nodo en un árbol de elementos con el dato buscado en su campo *InfoElemento* (búsqueda con éxito) o bien tras finalizar el recorrido del árbol de índices de fila sin haber encontrado ningún nodo con el dato buscado en los respectivos árboles de elementos de cada índice (búsqueda fallida).

En el caso de que sea necesario obtener todos los elementos que coinciden con el valor dado, el algoritmo será similar, diferenciándose únicamente en el criterio de finalización, que será siempre tras recorrer por completo la estructura.

En este algoritmo de búsqueda de un elemento en base a un dato de información, no existe ventaja de la implementación sobre árboles frente a la de listas, ya que se basa en un recorrido secuencial sobre la estructura.

La ventaja frente a los arrays estáticos se basa, al igual que en la implementación por listas, en el hecho de que en el *ABD* sólo se recorren y evalúan los elementos existentes, ya que no existen "posiciones vacias" como ocurre en los estáticos, las cuales, en estos últimos, también deben ser tratadas en el recorrido de búsqueda, repercutiendo de forma negativa en el rendimiento del algoritmo.

En la implementación del algoritmo de búsqueda de un elemento, se prescindirá de utilizar la solución recursiva para el recorrido, ya que éste debe finalizar en el momento de encontrar el primer elemento de información coincidente con el dato buscado.

En el algoritmo recursivo, si bien se puede definir esta situación como caso base con el fin de detener las llamadas del procedimiento a sí mismo, el sistema debe vaciar la pila, y en consecuencia ejecutar el conjunto de instrucciones que quedaban pendientes de las llamadas recursivas anteriores. Por tanto, el alcanzar el criterio base no implica finalizar el proceso.

En el algoritmo de búsqueda de todos los elementos que contengan un determinado dato de información, si se puede utilizar la solución recursiva, puesto que hay recorrer la estructura al completo.

En el Anexo B.4. se desarrollan en pseudocódigo los algoritmos correspondientes.

3.5.5. Inserción de un elemento

Para proceder a insertar un elemento en un ***ABD*** en una determinada posición de fila y columna, es necesario realizar los siguientes pasos:

a) Comprobar la existencia del índice de fila, y si no existe crearlo.

b) Comprobar la existencia del índice de columna, y si no existe crearlo.

c) Comprobar la no existencia de un elemento en la fila y columna.

d) Si ya existe un elemento en esa posición, devolver una condición de error.

e) Si no existe, crear el elemento e incluirlo en los árboles de elementos de su fila y de su columna en la posición adecuada.

El proceso de comprobación del índice de fila se realizará sobre el árbol de índices de fila, cuya raíz se encuentra apuntada por la variable *ArbolIF*. El valor del índice buscado se comparará con el valor del nodo índice en que está posicionado. Si son iguales finaliza el proceso, y si son distintos se desciende al hijo izquierdo o derecho según el índice buscado sea menor o mayor que el índice tratado. Esta secuencia se repetirá hasta encontrar el nodo índice buscado (búsqueda con éxito), o bien hasta llegar al final de la rama de nodos a lo largo de la cual se ha realizado la búsqueda (búsqueda fallida).

Cada vez que se desciende de nivel en el árbol para realizar una nueva comparación, se registra la dirección del nodo que se acaba de tratar, que será el padre del nuevo nodo a procesar. Esto se hace con el fin de que, en el caso de que no exista el índice buscado, quede registrada la dirección del que va a ser su padre en el momento de crearlo, y por tanto poder engancharlo a él, sin necesidad de realizar una búsqueda adicional para obtener la dirección de dicho padre.

En el caso de no existir el índice de fila, y en base al carácter dinámico del ***ABD*** propuesto, se deberá proceder a crearlo e insertarlo en el árbol de índices de fila.

A dicho nodo índice se le asignará el valor del índice de fila en su campo de información *InfoIndice*, y el valor nulo en sus campos *HijoIzqIndice* e *HijoDerIndice*, puesto que al insertar un nodo en un árbol binario de búsqueda siempre queda como hoja. Ahora bien, al ser un árbol AVL se debe comprobar si tras la inserción se ha modificado la condición de equilibrio del árbol, y en ese caso rehacer la estructura del mismo.

En principio se asignará también el valor nulo al campo *ArbolElem*, hasta que se cree el nodo elemento a insertar. En ese momento, se asignará a dicho campo la dirección del nuevo nodo generado.

El nodo índice generado deberá colgarse de su padre en el árbol de índices de fila. La dirección de dicho padre ha quedado registrada en el proceso de búsqueda. Habrá que evaluar el caso excepcional de que dicho padre no exista, situación que ocurrirá cuando el árbol de índices de fila esté vacío, es decir, cuando no exista ningún elemento en el ***ABD***. En este caso el nuevo índice generado será la raíz del árbol de índices de fila, y su dirección se asignará por tanto a la variable *ArbolIF*.

Si existe el nodo padre, falta por determinar si el nuevo índice debe quedar como hijo izquierdo o como hijo derecho suyo. Para ello es suficiente con comparar el índice nuevo y su índice padre. Si el índice nuevo es menor que el índice padre, deberá engancharse por la izquierda y si es mayor deberá engancharse por la derecha, para seguir conservando la estructura de árbol binario de búsqueda. Para ello bastará con asignar la dirección del nuevo índice al campo *HijoIzqIndice* o al campo *HijoDerIndice* del índice padre, respectivamente.

El mismo proceso realizado para el índice de fila deberá realizarse para el índice de columna. En este caso la comprobación del índice de columna se realizará sobre el árbol de índices de columna, cuya raíz se encuentra apuntada por la variable *ArbolIC*. El proceso será el mismo descrito anteriormente, partiendo

de la raiz y descendiendo de nivel por cada comparación fallida, y registrando la dirección del predecesor.

En el caso de no existir el índice de columna, y en base al carácter dinámico del ***ABD***, se deberá proceder a crearlo e insertarlo en el árbol de índices de columna.

A dicho nodo índice se le asignará el valor del índice de columna en su campo de información *InfoIndice*, y el valor nulo en sus campos *HijoIzqIndice* e *HijoDerIndice*, puesto que al insertar un nodo en un árbol binario de búsqueda siempre queda como hoja. Ahora bien, al ser un árbol AVL se debe comprobar si tras la inserción se ha modificado la condición de equilibrio del árbol, y en ese caso rehacer la estructura del mismo.

En principio se asignará también el valor nulo al campo *ArbolElem*, hasta que se cree el nodo elemento a insertar. En ese momento, se asignará a dicho campo la dirección del nuevo nodo generado.

El nodo índice generado deberá colgarse de su padre en el árbol de índices de columna. La dirección de dicho padre ha quedado registrada en el proceso de búsqueda. Habrá que evaluar el caso excepcional de que dicho padre no exista, situación que ocurrirá cuando el árbol de índices de columna esté vacio, es decir, cuando no exista ningún elemento en el ***ABD***. En este caso el nuevo índice generado será la raiz del árbol de índices de columna, y su dirección se asignará por tanto a la variable *ArbolIC*.

Si existe el nodo padre, falta por determinar si el nuevo índice debe quedar como hijo izquierdo o como hijo derecho suyo. Para ello es suficiente con comparar el índice nuevo y su índice padre. Si el índice nuevo es menor que el índice padre, deberá engancharse por la izquierda y si es mayor deberá engancharse por la derecha, para seguir conservando la estructura de árbol binario de búsqueda. Para ello bastará con asignar la dirección del nuevo índice al campo *HijoIzqIndice* o al campo *HijoDerIndice* del índice padre, respectivamente.

En el caso de que haya sido necesario crear alguno de los dos índices de fila y columna, lógicamente no podrá existir el elemento referenciado por ambos índices, por lo que bastará con generar el nuevo nodo con la información

correspondiente, e insertarlo en la posición adecuada en los árboles de elementos referenciados por el índice de fila y por el índice de columna.

Si por el contrario, ambos índices existían ya previamente, será necesario comprobar previamente la posible existencia del elemento en la fila y columna. Para ello se podrá realizar la búsqueda en el árbol de elementos del índice de fila, localizando un nodo cuyo campo *ClaveArbolF* referencie a un nodo índice cuyo campo *InfoIndice* sea igual al índice de columna, o bien se podrá realizar la búsqueda en el árbol de elementos del índice de columna, localizando un nodo cuyo campo *ClaveArbolC* referencie a un nodo índice cuyo campo *InfoIndice* sea igual al índice de fila.

Si existe el elemento se devolverá un código de error, y en caso contrario, se deberá generar el nodo e insertarlo en la posición adecuada en los árboles de elementos del índice de fila y del índice de columna.

Se detallan a continuación los tratamientos a realizar en cada caso:

1) *Insertar1*. Necesario crear el índice de fila y el de columna.

En este caso no será necesario comprobar la existencia previa del elemento, puesto que al no existir con anterioridad ni su índice de fila, ni su índice de columna, evidentemente no podrá existir el elemento referenciado por tales índices.

Bastará con generar el nodo, completando su campo de información (*InfoElemento*) con el dato a insertar, su campo de clave para el árbol de fila (*ClaveArbolF*) con la dirección del índice de columna *(J)* en el árbol de índices de columna, y su campo de clave para el árbol de columna (*ClaveArbolC*) con la dirección del índice de fila *(I)* en el árbol de índices de fila.

A los campos que contienen las direcciones de sus hijos izquierdo y derecho tanto para el árbol de fila, como para el árbol de columna, (*HijoIzqArbolF*, *HijoDerArbolF*, *HijoIzqArbolC*, *HijoDerArbolC*) se les asignará el valor nulo, puesto que siempre al insertar un nodo en un árbol binario de búsqueda, en principio queda como hoja.

El nodo creado se enganchará a su árbol de fila como raiz del mismo, puesto que al no existir anteriormente la fila, dicho nodo elemento será el único de su fila, y por tanto la raiz del árbol de elementos de dicha fila. Por tanto la dirección del nodo elemento se asignará al campo *ArbolElem* del índice de fila, el cual se ha creado previamente.

Lo mismo ocurrirá al enganchar dicho nodo a su árbol de columna. Al no existir la columna con anterioridad, éste nodo será el único de dicho árbol de columna, y por tanto la raiz del mismo. Su dirección se asignará, por tanto, al campo *ArbolElem* del índice de columna, el cual habrá sido creado previamente.

En ninguno de los dos árboles de elementos (el de la fila y el de la columna) puede ocurrir que tras la inserción del nodo, se rompa el equilibrio del árbol AVL correspondiente, puesto que pasan de estar vacios a tener un nodo, con lo cual la condición de equilibrio permanece inalterada.

2) *Insertar2*. Necesario crear el índice de fila y no el de columna.

Al igual que en el caso anterior, existe la seguridad de que el elemento a insertar no existe previamente en el ***ABD***, puesto que uno de sus índices de referencia, el de fila concretamente, ha sido necesario crearlo por no existir.

En principio, de forma similar al caso 1, se generará el nodo, completando su campo de información (*InfoElemento*) con el dato a insertar, su campo de clave para el árbol de fila (*ClaveArbolF*) con la dirección del índice de columna *(J)* en el árbol de índices de columna, y su campo de clave para el árbol de columna (*ClaveArbolC*) con la dirección del índice de fila *(I)* en el árbol de índices de fila.

A los campos que contienen las direcciones de sus hijos izquierdo y derecho tanto para el árbol de fila, como para el árbol de columna, (*HijoIzqArbolF*, *HijoDerArbolF*, *HijoIzqArbolC*, *HijoDerArbolC*) se les asignará el valor nulo, puesto que siempre al insertar un nodo en un árbol binario de búsqueda, en principio queda como hoja.

El elemento creado se convertirá en raiz del árbol de elementos de su fila, puesto que ésta no existia con anterioridad. Por tanto al campo *ArbolElem* del nodo indice de fila se le asignará la dirección del nodo elemento creado. No será necesario restablecer la condición de equilibrio del árbol AVL de elementos de la fila, puesto que ha pasado de no tener ningún nodo, a tener uno, con lo cual no se altera el equilibrio.

Por último, falta enganchar el nuevo nodo en la posición adecuada en el árbol de elementos de su columna. Para ello, partiendo de la raiz de dicho árbol, que será el nodo apuntado por el campo *ArbolElem* del nodo indice de columna, se irá descendiendo a lo largo de la rama en cuyo extremo va a quedar incluido el elemento. Nunca de entrada este nodo podrá ser la raiz, puesto que el árbol no estaba vacio, debido a que ya existía la columna.

Desde la raiz se irá comparando el indice de fila *(I)* del elemento insertado con el campo *InfoIndice* del nodo índice referenciado por el campo clave de árbol de columna (*ClaveArbolC*) de cada nodo tratado. Si es inferior se avanza hacia el subárbol izquierdo mediante la dirección contenida en el campo *HijoIzqArbolC*, y si es superior se avanza hacia el subárbol derecho en base a la dirección contenida en el campo *HijoDerArbolC*.

No puede darse el caso de que la comparación resulte por igual, puesto que tal circunstancia ocurriria cuando en el árbol de elementos de la columna, existiese alguno que tuviera como clave el valor *(I)*, lo cual es imposible puesto que no existia ese indice de fila.

Este proceso de comparaciones terminará cuando al realizar un avance hacia izquierda o derecha no exista nodo en dicha posición, siendo el último nodo tratado en esa rama el padre del nuevo elemento insertado en él árbol de elementos de su columna.

Si el indice de fila *(I)* es inferior al valor de *ClaveArbolC^.InfoIndice* del nodo padre, el nuevo nodo se colocará como hijo izquierdo de él, asignándose su dirección al campo *HijoIzqArbolC* del nodo padre. Si por el contrario, el indice de fila *(I)* es superior al contenido de *ClaveArbolC^.InfoIndice* del nodo padre, el nuevo

nodo deberá colocarse como hijo derecho de él, asignándose su dirección al campo *HijoDerArbolC* del nodo padre.

Si como consecuencia de la inserción del nuevo nodo en su árbol de columna, se hubiera perdido el equilibrio de éste, será necesario proceder a reestructurarlo para que siga manteniendo su condición de AVL.

3) *Insertar3*. Necesario crear el índice de columna y no el de fila.

Lo mismo que ocurre en los dos casos anteriores, aquí también existe la seguridad de que el elemento a insertar no existe previamente en el ***ABD***, puesto que uno de sus índices de referencia, el de columna concretamente, ha sido necesario crearlo por no existir.

En principio se generará el nodo, completando su campo de información (*InfoElemento*) con el dato a insertar, su campo de clave para el árbol de fila (*ClaveArbolF*) con la dirección del índice de columna *(J)* en el árbol de índices de columna, y su campo de clave para el árbol de columna (*ClaveArbolC*) con la dirección del índice de fila *(I)* en el árbol de índices de fila.

A los campos que contienen las direcciones de sus hijos izquierdo y derecho tanto para el árbol de fila, como para el árbol de columna, (*HijoIzqArbolF*, *HijoDerArbolF*, *HijoIzqArbolC*, *HijoDerArbolC*) se les asignará el valor nulo, puesto que siempre al insertar un nodo en un árbol binario de búsqueda, en principio queda como hoja.

El elemento creado se convertirá en raíz del árbol de elementos de su columna, puesto que ésta no existía con anterioridad. Por tanto al campo *ArbolElem* del nodo índice de columna se le asignará la dirección del nodo elemento creado. No será necesario restablecer la condición de equilibrio del árbol AVL de elementos de la columna, puesto que ha pasado de no tener ningún nodo, a tener uno, con lo cual no se altera el equilibrio.

Por último, falta enganchar el nuevo nodo en la posición adecuada en el árbol de elementos de su fila. Para ello, partiendo de la raiz de dicho árbol, que será el nodo apuntado por el campo *ArbolElem* del nodo índice de fila, se irá

descendiendo a lo largo de la rama en cuyo extremo va a quedar incluido el elemento. Nunca de entrada este nodo podrá ser la raíz, puesto que el árbol no estaba vacío, debido a que ya existía la fila.

Desde la raíz se irá comparando el índice de columna *(J)* del elemento insertado con el campo *InfoIndice* del nodo índice referenciado por el campo clave de árbol de fila (*ClaveArbolF*) de cada nodo tratado. Si es inferior se avanza hacia el subárbol izquierdo mediante la dirección contenida en el campo *HijoIzqArbolF*, y si es superior se avanza hacia el subárbol derecho en base a la dirección contenida en el campo *HijoDerArbolF*.

No puede darse el caso de que la comparación resulte por igual, puesto que tal circunstancia ocurriría cuando en el árbol de elementos de la fila, existiese alguno que tuviera como clave el valor *(J)*, lo cual es imposible puesto que no existía ese índice de columna.

Este proceso de comparaciones terminará cuando al realizar un avance hacia izquierda o derecha no exista nodo en dicha posición, siendo el último nodo tratado en esa rama el padre del nuevo elemento insertado en el árbol de elementos de su fila.

Si el índice de columna *(J)* es inferior al valor de *ClaveArbolF^.InfoIndice* del nodo padre, el nuevo nodo se colocará como hijo izquierdo de él, asignándose su dirección al campo *HijoIzqArbolF* del nodo padre. Si por el contrario, el índice de columna *(J)* es superior al contenido de *ClaveArbolF^.InfoIndice* del nodo padre, el nuevo deberá colocarse como hijo derecho de él, asignándose su dirección al campo *HijoDerArbolF* del nodo padre.

Si como consecuencia de la inserción del nuevo nodo en su árbol de fila, se hubiera perdido el equilibrio de éste, será necesario proceder a reestructurarlo para que siga manteniendo su condición de AVL.

4) *Insertar4*. Existen el índice de fila y el de columna.

En tal situación, a diferencia de los casos anteriores, es posible que exista ya en la estructura el elemento referenciado por tales índices, por lo que previamente habrá que realizar un proceso de búsqueda.

Dicho proceso se puede realizar sobre el árbol de elementos de la fila, partiendo de su nodo raíz, cuya dirección se encuentra en el campo *ArbolElem* del nodo índice de fila.

Los elementos de la fila, se encuentran ordenados en su árbol, en base al valor de *ClaveArbolF^.InfoIndice*, que proporciona para cada nodo el índice de columna. Por tanto se deberá comparar el índice de columna *(J)* del elemento buscado con el contenido de *ClaveArbolF^.InfoIndice* de cada nodo tratado en el árbol comenzando por su raíz.

Si son iguales se dará por finalizada la búsqueda, puesto que ha sido encontrado el nodo, en cuyo caso el algoritmo de inserción devolverá un código de error, debido a que no es posible realizar la inserción puesto que ya existe un elemento en la posición referenciada por los índices de fila y columna.

Si son distintos, se deberá decidir si se continua la búsqueda en el subárbol izquierdo o en el derecho del nodo tratado. Si *(J)* es menor al valor de *ClaveArbolF^.InfoIndice* del nodo procesado, se avanzará hacia el subárbol izquierdo, estando la dirección de la raíz de dicho subárbol en el campo *HijoIzqArbolF* del nodo. Si por el contrario, *(J)* es mayor al contenido de *ClaveArbolF^.InfoIndice* del nodo procesado, se avanzará hacia el subárbol derecho, estando la dirección de la raíz de dicho subárbol en el campo *HijoDerArbolF* del nodo.

Si tras las sucesivas comparaciones se alcanza el final de la rama del árbol utilizada en la búsqueda, quiere decir que no existe ningún elemento referenciado por tales índices de fila y columna, por lo que se puede proceder a la inserción.

En tal caso se generará el nodo, completando su campo de información (*InfoElemento*) con el dato a insertar, su campo de clave para el árbol de fila (*ClaveArbolF*) con la dirección del índice de columna *(J)* en el árbol de índices de columna, y su campo de clave para el árbol de columna (*ClaveArbolC*) con la dirección del índice de fila *(I)* en el árbol de índices de fila.

A los campos que contienen las direcciones de sus hijos izquierdo y derecho tanto para el árbol de fila, como para el árbol de columna, (*HijoIzqArbolF*, *HijoDerArbolF*, *HijoIzqArbolC*, *HijoDerArbolC*) se les asignará el valor nulo, puesto que siempre al insertar un nodo en un árbol binario de búsqueda, en principio queda como hoja.

A continuación habrá que enganchar el elemento en la posición adecuada, tanto en el árbol de elementos de su fila, como en el árbol de elementos de su columna. En ninguno de los dos quedará como raíz, puesto que ya existían los índices de fila y columna, y por tanto los árboles de elementos de ambos no están vacíos.

En la búsqueda anterior del elemento en el árbol de su fila, el último nodo tratado en la rama recorrida, será el padre del nuevo elemento insertado. Si el índice de columna *(J)* es inferior al valor de *ClaveArbolF^.InfoIndice* del nodo padre, el nuevo nodo insertado se colocará como hijo izquierdo de él, asignándose su dirección al campo *HijoIzqArbolF* del nodo padre. Si por el contrario, el índice de columna *(J)* es superior al contenido de *ClaveArbolF^.InfoIndice* del nodo padre, el nuevo nodo insertado deberá colocarse como hijo derecho de él, asignándose su dirección al campo *HijoDerArbolF* del nodo padre.

Si como consecuencia de la inserción del nuevo nodo en su árbol de fila, se hubiera perdido el equilibrio de éste, será necesario proceder a reestructurarlo para que siga manteniendo su condición de AVL.

Para enganchar el nuevo nodo en el árbol de elementos de su columna habrá que partir de la raíz de dicho árbol, que será el nodo apuntado por el campo *ArbolElem* del nodo índice de columna, e ir descendiendo a lo largo de la rama en cuyo extremo va a quedar incluido el nodo.

Desde la raíz se irá comparando el índice de fila *(I)* del elemento insertado con el campo *InfoIndice* del nodo índice referenciado por el campo clave de árbol de columna (*ClaveArbolC*) de cada nodo tratado. Si es inferior se avanza hacia el subárbol izquierdo mediante la dirección contenida en el campo *HijoIzqArbolC*, y si es superior se avanza hacia el subárbol derecho en base a la dirección contenida en el campo *HijoDerArbolC*. No puede darse el caso de que la comparación resulte por igual, puesto que ya se ha determinado previamente que no existía el elemento en el array.

Este proceso de comparaciones terminará cuando al realizar un avance hacia izquierda o derecha no exista nodo en dicha posición, siendo el último nodo tratado en esa rama el padre del nuevo elemento insertado en el árbol de elementos de su columna.

Si el índice de fila *(I)* es inferior al valor de *ClaveArbolC^.InfoIndice* del nodo padre, el nuevo nodo se colocará como hijo izquierdo de él, asignándose su dirección al campo *HijoIzqArbolC* del nodo padre. Si por el contrario, el índice de fila *(I)* es superior al contenido de *ClaveArbolC^.InfoIndice* del nodo padre, el nuevo nodo deberá colocarse como hijo derecho de él, asignándose su dirección al campo *HijoDerArbolC* del nodo padre.

Si como consecuencia de la inserción del nuevo nodo en su árbol de columna, se hubiera perdido el equilibrio de éste, será necesario proceder a reestructurarlo para que siga manteniendo su condición de AVL.

En este algoritmo de inserción de un elemento en el ***ABD***, la ventaja de la implementación mediante árboles AVL frente a la implementación por listas enlazadas, se basa en la mayor eficiencia de las búsquedas binarias $O(\log_2 n)$ frente a las búsquedas secuenciales $O(n)$.

En el Anexo B.5. se desarrollan en pseudocódigo los algoritmos correspondientes.

3.5.6. Borrado de un elemento referenciado por sus índices

El proceso de eliminación de un elemento referenciado por sus índices en un ***ABD*** comprende los siguientes pasos:

a) Comprobar la existencia del índice de fila. Si no existe, devolver condición de error y fin del algoritmo.

b) Comprobar la existencia del índice de columna. Si no existe, devolver condición de error y fin del algoritmo.

c) Comprobar la existencia del elemento en la fila y columna. Si no existe, devolver condición de error y fin del algoritmo.

d) Si existe, proceder a eliminarlo del árbol de elementos del índice de fila y del árbol de elementos del índice de columna, reestructurando dichos árboles.

e) Evaluar la condición de equilibrio en los árboles y si se ha perdido restablecerla, modificando la estructura de los mismos.

f) Si tras la eliminación del elemento, el árbol de elementos de alguno de los dos índices se queda vacío, proceder a eliminar el índice en el árbol de índices correspondiente.

g) En tal caso, si se pierde la condición de equilibrio, restablecerla en el árbol de índices que corresponda.

En principio, habrá que localizar el índice de fila *(I)* en el árbol de índices de fila. Para ello se parte del nodo raíz, cuya dirección se encuentra en la variable *ArbolIF*. Se compara su dato de información *InfoIndice* con el índice buscado *(I)*. Si son iguales finaliza el proceso (búsqueda con éxito) y si son distintos se decide en cual de los dos subárboles izquierdo o derecho debe encontrarse el índice *(I)* en caso de existir.

Si el índice buscado *(I)* es menor que el dato del nodo tratado, se avanza hacia el subárbol izquierdo cuya raiz se encuentra en el campo *HijoIzqIndice* del nodo. Por el contrario, si el índice *(I)* es mayor que el dato del nodo evaluado, se avanza hacia la raíz del subárbol derecho mediante la dirección contenida en el campo *HijoDerIndice* del nodo.

Este proceso se repetirá de forma iterativa hasta encontrar el índice (búsqueda con éxito), o bien hasta llegar al final de la rama del árbol por la que se ha descendido sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de fila se encuentra fuera de rango del conjunto de índices.

En el caso de existir el índice de fila, se podría comenzar a buscar el elemento referenciado por [I , J] en el árbol de elementos del índice de fila *(I)*. Sin embargo, si no existiera el elemento, no se podría determinar si existia o no el índice de columna. El conocer este dato puede resultar interesante en muchas situaciones, por lo que previamente a la búsqueda del elemento, se realizará la búsqueda del índice de columna *(J)*.

Esta búsqueda se llevará a cabo en el árbol de índices de columna, comenzando por la raíz del mismo, cuya dirección se encuentra en la variable *ArbolIC*. El procedimiento será el mismo comentado para el árbol de índices de fila, buscando en este caso un nodo con valor *(J)*.

Este proceso de búsqueda del índice de columna finalizará bien cuando se encuentre un nodo índice con el valor *(J)* (búsqueda con éxito), o bien al llegar al final de la rama recorrida del árbol sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de columna se encuentra fuera de rango del conjunto de índices.

Una vez determinada la existencia de los índices de fila y columna, para acceder al elemento referenciado por dichos índices, se puede o bien buscar en el árbol de elementos de la fila *(I)*, un nodo con clave igual al índice de columna *(J)*, o bien buscar en el árbol de elementos de la columna *(J)*, un nodo con clave igual al índice de fila *(I)*.

Basándose en el primer método, la dirección del nodo raíz del árbol de elementos de la fila *(I)* se encuentra en el campo *ArbolElem* del nodo índice cuyo campo de información *InfoIndice* tiene el valor *(I)*, localizado previamente.

El proceso comenzará comparando el valor de *ClaveArbolF^.InfoIndice* (que contiene para cada nodo elemento el valor de su índice de columna, clave de ordenación en su árbol de fila) de la raíz con el índice *(J)*. Si son iguales se da por finalizada la búsqueda puesto que se ha localizado el elemento, mientras que si son distintos se ha de decidir en cual de los dos subárboles izquierdo o derecho se encuentra el elemento en caso de existir.

Si *(J)* es menor que el valor de *ClaveArbolF^.InfoIndice* del nodo, se habrá de continuar la búsqueda en el subárbol izquierdo, por lo que se avanza hacia la raíz de dicho subárbol en base al campo *HijoIzqArbolF* del nodo en que estamos posicionados. Se registrará también la dirección de dicho nodo como padre del nodo hacia el que se desciende.

Por el contrario, si *(J)* es mayor que el contenido de *ClaveArbolF^.InfoIndice* del nodo, se deberá continuar la búsqueda en el subárbol derecho, por lo que se avanza hacia la raíz de dicho subárbol en base al campo *HijoDerArbolF* del nodo en que nos encontramos posicionados. Se registrará también la dirección de dicho nodo como padre del nodo hacia el que se desciende.

El proceso se repetirá iterativamente hasta encontrar un elemento para el que el valor de *ClaveArbolF^.InfoIndice* tenga el valor *(J)* (búsqueda con éxito), o bien hasta llegar al final de la rama del árbol por la cual se ha realizado la búsqueda sin encontrar el elemento (búsqueda fallida). En este último caso se devolverá un código de error indicativo de la no existencia del elemento referenciado por los índices [I , J].

En caso de existir se procederá a localizar el elemento en el árbol de elementos de su columna con el objeto de obtener la dirección de su padre en dicho árbol. Para ello partiendo de la raíz, cuya dirección se encuentra en el campo *ArbolElem* del nodo índice de columna, se irá descendiendo de nivel a izquierda o derecha (según el índice *(I)* sea menor o mayor que el contenido de

ClaveArbolC^.InfoIndice de cada nodo), y registrando la dirección del padre, hasta llegar al nodo que se ha de eliminar y previamente localizado en el proceso anterior.

A continuación se debe proceder a desenganchar el nodo de los árboles de elementos de su fila y de su columna, y destruir el nodo.

Para eliminar un nodo en un árbol binario de búsqueda, se deben distinguir los siguientes casos:

a) ***El nodo no tiene hijos.***

En este caso bastará con asignar el valor nulo al campo del nodo padre que contiene la dirección del nodo a eliminar. Ese campo será el de hijo izquierdo o derecho, según la posición relativa del nodo a eliminar respecto a su padre.

La excepción a tener en cuenta se producirá cuando el nodo a borrar sea la raiz (no tiene padre), en cuyo caso el valor nulo se asignará a la variable puntero externa que apunta al nodo raiz.

b) ***El nodo tiene un solo hijo.***

En este caso se deberá asignar la dirección del hijo al campo del nodo padre que contiene la dirección del nodo a eliminar. Como en el caso anterior, este campo será el de hijo izquierdo o derecho, según la posición relativa del nodo a eliminar respecto a su padre.

La excepción a tener en cuenta se producirá cuando el nodo a borrar sea la raiz (no tiene padre), en cuyo caso la dirección del hijo se asignará a la variable puntero externa que apunta al nodo raiz.

De hecho, ambos casos son idénticos, puesto que el caso a es igual al caso b siendo la dirección del hijo el valor nulo.

c) *El nodo tiene dos hijos.*

En esta situación se debe reemplazar la posición del nodo que se quiere eliminar por uno de los nodos ya existentes, y para que este procedimiento sea correcto, y el árbol siga manteniendo la estructura de binario de búsqueda, es necesario que el nodo que va a reemplazar al que se va a borrar sea su predecesor en inorden (el mayor de los menores a él) o su sucesor en inorden (el menor de los mayores a él).

Si se elige la sustitución del nodo a borrar por su predecesor en inorden, habrá que localizar, en principio, dicho predecesor en inorden. Al ser el elemento inmediatamente inferior al que se va a eliminar, se encontrará localizado en su subárbol izquierdo (nodos menores a él) en el extremo de la rama situada más a la derecha del subárbol (máximo).

Si se hubiese optado por reemplazar el nodo a borrar por su sucesor en inorden (inmediatamente superior), éste se encontraría situado en el subárbol derecho del nodo (nodos mayores a él) en el extremo de la rama situada más a la izquierda del subárbol (mínimo).

Una vez localizado el predecesor en inorden, se debe desenganchar temporalmente del árbol para engancharlo en la posición que ocupa el nodo a borrar. El predecesor en inorden como máximo podrá tener hijo izquierdo, pero nunca hijo derecho puesto que es el extremo derecho del subárbol izquierdo. Por lo tanto para desengancharlo del árbol se aplicará el caso a (si no tiene hijos) o el caso b (si tiene hijo izquierdo).

A continuación deberá pasar a ocupar en el árbol la posición que ocupa el nodo a suprimir. Para ello su dirección se le asignará al campo del nodo padre del nodo a suprimir que contiene la dirección de dicho nodo. Ese campo será el de hijo izquierdo o derecho, según la posición relativa del nodo a eliminar respecto a su padre. La excepción a tener en cuenta se producirá cuando el nodo a borrar sea la raiz (no tiene padre), en cuyo caso la dirección del predecesor en inorden se asignará a la variable puntero externa que apunta al nodo raiz.

Por último, para reemplazar el nodo a borrar, será necesario asignar las direcciones de los hijos izquierdo y derecho de éste nodo, a los campos de hijo izquierdo y derecho respectivamente del predecesor en inorden.

Una vez que el nodo a suprimir ha sido reemplazado en el árbol por su predecesor en inorden, se puede destruir para recuperar el espacio en memoria que estaba ocupando.

Este procedimiento se deberá aplicar doblemente para desenganchar el nodo a suprimir del árbol de elementos de su fila y del árbol de elementos de su columna. Sin embargo, habrá que tener en cuenta que la destrucción física del nodo deberá realizarse una única vez, tras desenganchar el nodo de ambos árboles.

Se habrá de evaluar la situación de equilibrio tanto en el árbol AVL de elementos de fila como en el de columna donde se encontraba el elemento eliminado, y proceder, si es necesario, a reequilibrarlo.

Puede ocurrir que como consecuencia de la eliminación del nodo, el árbol de elementos de su fila y/o de su columna se quedasen vacios.

Si el árbol de elementos de su fila se quedase vacío, sería debido a que el elemento borrado era el único de su fila. En ese caso, no tiene sentido ya la existencia del índice de fila, y se procederá a eliminarlo del árbol de índices de fila siguiendo el tratamiento descrito anteriormente.

Si como consecuencia de la eliminación del nodo índice de fila se pierde la condición de equilibrio del AVL de índices de fila se deberá proceder a reorganizar dicho árbol con el fin de restablecer de nuevo su condición de equilibrio.

Lo mismo ocurrirá si tras la eliminación del nodo, el árbol de elementos de su columna se queda vacío. Este hecho sería debido a que el elemento borrado era el único de su columna. En ese caso, no tiene sentido ya la existencia del índice de columna, y se procederá a eliminarlo del árbol de índices de columna siguiendo el tratamiento descrito anteriormente.

Si como consecuencia de la eliminación del nodo índice de columna se pierde la condición de equilibrio del AVL de índices de columna se deberá proceder a reorganizar dicho árbol con el fin de restablecer de nuevo su condición de equilibrio.

En este algoritmo de borrado, la ventaja de la implementación del ***ABD*** mediante AVL frente a la implementación por listas enlazadas, se basa al igual que en los algoritmos anteriores, en la mayor eficiencia de las búsquedas binarias frente a las búsquedas secuenciales.

Además, como ya ha quedado reflejado en la implementación por listas, la ventaja del ***ABD*** frente a los arrays estáticos estriba en la recuperación del espacio de los elementos borrados, el reajuste dinámico en los conjuntos de índices cuando éstos dejen de referenciar elementos, y el control del error debido a la no existencia de la posición referenciada por índices fuera de rango. Cuando los índices son correctos, el acceso a la posición en los arrays estáticos es más rápida, debido a su implementación interna.

En el Anexo B.6. se desarrollan en pseudocódigo los algoritmos correspondientes.

3.5.7. Borrado de un elemento (o todos) en base a un campo de información

En este algoritmo es necesario resaltar que en el ***ABD*** pueden existir varios elementos con el mismo dato en el campo de información utilizado en la búsqueda si no se trata de un campo clave, por lo que el borrado aplicado al primer elemento que contenga dicho dato, generará resultados diferentes según el criterio de recorrido utilizado en el array.

Se diseñarán dos algoritmos diferentes: uno que borre el primer elemento encontrado que contenga el dato proporcionado, y otro que borre todos los elementos existentes en el array que cumplan dicha condición.

En el ***ABD*** el proceso de borrado del primer elemento que contenga la información proporcionada, se basará también en el algoritmo de búsqueda de un elemento en base a un campo de información.

Se elegirá un sentido de recorrido determinado, que en este caso, y por similitud al algoritmo de búsqueda puede ser por filas. En tal caso, se irá recorriendo en inorden el árbol de índices de fila (para tratarlos en orden creciente), y por cada nodo índice de este árbol se recorrerá también en inorden el árbol de elementos referenciados por dicho índice de fila (para tratarlos en orden creciente de columna).

Para cada nodo elemento se comparará el campo de información con el dato de borrado. Dicho proceso finalizará cuando se cumpla alguna de las dos condiciones siguientes:

a) Se accede a un nodo cuyo campo de información coincide con el dato a borrar. Se procede a eliminar dicho nodo, reorganizando los árboles de elementos de su fila y de su columna (según se ha descrito en el procedimiento anterior, en función de que el nodo tenga 0, 1 o 2 hijos), y si fuese necesario los de índices (en caso de que el elemento fuese el único de su fila o de su columna). Asimismo se restablecerá el equilibrio de los AVL si se hubiese visto alterado.

b) Se alcanza el final de la estructura sin encontrar ningún nodo coincidente en su campo de información con el dato suministrado.

En el caso de que sea necesario borrar todos los nodos coincidentes en su campo de información con el valor dado, el proceso únicamente diferirá del anterior en el criterio de terminación, que será siempre al llegar al final de la estructura, puesto que será necesario evaluar todos los elementos del array.

Para facilitar la implementación recursiva del algoritmo, se realizará en primer lugar un recorrido por filas del array, desenganchando de sus árboles de fila todos los elementos coincidentes en información con el dato proporcionado.

A continuación se realizará un recorrido de la matriz por columnas, procediendo a desconectar dichos elementos de sus correspondientes árboles de columna y a eliminarlos físicamente de memoria.

Ahora bien, en este caso, para simplificar la implementación del proceso, si bien los árboles de índices se recorrerán en inorden (con el fin de tratar ordenadamente las filas en el primer recorrido y las columnas en el segundo), los árboles de elementos de cada fila y de cada columna se recorrerán en postorden.

Esto es debido a que se van a ir eliminando nodos a medida que se realiza el recorrido, por lo que resulta imprescindible antes de tratar un nodo, haber tratado ya sus hijos para evitar complicar en exceso el proceso. El recorrido en postorden asegura este hecho, debido a que dado un árbol binario, recorre previamente los subárboles izquierdo y derecho antes de tratar la raíz.

Por lo tanto, los elementos de cada fila no se van a recorrer en orden de columna, pero este hecho no tiene ninguna importancia puesto que se van a procesar todos los elementos, y el orden en que se traten no varía en nada el resultado final de este algoritmo. Lo mismo ocurrirá con los elementos de cada columna.

El proceso de reorganizar los árboles de índices de fila y columna se realizará tras haber borrado los elementos coincidentes en información al dato proporcionado. Se deberán recorrer ambos árboles y proceder a eliminar todos aquellos nodos índices cuyo campo *ArbolElem* contenga el valor nulo, es decir, aquellos índices que ya no referencian ningún elemento.

La forma de recorrer ambos árboles será también en postorden, por el mismo motivo explicado anteriormente (antes de tratar un nodo haber tratado ya sus hijos).

Asimismo será necesario restablecer el estado de equilibrio en todos aquellos árboles, tanto de índices como de elementos, que a consecuencia de los borrados hayan perdido su condición de AVL.

El rendimiento de este algoritmo sobre AVL es algo inferior al diseñado sobre listas, debido a que en la implementación por listas, al ser éstas dobles, una vez localizado un elemento a borrar, se obtienen de forma inmediata las direcciones de sus predecesores en fila y columna, al estar almacenadas en el

nodo. Por el contrario, en los árboles no se tiene acceso directo desde cada nodo a su nodo padre, requiriéndose un tratamiento adicional para obtener su dirección.

En cualquier caso en este proceso, la ventaja frente a los arrays estáticos sigue siendo considerable, debido a que en el recorrido del ***ABD*** únicamente se tratan los elementos presentes en la estructura, a diferencia de los arrays estáticos donde se deben tratar tanto los elementos existentes como las posiciones vacías reservadas para futuros elementos. Por tanto, la diferencia entre ambos será tanto mayor cuanto más grande sea el array y más bajo su grado de ocupación.

Además en el caso del ***ABD*** se recupera el espacio de los elementos eliminados y de los índices inutilizados, cosa que no ocurre en los estáticos, al mantener éstos siempre las mismas dimensiones y ocupación de memoria a lo largo de la vida del array.

En el Anexo B.7. se desarrollan en pseudocódigo los algoritmos correspondientes.

3.5.8. Modificación de un elemento referenciado por sus índices

El proceso de modificación de un elemento referenciado por sus índices en un ***ABD*** exige realizar los siguientes pasos:

a) Comprobar la existencia del índice de fila. Si no existe, devolver condición de error y fin del algoritmo.

b) Comprobar la existencia del índice de columna. Si no existe, devolver condición de error y fin del algoritmo.

c) Comprobar la existencia del elemento en la fila y columna. Si no existe, devolver condición de error y fin del algoritmo.

d) Si existe, proceder a reemplazar su información por la nueva proporcionada.

En principio, habrá que localizar el índice de fila *(I)* en el árbol de índices de fila. Para ello se parte del nodo raíz, cuya dirección se encuentra en la variable *ArbolIF*. Se compara su dato de información *InfoIndice* con el índice buscado *(I)*. Si son iguales finaliza el proceso (búsqueda con éxito) y si son distintos se decide en cual de los dos subárboles izquierdo o derecho debe encontrarse el índice *(I)* en caso de existir.

Si el índice buscado *(I)* es menor que el dato del nodo tratado, se avanza hacia el subárbol izquierdo cuya raíz se encuentra en el campo *HijoIzqIndice* del nodo. Por el contrario, si el índice *(I)* es mayor que el dato del nodo evaluado, se avanza hacia la raíz del subárbol derecho mediante la dirección contenida en el campo *HijoDerIndice* del nodo.

Este proceso se repetirá de forma iterativa hasta encontrar el índice (búsqueda con éxito), o bien hasta llegar al final de la rama del árbol por la que se ha descendido sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de fila se encuentra fuera de rango del conjunto de índices.

En el caso de existir el índice de fila, se podría comenzar a buscar el elemento referenciado por [I , J] en el árbol de elementos del índice de fila *(I)*. Sin embargo, si no existiera el elemento, no se podría determinar si existía o no el índice de columna. El conocer este dato puede resultar interesante en muchas situaciones, por lo que previamente a la búsqueda del elemento, se realizará la búsqueda del índice de columna *(J)*.

Esta búsqueda se llevará a cabo en el árbol de índices de columna, comenzando por la raíz del mismo, cuya dirección se encuentra en la variable *ArbolIC*. El procedimiento será el mismo comentado para el árbol de índices de fila, buscando en este caso un nodo con valor *(J)*.

Este proceso de búsqueda del índice de columna finalizará bien cuando se encuentre un nodo índice con el valor *(J)* (búsqueda con éxito), o bien al llegar al final de la rama recorrida del árbol sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de columna se encuentra fuera de rango del conjunto de índices.

Una vez determinada la existencia de los índices de fila y columna, para acceder al elemento referenciado por dichos índices, se puede o bien buscar en el árbol de elementos de la fila *(I)*, un nodo con clave igual al índice de columna *(J)*, o bien buscar en el árbol de elementos de la columna *(J)*, un nodo con clave igual al índice de fila *(I)*.

Basándose en el primer método, la dirección del nodo raíz del árbol de elementos de la fila *(I)* se encuentra en el campo *ArbolElem* del nodo índice cuyo campo de información *InfoIndice* tiene el valor *(I)*, localizado previamente.

El proceso comenzará comparando el valor de *ClaveArbolF^.InfoIndice* (que contiene para cada nodo elemento el valor de su índice de columna, clave de ordenación en su árbol de fila) de la raíz con el índice *(J)*. Si son iguales se da por finalizada la búsqueda puesto que se ha localizado el elemento, mientras que si son distintos se ha de decidir en cual de los dos subárboles izquierdo o derecho se encuentra el elemento en caso de existir.

Si *(J)* es menor que el valor de *ClaveArbolF^.InfoIndice* del nodo, se habrá de continuar la búsqueda en el subárbol izquierdo, por lo que se avanza hacia la raíz de dicho subárbol en base al campo *HijoIzqArbolF* del nodo en que estamos posicionados.

Por el contrario, si *(J)* es mayor que el contenido de *ClaveArbolF^.InfoIndice* del nodo, se deberá continuar la búsqueda en el subárbol derecho, por lo que se avanza hacia la raíz de dicho subárbol en base al campo *HijoDerArbolF* del nodo en que nos encontramos posicionados.

El proceso se repetirá iterativamente hasta encontrar un elemento en que el valor de *ClaveArbolF^.InfoIndice* tenga el valor *(J)* (búsqueda con éxito), o bien hasta llegar al final de la rama del árbol por la cual se ha realizado la búsqueda sin encontrar el elemento (búsqueda fallida). En este último caso se devolverá un código de error indicativo de la no existencia del elemento referenciado por los índices [I , J].

En caso de existir el elemento, se procederá a sustituir su información actual por la nueva suministrada como entrada al algoritmo.

En este algoritmo la ventaja de implementar el ***ABD*** con AVLs en vez de con listas enlazadas, reside, al igual que en algoritmos anteriores, en agilizar los procesos de búsqueda (índice de fila, índice de columna, y elemento) al utilizarse búsqueda binaria frente a búsqueda secuencial.

En el Anexo B.8. se desarrolla en pseudocódigo el algoritmo correspondiente.

3.5.9. Modificación de un elemento (o todos) en base a un campo de información

La operación de modificación de un elemento localizado en base a un campo de información, consiste en realizar una búsqueda del elemento, y a continuación si existe, proceder a sustituir su información por la nueva suministrada.

Puesto que en principio es necesario buscar el elemento, son válidas las consideraciones hechas en el algoritmo de búsqueda: si el campo de información utilizado en la búsqueda no es campo clave, pueden existir varios elementos que contengan el dato a modificar, por lo cual si se opta por modificar el primero que se encuentre en el proceso de búsqueda, el elemento seleccionado, y en consecuencia el resultado obtenido, será diferente según la forma de recorrido que se aplique.

En el diseño de los algoritmos consideraremos también el caso de modificar todos los elementos cuyo campo de información tenga el dato de búsqueda.

Se elegirá un sentido de recorrido determinado, que en este caso, y por similitud al algoritmo de búsqueda puede ser por filas. En tal caso, se irá recorriendo en inorden el árbol de índices de fila (para tratarlos en orden creciente), y por cada nodo índice de este árbol se recorrerá también en inorden el árbol de elementos referenciados por dicho índice de fila (para tratarlos en orden creciente de columna).

Para cada nodo elemento se comparará el campo de información con el dato a modificar proporcionado. Dicho proceso finalizará cuando se cumpla alguna de las dos condiciones siguientes:

a) Se accede a un elemento cuyo campo de información coincide con el dato a modificar. Se procede a sustituir su información por la nueva suministrada.

b) Se alcanza el final del array sin encontrar ningún elemento coincidente en su campo de información con el dato suministrado.

En el caso de que sea necesario modificar todos los nodos coincidentes en su campo de información con el valor dado, el proceso únicamente diferirá del anterior en el criterio de terminación, que será siempre al llegar al final de la estructura, puesto que será necesario evaluar todos los elementos del array.

Al tener que aplicar un recorrido secuencial por la estructura, la implementación por medio de árboles AVL es idéntica en eficiencia a la implementación por medio de listas.

Se mantiene por supuesto la ventaja frente a los arrays bidimensionales estáticos, ya que al no existir posiciones vacías inutilizadas, en el recorrido únicamente se procesan los elementos realmente existentes en el array. Por ello la diferencia entre ambos será tanto más acusada cuanto mayor sea el array y menos elementos tenga.

En el Anexo B.9. se desarrollan en pseudocódigo los algoritmos correspondientes.

3.5.10. Eliminación de una fila

Este proceso consiste en localizar el índice a eliminar en el árbol de índices de fila y proceder a su borrado, suprimiendo también todos los elementos de dicha fila (borrado del árbol de elementos de fila). Como consecuencia de este borrado múltiple, será necesario reorganizar los árboles de elementos de las columnas en que se encontraban los nodos eliminados, así como reajustar si procede el árbol de índices de columna.

En principio, habrá que localizar el índice de fila a eliminar *(I)* en el árbol de índices de fila. Para ello se parte del nodo raíz, cuya dirección se encuentra en la variable *ArbolIF*. Se compara su dato de información *InfoIndice* con el índice buscado *(I)*. Si son iguales finaliza el proceso (búsqueda con éxito) y si son distintos se decide en cual de los dos subárboles izquierdo o derecho debe encontrarse el índice *(I)* en caso de existir.

Si el índice buscado *(I)* es menor que el dato del nodo tratado, se avanza hacia el subárbol izquierdo cuya raíz se encuentra en el campo *HijoIzqIndice* del nodo. Por el contrario, si el índice *(I)* es mayor que el dato del nodo evaluado, se avanza hacia la raíz del subárbol derecho mediante la dirección contenida en el campo *HijoDerIndice* del nodo.

Este proceso se repetirá de forma iterativa hasta encontrar el índice (búsqueda con éxito), o bien hasta llegar al final de la rama del árbol por la que se ha descendido sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de fila se encuentra fuera de rango del conjunto de índices.

En caso de existir se deberá proceder a su eliminación del árbol de índices de fila según el procedimiento ya descrito, en función de que el nodo tenga 0, 1 o 2 hijos. Previamente al borrado se deberá salvar el contenido de su campo *ArbolElem*, que contiene la dirección de la raíz de su árbol de elementos.

A continuación se deberán eliminar todos los nodos que componen el árbol de elementos de la fila en cuestión. Para facilitar este proceso, se definirá recursivamente recorriendo el árbol en postorden de tal forma que el borrado se va aplicando sobre hojas, ya que en dicho tipo de recorrido antes de tratar un nodo se tratan previamente sus hijos izquierdo y derecho. Por lo tanto, se estarán eliminando siempre nodos con 0 hijos lo cual simplifica notablemente el proceso.

Para cada nodo a eliminar en el árbol de elementos de fila, será necesario eliminarlo también de su árbol de elementos de columna, por lo que se necesita conocer la dirección del padre del nodo en el árbol de elementos de su columna.

Para ello, partiendo de la raiz de dicho árbol (su dirección se encuentra en el campo *ArbolElem* del nodo indice referenciado por el campo *ClaveArbolF* del nodo) se va descendiendo de nivel a izquierda o derecha, en base a las comparaciones realizadas sobre el valor de *ClaveArbolC^.InfoIndice*, y registrando la dirección del padre en cada momento, hasta alcanzar el nodo que se va a eliminar.

Una vez obtenida la dirección del padre, ya es posible eliminar el elemento de su árbol de columna, aplicando el procedimiento descrito en función del número de hijos del nodo. Conviene observar que en este caso es factible que el nodo tenga 0, 1 o 2 hijos en su árbol de columna, a diferencia de su situación en el árbol de fila, que al recorrerlo en postorden aseguraba la no existencia de hijos.

Tras destruir todos los nodos del árbol de elementos del indice de fila a borrar, será necesario recorrer el árbol de indices de columna para eliminar todos aquellos nodos indices que, como consecuencia de la eliminación de la fila, hayan quedado sin elementos referenciados en su árbol.

Serán aquellos nodos indices cuyo campo *ArbolElem* haya quedado con valor nulo, y corresponderán a aquellas columnas donde existía un solo elemento, y éste se encontraba en la fila eliminada.

Durante el proceso descrito de eliminación de una fila, se ha realizado la supresión de nodos en los siguientes árboles:

- Árbol de índices de fila.
- Árbol de elementos de la fila.
- Arboles de elementos de columna.
- Árbol de índices de columna.

Excepto en el caso del árbol de elementos de la fila, el cual se destruirá por completo, en el resto de árboles puede ocurrir que al suprimir nodos se rompa la condición de equilibrio, por lo cual será necesario restablecerla para seguir manteniendo su estructura de árboles AVL.

Este proceso, implementado el ***ABD*** sobre árboles, presenta frente a la implementación en listas, la ventaja de la búsqueda más rápida del índice de fila.

Por el contrario presenta como desventaja que a la hora de eliminar cada elemento de la fila es necesario conocer la dirección de su padre o antecesor en la columna, lo cual obliga a realizar una búsqueda del padre del elemento a eliminar en el árbol de elementos de dicha columna.

En la implementación mediante listas, al ser estas dobles, se conocía ya la dirección de sus elementos antecesor y sucesor en la columna, al estar almacenadas en el propio nodo elemento.

Una forma de subsanar este problema en la implementación mediante árboles, consiste en incluir en cada nodo un campo puntero adicional que contenga la dirección de su padre, o bien al menos, aprovechar los punteros no utilizados en los nodos que no tienen dos hijos para incluir hebras hacia nodos de nivel superior (árboles enhebrados [KNU86] [LIPS87]).

En el Anexo B.10. se desarrolla en pseudocódigo el algoritmo correspondiente.

3.5.11. Eliminación de una columna

Este proceso consiste en localizar el índice a eliminar en el árbol de índices de columna y proceder a su borrado, suprimiendo también todos los elementos de dicha columna (borrado del árbol de elementos de columna). Como consecuencia de este borrado múltiple, será necesario reorganizar los árboles de elementos de las filas en que se encontraban los nodos eliminados, así como reajustar si procede el árbol de índices de fila.

En principio, habrá que localizar el índice de columna a eliminar *(J)* en el árbol de índices de columna. Para ello se parte del nodo raíz, cuya dirección se encuentra en la variable *ArbolIC*. Se compara su dato de información *InfoIndice* con el índice buscado *(J)*. Si son iguales finaliza el proceso (búsqueda con éxito) y si son distintos se decide en cual de los dos subárboles izquierdo o derecho debe encontrarse el índice *(J)* en caso de existir.

Si el índice buscado *(J)* es menor que el dato del nodo tratado, se avanza hacia el subárbol izquierdo cuya raíz se encuentra en el campo *HijoIzqIndice* del nodo. Por el contrario, si el índice *(J)* es mayor que el dato del nodo evaluado, se avanza hacia la raíz del subárbol derecho mediante la dirección contenida en el campo *HijoDerIndice* del nodo.

Este proceso se repetirá de forma iterativa hasta encontrar el índice (búsqueda con éxito), o bien hasta llegar al final de la rama del árbol por la que se ha descendido sin haberlo encontrado (búsqueda fallida). En este último caso se devolverá una condición de error, indicativa de que el índice de columna se encuentra fuera de rango del conjunto de índices.

En caso de existir se deberá proceder a su eliminación del árbol de índices de columna según el procedimiento ya descrito, en función de que el nodo tenga 0, 1 o 2 hijos. Previamente al borrado se deberá salvar el contenido de su campo *ArbolElem*, que contiene la dirección de la raíz de su árbol de elementos.

A continuación se deberán eliminar todos los nodos que componen el árbol de elementos de la columna en cuestión. Para facilitar este proceso, se definirá recursivamente recorriendo el árbol en postorden de tal forma que el borrado se va aplicando sobre hojas, ya que en dicho tipo de recorrido antes de tratar un nodo se tratan previamente sus hijos izquierdo y derecho. Por lo tanto, se estarán eliminando siempre nodos con 0 hijos lo cual simplifica notablemente el proceso.

Para cada nodo a eliminar en el árbol de elementos de columna, será necesario eliminarlo también de su árbol de elementos de fila, por lo que se necesita conocer la dirección del padre del nodo en el árbol de elementos de su fila.

Para ello, partiendo de la raíz de dicho árbol (su dirección se encuentra en el campo *ArbolElem* del nodo índice referenciado por el campo *ClaveArbolC* del nodo) se va descendiendo de nivel a izquierda o derecha, en base a las comparaciones realizadas sobre el valor de *ClaveArbolF^.InfoIndice*, y registrando la dirección del padre en cada momento, hasta alcanzar el nodo que se va a eliminar.

Una vez obtenida la dirección del padre, ya es posible eliminar el elemento de su árbol de fila, aplicando el procedimiento descrito en función del número de hijos del nodo. Conviene observar que en este caso es factible que el nodo tenga 0, 1 o 2 hijos en su árbol de fila, a diferencia de su situación en el árbol de columna, que al recorrerlo en postorden aseguraba la no existencia de hijos.

Tras destruir todos los nodos del árbol de elementos del índice de columna a borrar, será necesario recorrer el árbol de índices de fila para eliminar todos aquellos nodos índices que, como consecuencia de la eliminación de la columna, hayan quedado sin elementos referenciados en su árbol.

Serán aquellos nodos índices cuyo campo *ArbolElem* haya quedado con valor nulo, y corresponderán a aquellas filas donde existía un solo elemento, y éste se encontraba en la columna eliminada.

Durante el proceso descrito de eliminación de una columna, se ha realizado la supresión de nodos en los siguientes árboles:

- Árbol de índices de columna.
- Árbol de elementos de la columna.
- Arboles de elementos de fila.
- Árbol de índices de fila.

Excepto en el caso del árbol de elementos de la columna, el cual se destruirá por completo, en el resto de árboles puede ocurrir que al suprimir nodos se rompa la condición de equilibrio, por lo cual será necesario restablecerla para seguir manteniendo su estructura de árboles AVL.

Este proceso, implementado el ***ABD*** sobre árboles, presenta frente a la implementación en listas, la ventaja de la búsqueda más rápida del índice de columna.

Por el contrario presenta como desventaja que a la hora de eliminar cada elemento de la columna es necesario conocer la dirección de su padre o antecesor en la fila, lo cual obliga a realizar una búsqueda del padre del elemento a eliminar en el árbol de elementos de dicha fila.

En la implementación mediante listas, al ser estas dobles, se conocía ya la dirección de sus elementos antecesor y sucesor en la fila, al estar almacenadas en el propio nodo elemento.

Una forma de subsanar este problema en la implementación mediante árboles, consiste en incluir en cada nodo un campo puntero adicional que contenga la dirección de su padre, o bien al menos, aprovechar los punteros no utilizados en los nodos que no tienen dos hijos para incluir hebras hacia nodos de nivel superior (árboles enhebrados [KNU86] [LIPS87]).

En el Anexo B.11. se desarrolla en pseudocódigo el algoritmo correspondiente.

3.5.12. Modificación de un índice de fila

Este proceso consiste en reemplazar el valor de un índice (IA) del conjunto de índices de fila por otro valor proporcionado (IN).

Para poder aplicar este tratamiento será necesario que en principio se cumplan dos premisas:

a) Que exista el índice a reemplazar (IA).
b) Que no exista el nuevo índice (IN).

El algoritmo comenzará comprobando las dos premisas expuestas.

Para comprobar la premisa a sería suficiente con aplicar el procedimiento de búsqueda binaria al árbol de índices de fila, cuya raíz se encuentra apuntada por la variable *ArbolIF*.

Se compararía el contenido del campo de información *InfoIndice* de cada nodo índice en que se encuentra posicionado el algoritmo con el valor IA. En caso de ser iguales se da por finalizada la búsqueda (búsqueda con éxito: se cumple la premisa a), y en caso contrario se desciende de nivel en el árbol, avanzando hacia el hijo izquierdo si IA es menor al contenido de *InfoIndice* del nodo, y hacia el hijo derecho en caso de ser mayor. En el caso de que en alguno de los avances se encuentre con el final de la rama se da por finalizada la búsqueda (búsqueda fallida: no se cumple la premisa a).

Para comprobar la premisa b sería también suficiente con aplicar el procedimiento de búsqueda binaria al árbol de índices de fila, cuya raíz se encuentra apuntada por la variable *ArbolIF*.

Se compararía el contenido del campo de información *InfoIndice* de cada nodo índice en que se encuentra posicionado el algoritmo con el valor IN. En caso de ser iguales se da por finalizada la búsqueda (búsqueda con éxito: no se cumple la premisa b), y en caso contrario se desciende de nivel en el árbol, avanzando hacia el hijo izquierdo si IN es menor al contenido de *InfoIndice* del nodo, y hacia el

hijo derecho en caso de ser mayor. En el caso de que en alguno de los avances se encuentre con el final de la rama se da por finalizada la búsqueda (búsqueda fallida: se cumple la premisa b).

Ahora bien, para optimizar los tratamientos posteriores, en caso de cumplirse las dos premisas, es necesario conocer si la posición relativa que deberá ocupar el nuevo índice en la secuencia ordenada del conjunto de índices de fila, es la misma o no a la que ocupaba el índice antiguo.

Por lo tanto, si bien las búsquedas binarias serían el procedimiento más eficaz para la comprobación de ambas premisas, no nos permiten por contra determinar si el cambio del índice de fila implicaría cambios en la posición relativa de los índices de fila en el conjunto ordenado de los mismos.

Por este motivo, se aplicará un recorrido inorden para determinar si se cumple o no la premisa a (existe el índice IA), y en caso afirmativo, determinar también la posición relativa de dicho índice en la secuencia ordenada de los índices de fila.

Asimismo se aplicará un recorrido inorden para determinar si se cumple o no la premisa b (no existe el índice IN), y en caso afirmativo, determinar también la posición relativa que deberá ocupar dicho índice en la secuencia ordenada de los índices de fila. Se utiliza el recorrido inorden porque es el que proporciona, en un árbol binario de búsqueda, una secuencia ordenada de los elementos.

Si las dos premisas se cumplen el tratamiento a realizar dependerá, como ya se ha expuesto, de la posición relativa que debe ocupar el nuevo índice en el conjunto ordenado de índices de fila:

Caso A. *La posición del índice nuevo es la misma que la que ocupaba el índice antiguo.*

Esta situación se producirá cuando se cumpla alguna de las siguientes condiciones:

a) IA es el único índice del conjunto de índices de fila.

b) IA es el menor índice del conjunto de índices de fila y IN es menor que el índice sucesor a IA.

c) IA es el mayor índice del conjunto de índices de fila y IN es mayor que el índice predecesor a IA.

d) IA es un índice intermedio (ni primero ni último) en el conjunto de índices de fila, y IN es mayor que el índice predecesor a IA y menor que el índice sucesor a IA.

En este caso bastará con reemplazar en el campo de información (*InfoIndice*) el valor IA por IN, en el nodo índice cuyo campo de información tiene el valor IA.

A partir de ese momento todos los elementos de la fila referenciada anteriormente por IA, pasan a estar referenciados ahora por IN, sin necesidad de realizar ningún cambio en dichos elementos.

El hecho de no tener que modificar el árbol de elementos de la fila es debido a que en dichos nodos el dato que se almacena en su campo *ClaveArbolC* para registrar el índice de fila a la que pertenecen (el índice de fila se utiliza como criterio de ordenación en sus respectivas columnas), es la dirección del nodo índice y no su dato de información.

Al no modificarse la dirección del nodo índice no es necesario modificar los nodos elementos de dicha fila. Por el contrario, si se hubiese almacenado en el campo *ClaveArbolC* el dato de información del índice, al haberse modificado éste, hubiese sido necesario modificar dicho dato en todos los elementos de la fila.

Caso B. *La posición del índice nuevo es distinta a la que ocupaba el índice antiguo.*

En el caso de que no se cumpla ninguna de las situaciones expuestas en el punto anterior, la posición que debe ocupar el índice nuevo (IN) en el conjunto de

índices de fila es distinta a la que ocupaba el índice antiguo (IA), por lo que habrá que eliminar el índice antiguo del árbol de índices de fila, y crear un nuevo índice con el valor IN en su campo *InfoIndice*.

Tanto para la eliminación del índice antiguo, como para la creación del nuevo, será necesario aplicar previamente los procedimientos de búsqueda binaria de dichos índices, para obtener la dirección de los respectivos padres en el árbol de índices de fila.

Una vez conocida la dirección del padre del índice antiguo en el árbol de índices de fila, se puede proceder a su eliminación del mismo, aplicando el algoritmo de borrado en función de que el nodo índice tenga 0, 1 o 2 hijos.

Igualmente, una vez conocida la dirección del padre del índice nuevo en el árbol de índices de fila, se puede proceder a insertarlo en el mismo (quedará como hoja), como hijo izquierdo o derecho del padre, según el valor de IN sea menor o mayor al valor del índice padre.

Como consecuencia del cambio de valor y posición del nodo índice de fila, los elementos pertenecientes a dicha fila pueden sufrir cambios en la columna que ocupan, por lo cual será necesario reajustar los árboles de las columnas en las que se encuentran los elementos de la fila.

Por lo tanto, será necesario recorrer al completo el árbol de elementos de la fila y cada elemento desengancharlo de su árbol de columna, y engancharlo de nuevo en base al nuevo valor del índice de fila, que es el criterio de ordenación en el árbol de cada columna.

Al tener que tratar todos los elementos del árbol de la fila y no tener ninguna trascendencia el orden en que se realice dicho proceso, se puede utilizar cualquier criterio de recorrido de dicho árbol.

Para cada elemento, se deberá obtener la dirección de su padre en el árbol de elementos de su columna y proceder a desengancharlo de dicho árbol, realizando el tratamiento correspondiente, según tenga 0, 1 o 2 hijos.

Una vez desligado de su árbol de columna, es necesario volver a engancharlo en dicho árbol en su posición adecuada en función del nuevo valor del índice de fila. Para ello se deberá obtener la dirección de su nodo padre en dicho árbol.

Como al insertar un nodo en un árbol binario de búsqueda siempre queda como hoja, se deberá asignar el valor nulo a los campos *HijoIzqArbolC* y *HijoDerArbolC*, que contienen las direcciones del hijo izquierdo y derecho del nodo respectivamente en el árbol de columna. Además se deberá asignar al campo *ClaveArbolC* la dirección del nuevo índice de fila creado (puesto que se ha generado nuevo, ya que se ha eliminado el índice antiguo).

Si el elemento era el único de su columna, al engancharlo al árbol de elementos de su columna, quedará como raíz del mismo, por lo que habrá que asignar su dirección al campo *ArbolElem* del nodo índice de columna.

En caso contrario, habrá que engancharlo a su padre como hijo izquierdo o derecho según el valor del nuevo índice de fila sea menor o mayor que el de su nodo padre.

Si bien durante este proceso no varía el número de nodos índices ni el de nodos elementos, se realizan en el árbol de índices de fila y en los árboles de elementos de columna eliminaciones y nuevas inserciones, por lo que puede ocurrir que se rompa la condición de equilibrio en dichos árboles, lo que obligará a restablecerla para seguir manteniendo su estructura de árboles AVL.

Este proceso, implementado el ***ABD*** sobre árboles, presenta frente a la implementación en listas, la ventaja de las búsquedas más rápidas de los índices de fila y de las nuevas posiciones a ocupar por los elementos en sus columnas respectivas.

Por el contrario presenta como desventaja que a la hora de desenganchar cada nodo en su árbol de columna es necesario conocer la dirección de su padre en dicho árbol, lo cual obliga a realizar una búsqueda del mismo.

En la implementación mediante listas, al ser estas dobles, se conocía ya la dirección de sus elementos antecesor y sucesor en la columna, al estar almacenadas en el propio nodo.

Como ya se ha expuesto, una forma de subsanar este problema en la implementación mediante árboles, consiste en incluir en cada nodo un campo puntero adicional que contenga la dirección de su padre, o bien al menos, aprovechar los punteros no utilizados en los nodos que no tienen dos hijos para incluir hebras hacia nodos de nivel superior (árboles enhebrados [KNU86][LIPS87]).

En el Anexo B.12. se desarrolla en pseudocódigo el algoritmo correspondiente.

3.5.13. Modificación de un indice de columna

Este proceso consiste en reemplazar el valor de un índice (JA) del conjunto de índices de columna por otro valor proporcionado (JN).

Para poder aplicar este tratamiento será necesario que en principio se cumplan dos premisas:

a) Que exista el índice a reemplazar (JA).
b) Que no exista el nuevo índice (JN).

El algoritmo comenzará comprobando las dos premisas expuestas.

Para comprobar la premisa a sería suficiente con aplicar el procedimiento de búsqueda binaria al árbol de índices de columna, cuya raíz se encuentra apuntada por la variable *ArbolIC*.

Se compararía el contenido del campo de información *InfoIndice* de cada nodo índice en que se encuentra posicionado el algoritmo con el valor JA. En caso de ser iguales se da por finalizada la búsqueda (búsqueda con éxito: se cumple la

premisa a), y en caso contrario se desciende de nivel en el árbol, avanzando hacia el hijo izquierdo si JA es menor al contenido de *InfoIndice* del nodo, y hacia el hijo derecho en caso de ser mayor. En el caso de que en alguno de los avances se encuentre con el final de la rama se da por finalizada la búsqueda (búsqueda fallida: no se cumple la premisa a).

Para comprobar la premisa b sería también suficiente con aplicar el procedimiento de búsqueda binaria al árbol de índices de columna, cuya raiz se encuentra apuntada por la variable *ArbolIC*.

Se compararía el contenido del campo de información *InfoIndice* de cada nodo índice en que se encuentra posicionado el algoritmo con el valor JN. En caso de ser iguales se da por finalizada la búsqueda (búsqueda con éxito: no se cumple la premisa b), y en caso contrario se desciende de nivel en el árbol, avanzando hacia el hijo izquierdo si JN es menor al contenido de *InfoIndice* del nodo, y hacia el hijo derecho en caso de ser mayor. En el caso de que en alguno de los avances se encuentre con el final de la rama se da por finalizada la búsqueda (búsqueda fallida: se cumple la premisa b).

Ahora bien, para optimizar los tratamientos posteriores, en caso de cumplirse las dos premisas, es necesario conocer si la posición relativa que deberá ocupar el nuevo índice en la secuencia ordenada del conjunto de índices de columna, es la misma o no a la que ocupaba el índice antiguo.

Por lo tanto, si bien las búsquedas binarias serían el procedimiento más eficaz para la comprobación de ambas premisas, no nos permiten por contra determinar si el cambio del índice de columna implicaría cambios en la posición relativa de los índices de columna en el conjunto ordenado de los mismos.

Por este motivo, se aplicará un recorrido inorden para determinar si se cumple o no la premisa a (existe el índice JA), y en caso afirmativo, determinar también la posición relativa de dicho índice en la secuencia ordenada de los índices de columna.

Asimismo se aplicará un recorrido inorden para determinar si se cumple o no la premisa b (no existe el índice JN) y en caso afirmativo, determinar también la posición relativa que deberá ocupar dicho índice en la secuencia ordenada de los

índices de columna. Se utiliza el recorrido inorden porque es el que proporciona, en un árbol binario de búsqueda, una secuencia ordenada de los elementos.

Si las dos premisas se cumplen el tratamiento a realizar dependerá, como ya se ha expuesto, de la posición relativa que debe ocupar el nuevo índice en el conjunto ordenado de índices de columna:

Caso A. *La posición del índice nuevo es la misma que la que ocupaba el índice antiguo.*

Este caso ocurrirá cuando se cumpla alguna de las siguientes condiciones:

a) JA es el único índice del conjunto de índices de columna.

b) JA es el menor índice del conjunto de índices de columna y JN es menor que el índice sucesor a JA.

c) JA es el mayor índice del conjunto de índices de columna y JN es mayor que el índice predecesor a JA.

d) JA es un índice intermedio (ni primero ni último) en el conjunto de índices de columna, y JN es mayor que el índice predecesor a JA y menor que el índice sucesor a JA.

En este caso bastará con reemplazar en el campo de información (*InfoIndice*) el valor JA por JN, en el nodo índice cuyo campo de información tiene el valor JA.

A partir de ese momento todos los elementos de la columna referenciada anteriormente por JA, pasan a estar referenciados ahora por JN, sin necesidad de realizar ningún cambio en dichos elementos.

El hecho de no tener que modificar el árbol de elementos de la columna es debido a que en dichos nodos el dato que se almacena en su campo *ClaveArbolF* para registrar el índice de columna a la que pertenecen (el índice de columna se utiliza como criterio de ordenación en sus respectivas filas), es la dirección del nodo índice y no su dato de información.

Al no modificarse la dirección del nodo índice no es necesario modificar los nodos elementos de dicha columna. Por el contrario, si se hubiese almacenado en el campo *ClaveArbolF* el dato de información del índice, al haberse modificado éste, hubiese sido necesario modificar dicho dato en todos los elementos de la columna.

Caso B. *La posición del índice nuevo es distinta a la que ocupaba el índice antiguo.*

En el caso de que no se cumpla ninguna de las situaciones expuestas en el punto anterior, la posición que debe ocupar el índice nuevo (JN) en el conjunto de índices de columna es distinta a la que ocupaba el índice antiguo (JA), por lo que habrá que eliminar el índice antiguo del árbol de índices de columna, y crear un nuevo índice con el valor JN en su campo *InfoIndice*.

Tanto para la eliminación del índice antiguo, como para la creación del nuevo, será necesario aplicar previamente los procedimientos de búsqueda binaria de dichos índices, para obtener la dirección de los respectivos padres en el árbol de índices de columna.

Una vez conocida la dirección del padre del índice antiguo en el árbol de índices de columna, se puede proceder a su eliminación del mismo, aplicando el algoritmo de borrado en función de que el nodo índice tenga 0, 1 o 2 hijos.

Igualmente, una vez conocida la dirección del padre del índice nuevo en el árbol de índices de columna, se puede proceder a insertarlo en el mismo (quedará como hoja), como hijo izquierdo o derecho del padre, según el valor de JN sea menor o mayor al valor del índice padre.

Como consecuencia del cambio de valor y posición del nodo índice de columna, los elementos pertenecientes a dicha columna pueden sufrir cambios en la fila que ocupan, por lo cual será necesario reajustar los árboles de las filas en las que se encuentran los elementos de la columna

Por lo tanto, será necesario recorrer al completo el árbol de elementos de la columna y cada elemento desengancharlo de su árbol de fila, y engancharlo de nuevo en base al nuevo valor del índice de columna, que es el criterio de ordenación en el árbol de cada fila.

Al tener que tratar todos los elementos del árbol de la columna y no tener ninguna trascendencia el orden en que se realice dicho proceso, se puede utilizar cualquier criterio de recorrido de dicho árbol.

Para cada elemento, se deberá obtener la dirección de su padre en el árbol de elementos de su fila y proceder a desengancharlo de dicho árbol, realizando el tratamiento correspondiente, según tenga 0, 1 o 2 hijos.

Una vez desligado de su árbol de fila, es necesario volver a engancharlo en dicho árbol en su posición adecuada en función del nuevo valor del índice de columna. Para ello se deberá obtener la dirección de su nodo padre en dicho árbol.

Como al insertar un nodo en un árbol binario de búsqueda siempre queda como hoja, se deberá asignar el valor nulo a los campos *HijoIzqArbolF* y *HijoDerArbolF*, que contienen las direcciones del hijo izquierdo y derecho del nodo respectivamente en el árbol de fila. Además se deberá asignar al campo *ClaveArbolF* la dirección del nuevo índice de columna creado (puesto que se ha generado nuevo, ya que se ha eliminado el índice antiguo).

Si el elemento era el único de su fila, al engancharlo al árbol de elementos de su fila, quedará como raíz del mismo, por lo que habrá que asignar su dirección al campo *ArbolElem* del nodo índice de fila.

En caso contrario, habrá que engancharlo a su padre como hijo izquierdo o derecho según el valor del nuevo índice de columna sea menor o mayor que el de su nodo padre.

Si bien durante este proceso no varía el número de nodos índices ni el de nodos elementos, se realizan en el árbol de índices de columna y en los árboles de elementos de fila eliminaciones y nuevas inserciones, por lo que puede ocurrir que

se rompa la condición de equilibrio en dichos árboles, lo que obligará a restablecerla para seguir manteniendo su estructura de árboles AVL.

Este proceso, implementado el ***ABD*** sobre árboles, presenta frente a la implementación en listas, la ventaja de las búsquedas más rápidas de los índices de columna y de las nuevas posiciones a ocupar por los elementos en sus filas respectivas.

Por el contrario presenta como desventaja que a la hora de desenganchar cada nodo en su árbol de fila es necesario conocer la dirección de su padre en dicho árbol, lo cual obliga a realizar una búsqueda del mismo.

En la implementación mediante listas, al ser estas dobles, se conocía ya la dirección de sus elementos antecesor y sucesor en la fila, al estar almacenadas en el propio nodo.

Como ya se ha expuesto anteriormente, una forma de subsanar este problema en la implementación mediante árboles, consiste en incluir en cada nodo un campo puntero adicional que contenga la dirección de su padre, o bien al menos, aprovechar los punteros no utilizados en los nodos que no tienen dos hijos para incluir hebras hacia nodos de nivel superior (árboles enhebrados [KNU86] [LIPS87]).

En el Anexo B.13. se desarrolla en pseudocódigo el algoritmo correspondiente.

4. CONCLUSIONES

4.1. ANÁLISIS DE LAS APORTACIONES ORIGINALES DE LA INVESTIGACIÓN

El **Array Bidimensional Dinámico (ABD)** diseñado en esta investigación se plantea como una alternativa al array bidimensional clásico, estructura sencilla y ampliamente utilizada sobre todo en aplicaciones matemáticas, pero que presenta una serie de inconvenientes:

- Naturaleza estática debido a la necesidad de definir sus conjuntos de indices, y en consecuencia su tamaño, previamente a la introducción de los elementos.

- Desperdicio de espacio en memoria en matrices poco densas, y deficiente rendimiento en operaciones que implican un recorrido completo de la estructura.

- Finalización anormal o resultados imprevistos de los programas por indices fuera de rango.

La matriz o array bidimensional tradicional es la estructura más utilizada para almacenar información en disposición tabular, permitiendo un acceso rápido y directo a cada elemento, y manteniendo la diferenciación entre sus componentes por filas y columnas.

El *ABD* es un tipo abstracto de datos construido en base a apuntadores, lo cual le confiere un carácter dinámico que permite eliminar los inconvenientes del array clásico y aportar ventajas adicionales.

En una primera implementación de la estructura se ha hecho uso de listas enlazadas ortogonales, consistentes en una red de listas dobles con inserción al final entrecruzadas para representar los elementos, cada uno de los cuales forma parte de una lista de fila y de una lista de columna, y listas enlazadas simples para registrar los índices.

Si bien con este diseño se consiguen superar los inconvenientes de los arrays tradicionales planteados en un principio, la estructura presenta una limitación derivada de la utilización de listas enlazadas: los procesos de búsqueda de índices y elementos, intervinientes en las operaciones de manipulación del TAD, son secuenciales, y por consiguiente, de orden O(n). Esto implica que las operaciones a realizar se verán afectadas de forma proporcional al número de elementos a tratar. El hecho de que las listas enlazadas de índices y elementos estén ordenadas reduce al menos los tiempos de proceso en las búsquedas fallidas.

Con el fin de mejorar notablemente los procesos de búsqueda, reduciendo la dependencia lineal que presentan respecto al número de elementos a tratar, se ha desarrollado una segunda versión de la estructura basada en árboles binarios de búsqueda balanceados AVL.

La matriz consta, en este caso, de una red de árboles binarios AVL entrelazados, donde cada elemento forma parte de dos árboles, uno el de su fila que estará ordenado por el índice de columna, y otro el de su columna el cual se ordenará por índice de fila. Además existirán dos árboles de índices, uno por cada dimensión, ordenados por los valores de dichos índices.

Con esta disposición se asegura que los tratamientos de búsqueda se ajustan al orden $O(\log_2 n)$, con lo que se conseguirá disminuir los tiempos de proceso de gran parte de las operaciones de manejo del TAD.

La elección de árboles equilibrados AVL, y no simples árboles binarios de búsqueda, se debe a la necesidad de minimizar las búsquedas, garantizando que en todos los casos se ajustan al orden logarítmico en base 2, ya que en caso de no estar balanceado el árbol, el promedio de búsqueda aumenta un 39%.

Hay que advertir, sin embargo, del coste en tiempo que supone mantener balanceado el árbol, al aplicar los procedimientos de rotación propuestos por Adelson-Velskii y Landis [ADVL62] en las inserciones y borrados de nodos que rompan la condición de equilibrio del árbol.

El uso de este tipo de árboles está justificado siempre que los accesos a la estructura sean más numerosos que las inserciones y borrados, lo cual es muy frecuente, sobre todo en matrices matemáticas.

Conviene destacar que el cambio de implementación, al sustituir listas enlazadas por árboles binarios, implica adaptar las operaciones a la nueva implementación (visión interna del TAD), pero no afecta a la utilización que del ***ABD*** hagan los programas de usuario, puesto que el interfaz (visión externa) sigue siendo el mismo.

Esta característica, cualidad destacable y prioritaria de un tipo abstracto de datos, se puede comprobar en el Anexo C, donde aparecen codificados en Pascal en forma de unidades el TAD ***ABD*** implementado por medio de listas enlazadas (ABDL) , y el TAD ***ABD*** implementado por medio de árboles binarios (ABDA). El interfaz es idéntico en las dos unidades, por lo que el programa menú diseñado para manejar las operaciones es válido para ambas.

Por lo tanto, y concluyendo, se ha desarrollado un tipo abstracto de datos, denominado **Array Bidimensional Dinámico (ABD)**, destinado a almacenar información en disposición matricial, de naturaleza dinámica al estar implementado por estructuras basadas en apuntadores: listas enlazadas (primera versión) y árboles binarios AVL (versión mejorada).

La extrapolación de esta estructura bidimensional a una n-dimensional parece una deducción lógica y factible de implementar.

Entre las características más relevantes y diferenciadoras del ***ABD***, frente al array tradicional estático, destacan las siguientes:

a) Naturaleza dinámica y flexible. Los índices y los elementos se crean, modifican y destruyen de forma dinámica en ejecución, a medida que se requieren.

b) No es necesario conocer y definir los conjuntos de índices de filas y columnas, y por tanto las dimensiones del array, previamente a la incorporación de los elementos a la estructura.

c) No se precisa que los conjuntos de índices tengan que ser series de elementos ni ordinales, ni consecutivos, ni del mismo tipo en filas y columnas.

d) Control del error producido por un acceso inadecuado con índices fuera de rango, sin abortar la ejecución del programa ni generar resultados inesperados e incorrectos.

e) Economía de espacio en matrices poco densas. Sólo utiliza la memoria que requieren los elementos presentes, ya que no reserva espacio para las posiciones no utilizadas de la matriz.

f) Como consecuencia de lo anterior, mejor rendimiento en operaciones basadas en recorridos. En este tipo de matrices con bajo porcentaje de ocupación es donde la estructura propuesta alcanza su máxima expresión.

g) Permite acceder a un elemento en particular en base a sus índices y permite procesar filas y columnas aisladas de forma independiente, propiedades más destacables de los arrays estáticos tradicionales.

4.2. LÍNEAS FUTURAS DE INTERÉS

El tiempo de acceso por índices a un elemento depende de la posición del elemento y del grado de ocupación de la matriz, a diferencia de los arrays clásicos, cuya mayor virtud es el acceso casi inmediato e idéntico para todos los elementos e independiente del grado de ocupación. Por ello se dice que los arrays son conjuntos indexados.

Un objetivo importante a lograr consistiría en aproximar el tiempo de acceso a un elemento referenciado por sus índices al de un array estático. Esta aportación implicaría no sólo la optimización de los algoritmos correspondientes, sino también un probable cambio estructural.

Asimismo, en matrices muy densas, el espacio que se requiere para mantener los apuntadores, adicional al estrictamente necesario para almacenar la información, es considerable. Se deberían idear nuevas técnicas que, a partir de un determinado umbral de ocupación, permitiesen reducir el espacio utilizado.

En la implementación por medio de árboles, como se ha apuntado, habría que introducir una variante para mejorar aún más determinadas operaciones, que consistiría en incluir apuntadores desde cada nodo hacia su padre, o al menos utilizar árboles enhebrados para conseguir un acceso más rápido desde cada nodo a su predecesor, tratamiento necesitado con frecuencia en los algoritmos.

5. BIBLIOGRAFÍA

[ADVL62] ADELSON-VELSKII, G.M. y LANDIS, E.M.
Arboles AVL.
Dok. Akad. Nauk SSSR, 146, pp. 263-6; English traslation in Soviet Math., 3, 1259-63. 1962.

[AHO88] AHO, A.V. ; HOPCROFT, J.E. y ULLMAN, J.D.
Estructuras de Datos y Algoritmos.
Addison-Wesley, 1988.

[AND90] ANDERSSON, A. y LAI, T.W.
Fast Updating of Well-Balanced Trees.
Proceedings of the second Scandinavian workshop. Bergen, Sweden, Julio-1990.

[BANA91] BANACHOWSKY, L.; KRECZMAN, A. y WOJCIECH R
Analysis of Algorithms and Data Structures.
Addison-Wesley, 1991.

[BIER93] BIERNAT, Martin J.
Teaching Tools for Data Structures and Algorithms.
ACM Sigcse Bulletin, Vol.25, N° 4, 9-17, 1993.

[BIS89] BISHOP, Judy.
Abstracción de Datos.
Paraninfo, 1989.

[BRAS90] BRASSARD, G. y BRATLEY, P.
Algoritmica.
Masson, 1990

[CAI93] CAIRÓ, O. y GUARDATI, S.
Estructuras de Datos.
McGraw-Hill, 1993.

[COL78] COLEMAN, Derek.
Organización de Datos y Programación Estructurada.
Gustavo Gili, 1978.

[DALE86] DALE, Nell y LILLY Susan C.
Pascal y Estructura de Datos.
McGraw-Hill, 1986.

[FOLK92] FOLK, M.J. y ZOELLICK, B.
Estructuras de Archivos.
Addison-Wesley, 1992.

[GAL93] GALVE, J.; GONZALEZ, J.C.; SANCHEZ, A. y VELAZQUEZ, J.A.
Algoritmica. Diseño y Análisis de Algoritmos Funcionales e Imperativos.
Ra-ma, 1993.

[GOT78] GOTLIEB, C.C. y GOTLIEB, L.R.
Data Types and Structures.
Prentice-Hall, 1978.

[HOA72] HOARE, C.A.R.
Structured Programming.
Academic Press, 1972.

[HORO82] HOROWITZ, E. y SAHNI, S.
Fundamentals of Data Structures.
Computer Science Press, 1982.

[JARC94] JARC, Duane J.
Data Structures: A Unified View.
ACM Sigcse Bulletin, Vol.26, Nº 2, 2-4, 1994.

[JOYA88] JOYANES AGUILAR, Luis.
Fundamentos de Programación. Algoritmos y Estructura de Datos.
McGraw-Hill, 1988.

[KLE90] KLEIN, R. y WOOD, D.
A Tight Upper Bound for the Path Length of AVL Trees.
Theoretical Computer Science, Vol.72, 2, 3, 251-264, 1990.

[KHO90] KHOSRAVIYANI, Firooz.
Using Binary Search on a Linked List.
ACM Sigcse Bulletin, Vol.22, Nº 3, 7-10, 1990.

[KHO91] KHOSRAVIYANI, F. ; MOADAB, M.H. y HALE, D.F.
Time Distribution Analysis for Binary Search of a Linked List.
ACM Sigcse Bulletin, Vol.23, Nº 4, 7-12, 1991.

[KIN92] KING, Todd.
Dynamic Data Structures. Theory and Aplication.
Academic Press, 1992.

[KING90] KINGSTON, Jeffrey H.
Algorithms and Data Structures. Design, Correctness, Analysis.
Addison-Wesley, 1990.

[KNU86] KNUTH, Donald E.
El Arte de Programar Ordenadores, Vol I. Algoritmos Fundamentales.
Ed. Reverté, 1986.

[KNU87] KNUTH, Donald E.
El Arte de Programar Ordenadores, Vol III. Clasificación y Búsqueda.
Ed. Reverté, 1987.

[KRU88] KRUSE, Robert L.
Estructura de Datos y Diseño de Programas.
Prentice-Hall, 1988.

[LAB93] LABROCCA, P.J.
Dynamic Two-Dimensional Arrays.
C Users Journal Vol.11, Nº 11, 72-86, 1993.

[LEWI85] LEWIS, T.G. y SMITH, M.Z.
Estructuras de Datos.
Paraninfo, 1985.

[LIPS87] LIPSCHUTZ, Seymour.
Estructura de Datos.
McGraw-Hill, 1987.

[MEHL84] MEHLHORN, Kurt.
Data Structures and Algorithms 1: Sorting and Searching.
Springer-Verlag, 1984.

[MEHL93] MEHLHORN, K. y TSAKALIDIS, A.
Dynamic Interpolation Search.
Journal of the ACM, Vol.40, Nº 3, 621-634, 1993.

[OTT90] OTTMANN, T. y WOOD, D.
How to Update a Balanced Binary Tree with a constant number of rotations.
Proceedings of the second Scandinavian workshop. Bergen, Sweden, Julio-1990.

[PUGH90] PUGH, William.
Skip Lists: a Probabilistic Alternative to Balanced Trees.
Communications of the ACM, Vol.33, Nº 6, 668-676, 1990.

[REIN83] REINGOLD, E.M. y HANSEN W.J.
Data Structures.
Little, Brown and Company, 1983.

[ROSE88] ROSENSTEIN, Milton.
Data Structures for Programmers.
John Wiley & Sons, 1988.

[SAL94] SALMON, William I.
Structures and Abstractions.
Irwin, 1994.

[SCH94] SCHNEIER, Bruce.
Skip Lists.
Dr. Dobb's Journal, Vol.19, Nº 1, 50-52, 1994.

[SPRU92] SPRUGNOLI, R.
The Generation of Binary Trees as a Numerical Problem.
Journal of the ACM, Vol.39, Nº 2, 317-325, 1992.

[STAN80] STANDISH, Thomas A.
Data Structure Techniques.
Addison-Wesley, 1980.

[TEN86] TENENBAUM, A.M. y AUGENSTEIN, M.J.
Data Structures using Pascal (2nd. Edition).
Prentice-Hall, 1986.

[TEW73] TEWARSON, R.P.
Sparse Matrices.
Academic Press, 1973.

[VANA89] VAN AMSTEL, J.J. y POIRTERS, J.A.A.M.
The Design of Data Structures and Algorithms.
Prentice-Hall, 1989.

[VAND74] VAN DOREN, J. y GRAY, J.
An Algorithm for Maintaining Dynamic AVL Trees.
Information Systems, COINS IV, Nueva York, Plenum Press, 1974.

[VANK93] VAN KREVELD, M.J. y OVERMANS, M.H.
Union-Copy Structures and Dynamic Segment Trees.
Journal of the ACM, Vol.40, Nº 3, 635-639, 1993.

[VOLK93] VOLKMAN, Victor R.
Update to TAVLTree Library.
C Users Journal, Vol.11, Nº 12, 111-112, 1993.

[WIR76] WIRTH, Niklaus.
Algorithms + Data Structures = Programs.
Prentice-Hall, 1976.

[WIR87] WIRTH, Niklaus.
Algoritmos y Estructura de Datos.
Prentice-Hall, 1987.

[WOOD93] WOOD, Derick.
Data Structures, Algorithms, and Performance.
Addison-Wesley, 1993.