5 ÁRBOLES DE ORDEN N Y GENERALES.

En el tema anterior se presentaron el concepto de árbol y sus distintas clases y se estudiaron los árboles binarios en sus diferentes tipos. Son numerosas las situaciones en que un árbol binario no puede representar correctamente la naturaleza del problema y hemos de recurrir a un árbol de orden N o a uno general. Por ejemplo, los árboles de juegos representan las distintas jugadas que se pueden hacer en una partida de un juego desde una situación dada. Esta situación corresponde al nodo raíz. Por cada posible jugada tendremos un nuevo hijo para ese nodo. Si pensamos en el ajedrez, con una disposición de las fichas el jugador al que le toque mover podrá elegir entre diferentes piezas y diferentes movimientos. Cada jugada distinta genera un nuevo hijo de la jugada actual. A su vez, estas nuevas jugadas tendrán un número diferente de nuevas posibilidades de juego.

En este tema nos centraremos en el estudio de los árboles en los que cada nodo puede tener más de dos nodos hijos. Si todos los nodos con hijos tienen que tener el mismo número de hijos, se llamarán *árboles de orden N* y si cada nodo puede tener un número distinto de hijos, se denominarán *árboles generales*. Según esta definición, un árbol binario es un árbol de orden N con N = 2. Finalmente, se estudiará un tipo de árbol equilibrado que asegura una altura mínima y, por tanto, un algoritmo de búsqueda eficiente. Estos árboles, denominados 2-3, tienen la particularidad de almacenar más de un elemento en cada nodo. Como generalización del concepto de árbol 2-3 se presentan los árboles B, usados en la organización rápida de información almacenada en dispositivos de acceso lento.

5.1 Árboles de orden N.

Formalmente, un árbol de orden N (con $N \geq 2$), con nodos de tipo T, es una lista vacía (correspondiente al caso del árbol vacío) o un par (r, L_A), formado por un nodo r, la **raíz**, y una tupla L_A (**bosque**), con N árboles, exactamente, del mismo tipo (los **subárboles**, o **hijos**, de la raíz). En este caso, suele escribirse explícitamente (r, A_1, A_2, ..., A_n).

El número de elementos de un bosque es predeterminado y fijo. Aunque en un bosque pueda haber árboles vacíos, nunca podremos borrar un componente de un bosque sin poner otro en su lugar.

5.1.1 Especificación de árboles de orden N.

Como generadores de este tipo contaremos con dos operaciones:

1. CrearÁrbolN: Operación constante que genera un árbol de orden N vacío.

2. Árbol_Orden_N: Operación que, dado un nodo y un bosque de árboles de orden N, genera un árbol de orden N cuya raíz es el nodo y los árboles del bosque son sus hijos.

Tendremos también funciones de acceso a cada una de las partes del árbol, tanto para la raíz, como para cada uno de los hijos que forman el bosque:

1. Raíz: Operación que devuelve el nodo raíz del árbol. Será una función parcial, pues no podrá aplicarse a árboles vacíos.

2. Hijo: Operación que devuelve el iésimo hijo de un árbol. También será una operación parcial que no podrá aplicarse sobre árboles vacíos. Una opción alternativa consiste en definir una función de acceso para cada hijo (PRIMERO, SEGUNDO, TERCERO, etc.). Esta es la opción usada en los árboles binarios con las funciones *Hijo_izq* e *Hijo_dch*.

Como en otros tipos, definiremos la operación Es_Vacío, que nos dirá si un árbol de orden N está vacío o no.

Como nuestro tipo se basa en otro tipo, el **bosque**, que no es más que una lista de árboles, habrá que definir las operaciones de este tipo. Como generadores tendremos:

1. BosqueNulo: Operación que genera un bosque con *n* árboles de orden N vacíos.

2. ActComp: Operación que genera un bosque con el valor del iésimo árbol de orden N actualizado.

3. Comp: Operación que devuelve el iésimo hijo del bosque.

Por tanto la especificación quedaría:

tipo BOSQUEN
dominios N, ARBOLN
generadores
CrearBosqueN: $\longrightarrow$ BOSQUEN
ActComp: BOSQUEN $\times$ N $\times$ ARBOLN $\not\longrightarrow$ BOSQUEN
selectores
Comp: BOSQUEN $\times$ N $\not\longrightarrow$ ARBOLN
auxiliares
TMáx: $\longrightarrow$ N
precondiciones B: BOSQUEN; i: N; A: ARBOLN
pre ActComp(B, i, A): $1 \leq i \leq$ TMáx
pre Comp(B, i): $1 \leq i \leq$ TMáx
ecuaciones B: BOSQUEN; i1, i2: N; A1, A2: ARBOLN
TMáx == $suc^n(0)$
ActComp(ActComp(B, i1, A1), i2, A2) ==
SI i1 = i2 ENTONCES
ActComp(B, i2, A2)
SI NO
ActComp(ActComp(B, i2, A2), i1, A1)
Comp(CrearBosqueN, i) == CrearÁrbolN
Comp(ActComp(B, i1, A), i2) == SI i1 = i2 ENTONCES
A
SI NO
Comp(B, i2)
fin

tipo ARBOLN
dominios ELEMENTO, LOGICO, N, BOSQUEN
generadores
CrearÁrbolN: ⟶ ARBOLN
Árbol_Orden_N: ELEMENTO × BOSQUEN ⟶ ARBOLN
constructores
Hijo: N × ARBOLN —/→ ARBOLN
selectores
Raíz: ARBOLN —/→ ELEMENTO
Es_Vacío: ARBOLN ⟶ LOGICO
auxiliares
TMáx: ⟶ N
precondiciones A: ARBOLN; i: N;
pre Raíz(A): not Es_Vacío(A)
pre Hijo(A, i): not Es_Vacío(A) and $1 \leq i \leq$ TMáx
ecuaciones A: ARBOLN; B: BOSQUEN; i, i1, i2: N; e: ELEMENTO;
TMáx == $suc^n(0)$
Es_Vacío(CrearÁrbolN) == V
Es_Vacío(Árbol_Orden _N(e, B)) == F
Raíz(Árbol_Orden_N(e, B)) == e
Hijo(i, Árbol_Orden_N(e, B)) == Comp(B, i)
fin

A este definición básica se pueden añadir cuantas operaciones resulten necesarias para desarrollar toda la potencia del tipo. Cada operación que se defina sobre el árbol requerirá la definición de una operación análoga para el tipo BOSQUEN.

Igual_FormaA es una función que devuelve VERDADERO cuando dos árboles de orden N tienen la misma forma, aunque los elementos sean diferentes.

Igual_FormaA: ARBOLN × ARBOLN ⟶ LOGICO
Igual_FormaB: N × BOSQUEN × BOSQUEN —/→ LOGICO
pre Igual_FormaB(i, B): $1 \leq i \leq$ TMáx

Igual_FormaA(CrearÁrbolN, A) == Es_Vacío(A)
Igual_FormaA(Árbol_Orden_N(e, B), CrearÁrbolN) == F
Igual_FormaA(Árbol_Orden_N(e_1, B_1), Árbol_Orden_N(e_2, B_2)) ==
Igual_FormaB(1, B_1, B_2)

Igual_FormaB(i, B_1, B_2) == SI i = N ENTONCES
Igual_FormaA(Comp(B_1,N), Comp(B_2, N))
SI NO
Igual_FormaA(Comp(B_1,i), Comp(B_2, i)) AND
Igual_FormaB(suc(i), B_1, B_2))

Número_NodosA es una función que devuelve el número de elementos de un árbol.
Número_NodosA: ARBOLN ⟶ N

Número_NodosB: N × BOSQUEN —/→ N
pre Número_NodosB(i, B): 1 ≤ i ≤ tMáx
Número_NodosA(CrearÁrbolN) == 0
Número_NodosA(Árbol_Orden_N(e, B)) == 1 + Número_NodosB(1, B)

```
Número_NodosB(i, B) ==
SI i = N ENTONCES
        Número_NodosA(Comp(B,N))
    SI NO
        Número_NodosA(Comp(B, i)) + Número_NodosB(suc(i), B)
```

AlturaA es una función que devuelve la máxima longitud de un camino de la raíz a una hoja.

AlturaA: ARBOLN ⟶ N
AlturaB: N × BOSQUEN —/→ N
Máximo: N ⟶ N
pre AlturaB(i, B): 1 ≤ i ≤ TMáx

```
Máximo(m,n) ==      SI m > n ENTONCES
                        m
                    SI NO
                        n
```

AlturaA(CrearÁrbolN) == 0
AlturaA(Árbol_Orden_N(e, B)) == 1 + AlturaB(B)

```
AlturaB(B) == SI i = N ENTONCES
                    AlturaA(Comp(B,N))
              SI NO
                    Máximo(AlturaA(Comp(B, i)), AlturaB(suc(i), B))
```

EstáA es una función que devuelve VERDADERO cuando un elemento está presente en un árbol.

EstáA: ELEMENTO × ARBOLN ⟶ LOGICO
EstáB: N × ELEMENTO × BOSQUEN —/→ LOGICO
pre EstáB(i, B): 1 ≤ i ≤ TMáx
EstáA(e, CrearÁrbolN) == F
EstáA(e, Árbol_Orden_N(r, B)) == (e = r) OR EstáB(1, e, B)

```
EstáB(i, e, B) ==       SI  i = N ENTONCES
EstáA(e, Comp(B, N))
                        SI NO
                            EstáA(e,Comp(B,i)) OR EstáB(suc(i), e, B)
```

IgualA es una función que devuelve VERDADERO cuando dos árboles de orden N tienen la misma forma, y los mismos elementos en las mismas posiciones. En este caso el tipo ya no es genérico, ya que exigimos una relación de equivalencia para el tipo ITEM.

IgualA: ARBOLN × ARBOLN ⟶ LOGICO
IgualB: N × BOSQUEN × BOSQUEN —/→ LOGICO
pre IgualB(i, B): 1 ≤ i ≤ TMáx

IgualA(CrearÁrbolN, A) == Es_Vacío(A)
IgualA(Árbol_Orden_N(e, A), CrearÁrbolN) == F
IgualA(Árbol_Orden_N(e_1, B_1), Árbol_Orden_N(e_2, B_2)) ==
    (e_1 = e_2) AND IgualB(1, B_1, B_2)

IgualB(i, B_1, B_2) == SI i = N ENTONCES
        IgualA(Comp(B_1, N), Comp(B_2, N))
    SI NO
        IgualA(Comp(B_1, i), Comp(B_2, i)) AND
        IgualB(suc(i), B_1, B_2)

PreordenA es una función que devuelve una lista con los elementos de un árbol de orden N recorriéndolo en profundidad por el criterio de preorden: primero la raíz y luego los hijos del primero al último recorridos en preorden.

PreordenA: ARBOLN $\longrightarrow$ LISTA
PreordenB: N $\times$ BOSQUEN $\not\longrightarrow$ LISTA
pre PreordenB(i, B): $1 \leq i \leq$ TMáx
PreordenA(CrearÁrbolN) == Crear
PreordenA(Árbol_Orden_N(r, B)) == Cons(r, PreordenB(B))

PreordenB(i, B) == SI i = N ENTONCES
        PreordenA(Comp(B, N))
    SI NO
        Concatenar(PreordenA(Comp(B, i)),
                    PreordenB(suc(i), B))

5.1.2 Implementación de árboles de orden N.

Las estructuras para implementar árboles de orden N son análogas a las que se vieron en el tema anterior para árboles binarios.

5.1.2.1 Representaciones dinámicas.

La representación dinámica de un árbol consta de un registro con el valor del nodo y una lista con punteros a los hijos. Esta lista puede implementarse mediante un array o mediante una lista enlazada.

El uso del array está indicado si el árbol está bastante lleno, para reducir el desperdicio de memoria. Se puede optar por tener un puntero a un array en el nodo en vez del array en sí, y sólo reservar memoria para el array cuando de agregue un hijo.

La estructura de tipos sería la siguiente:

```
CONS
    MAX = .....;
TYPE
    ARBOLN = POINTER TO ESTRUCTURA;
    ESTRUCTURA =  RECORD
                      raiz: ELEMENTO;
```

```
        bosque: ARRAY [1 .. MAX] OF ARBOLN;
END;
```

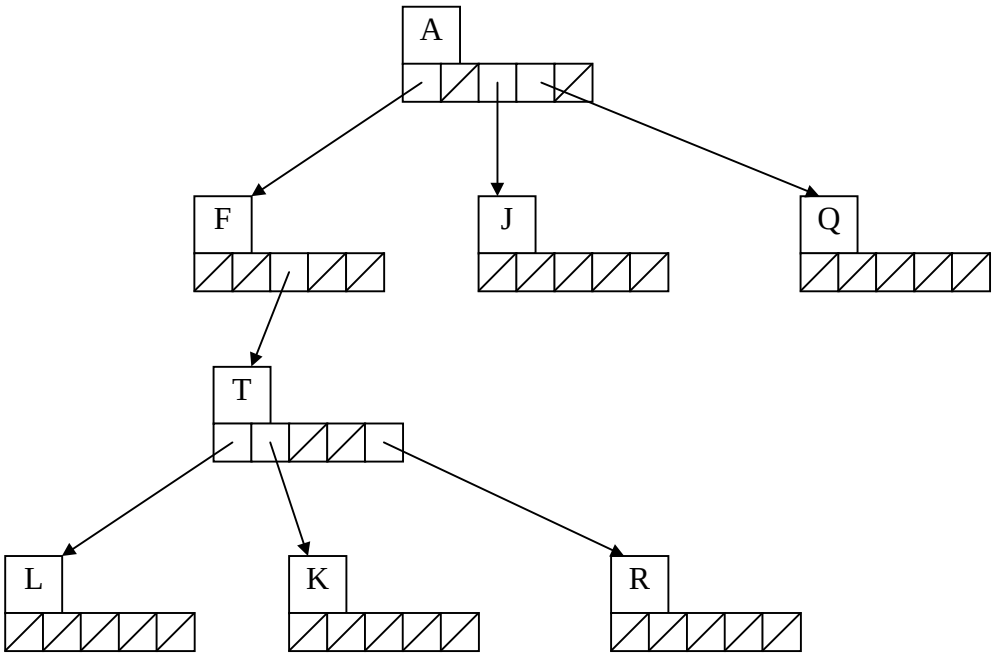


Figura 5. 1. Árbol de orden 5 con estructuras dinámicas y arrays para el bosque de hijos.

Si usamos una lista enlazada tenemos que asegurarnos que no se van a incluir más hijos de los que haya permitidos y que no se repitan dos árboles en la misma posición de la lista. Además, cada elemento de la lista ha de tener el puntero a la raíz del hijo y un indicador de su posición dentro de la lista.

La estructura de tipos sería la siguiente:

```
CONS
        MAX_HIJOS = .....;
TYPE
        ARBOLN = POINTER TO ESTRUCTURA;
        LISTAN = POINTER TO NODOLISTA;
        ESTRUCTURA =    RECORD
                                raiz: ELEMENTO;
                                bosque: LISTAN;
                        END;
        NODOLISTA =     RECORD
                                posicion: [1.. MAX_HIJOS];
                                hijo: ARBOLN;
                                sig := LISTAN;
                        END;
```

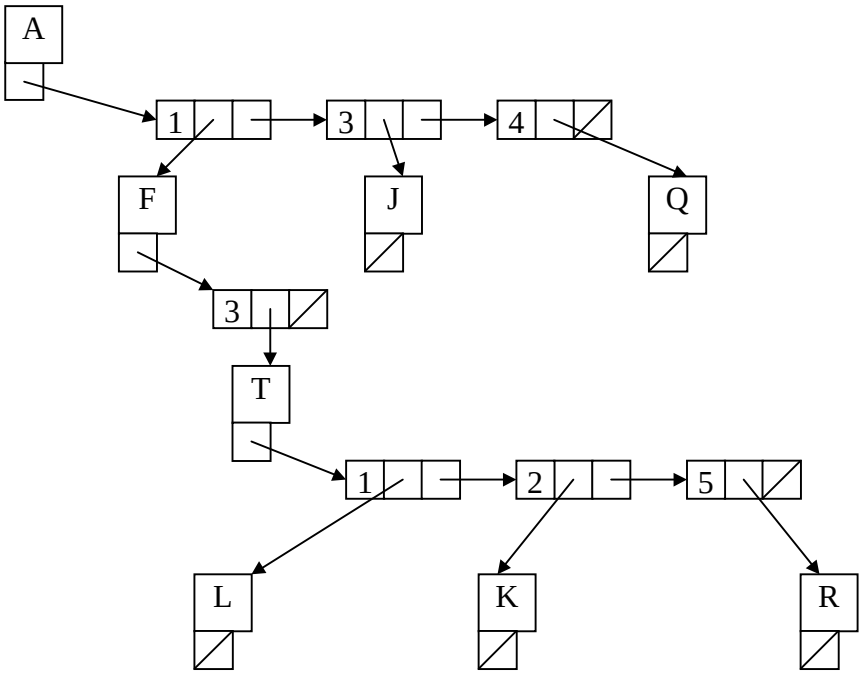


Figura 5. 2. El árbol de la figura anterior usando una lista enlazada para el bosque de hijos.

5.1.2.2 Representaciones estáticas.

Si usamos la representación estática basada en cursores, por cada casilla del array tendríamos el valor del nodo y un array con las posiciones de las raíces de los hijos. Esto provoca un desperdicio de memoria muy grande, sobre todo si el máximo número de hijos es elevado y el tamaño de la información útil no es mucho mayor que el del cursor.

Otra forma que no desperdicia tanto espacio es la representación *dispersa en amplitud*. En esta representación, los nodos se almacenan siguiendo el recorrido en amplitud del árbol. Cada elemento ocupará la posición del array que corresponda a su orden dentro de ese recorrido suponiendo que el árbol esté lleno. Esto implica que hemos de reservar casillas para los nodos del árbol que no estén ocupados.

El acceso a un elemento cualquiera de un determinado nivel es directo usando la relación entre la posición del array y la posición en el árbol. El i-ésimo elemento del j-ésimo nivel se encontrará en la posición N^{j-1} + (i-1). El padre del nodo situado en la posición k se encontrará en la posición k DIV N y su i-ésimo hijo se encontrará en la posición (N * k) + (i - 2).

Para que el uso de memoria sea eficiente, los árboles representados deben estar casi llenos o completos. Además es necesario distinguir aquellas casillas que estén vacías, bien con un valor especial o con un campo adicional. El tipo sería el siguiente:

```
CONS
        ALTURA_MAX = .....;
        MAX_NODOS = 2**ALTURA_MAX - 1;
TYPE
```

```
ARBOLN = POINTER TO ESTRUCTURA;
ESTRUCTURA = ARRAY [1 .. MAX] OF PAR;
PAR = RECORD
            libre: BOOLEAN;
            valor: ELEMENTO;
      END;
```

A	F		J	Q				T												L	K			R						
u	u	l	u	u	l	l	l	u	l	l	l	l	l	l	l	l	l	l	l	u	u	l	l	u	l	l	l	l	l	l

Figura 5. 3. El árbol de la figura anterior representado de forma dispersa con cursores.

5.2 Árboles generales.

Un árbol general con nodos de tipo T es un par (r, L_A), formado por un nodo r (la **raíz**) y una lista L_A de árboles generales del mismo tipo. La lista se conoce como **bosque** y puede estar vacía. Cada uno de los árboles del bosque es un hijo. Si el orden de los hijos no es relevante L_A, es un conjunto en vez de una lista. Nótese que con esta definición, no puede haber árboles generales vacíos, sino que tiene que haber al menos un elemento, la raíz

Si contempláramos el caso del árbol general vacío, podríamos tener un bosque de árboles generales vacíos que, conceptualmente, sería igual que el bosque vacío. Esto nos obligaría a definir nuevas ecuaciones que llevarían a un conjunto de generadores no libres, complicando la especificación del tipo.

En un árbol general se define el *grado de un nodo* como el número de hijos de ese nodo y el *grado del árbol* como el máximo de los grados de los nodos del árbol.

5.2.1 Especificación de árboles generales.

En este tipo contaremos con una única operación generadora, ÁrbolGen, que construye un árbol a partir de un elemento y un bosque de árboles. Como selectores contaremos con una operación Raíz que devuelva la raíz de un árbol y otra operación Hijos que devuelva el bosque de hijos de un árbol.

Respecto al tipo BOSQUE usado en la especificación del árbol general tendremos una operación para generar un bosque vacío y otra para añadir un árbol a un bosque. Además contaremos con operaciones que devuelvan el primer árbol del bosque y el resto de los árboles del bosque. Como se puede ver, esta especificación coincide básicamente con la de la lista funcional. Además podremos preguntar si un bosque está vacío o no.

tipo BOSQUEG
dominios N, ARBOLG
generadores
CrearBosqueG: $\longrightarrow$ BOSQUEG
Insertar: ARBOLG $\times$ BOSQUEG $\longrightarrow$ BOSQUEG

constructores
Resto: BOSQUEG ⟶ BOSQUEG
selectores
Primero: BOSQUEG —/→ ARBOLG
EsBosqueGVacío: BOSQUEG ⟶ LOGICO
precondiciones B: BOSQUEG; i: N; A: ARBOLG
pre Primero(B): not EsBosqueGVacío(B)
ecuaciones B: BOSQUEG; A: ARBOLG
EsBosqueGVacío(CrearBosqueG) == V
EsBosqueGVacío(Insertar(A, B)) == F
Primero(Insertar(A, B)) == A
Resto(CrearBosqueG) == CrearBosqueG
Resto(Insertar(A, B)) == B
fin

tipo ARBOLG
dominios ELEMENTO, LOGICO, N, BOSQUEg
generadores
ÁrbolGen: ELEMENTO × BOSQUEG ⟶ ARBOLG
selectores
Raíz: ARBOLG ⟶ ELEMENTO
Hijos: ARBOLG ⟶ BOSQUEG
ecuaciones A: ARBOLG; B: BOSQUEG;e: ELEMENTO;
Raíz(ÁrbolGen(e, B)) == e
Hijos(ÁrbolGen(e, B)) == B
fin

Nótese que, al no existir el concepto de árbol general vacío, las operaciones que obtienen la raíz y los hijos son totales, a diferencia de lo que ocurre en los árboles de orden N.

A este definición básica se pueden añadir cuantas operaciones resulten necesarias para desarrollar toda la potencia del tipo. Cada operación que se defina sobre el árbol requerirá la definición de una operación análoga para el tipo BOSQUEG. El significado de las funciones es el mismo que para los árboles de orden N. A este tipo de árboles se añade una operación que no tenía sentido plantearse sobre los otros: la operación GradoA, que devuelve el mayor número de hijos que tiene un nodo cualquiera de un árbol.

Igual_FormaA: ARBOLG × ARBOLG ⟶ LOGICO
Igual_FormaB: BOSQUEG × BOSQUEG ⟶ LOGICO
Número_NodosA: ARBOLG ⟶ N
Número_NodosB: BOSQUEG ⟶ N
AlturaA: ARBOLG ⟶ N
AlturaB: BOSQUEG ⟶ N
GradoA: ARBOLG ⟶ N
GradoB: BOSQUEG ⟶ N
Longitud: BOSQUEG ⟶ N
EstáA: ELEMENTO × ARBOLG ⟶ LOGICO

EstáB: ELEMENTO × BOSQUEG ⟶ LOGICO
IgualA: ARBOLG × ARBOLG ⟶ LOGICO
IgualB: BOSQUEG × BOSQUEG ⟶ LOGICO
PreordenA: ARBOLG ⟶ LISTA
PreordenB: BOSQUEG ⟶ LISTA

Igual_FormaA(ÁrbolGen(r, B_1), ÁrbolGen(s, B_2)) == Igual_FormaB(B_1, B_2)

Igual_FormaB(CrearBosqueG, CrearBosqueG) == V
Igual_FormaB(CrearBosqueG, Insertar(A,B)) == F
Igual_FormaB(Insertar(A,B), CrearBosqueG) == F
Igual_FormaB(Insertar(A_1,B_1), Insertar(A_2,B_2)) ==
Igual_FormaA(A_1, A_2) AND Igual_FormaB(B_1, B_2)

Número_NodosA(ÁrbolGen(r, B)) == 1 + Número_NodosB (B)

Número_NodosB(CrearBosqueG) == 0
Número_NodosB(Insertar(A,B)) == Número_NodosA(A) + Número_NodosB(B)

AlturaA(ÁrbolGen(r, B)) == 1 + AlturaB(B)

AlturaB(CrearBosqueG) == 0
AlturaB(Insertar(A,B)) == Máximo(AlturaA(A), AlturaB(B))

Longitud(CrearBosqueG) == 0
Longitud(Insertar(A, B)) == 1 + Longitud(B)

GradoA(ÁrbolGen(r, B)) == Máximo(Longitud(B), GradoB(B))

GradoB(CrearBosqueG) == 0
GradoB(Insertar(A,B)) == Máximo(GradoA(A), GradoB(B))

EstáA(i, ÁrbolGen(r, B)) == (i = r) OR EstáB(i, B)

EstáB(i, CrearBosqueG) == F
EstáB(i, Insertar(A,B)) == EstáA(i, A) OR EstáB(i, B)

IgualA(ÁrbolGen(r, B_1), ÁrbolGen(s, B_2)) == (r = s) AND IgualB(B_1, B_2)

IgualB(CrearBosqueG, CrearBosqueG) == V
IgualB(CrearBosqueG, Insertar(A,B)) == F
IgualB(Insertar(A,B), CrearBosqueG) == F
IgualB(Insertar(A_1,B_1), Insertar(A_2,B_2)) ==
IgualA(A_1, A_2) AND IgualB(B_1, B_2)

PreordenA(ÁrbolGen(r, B)) == Cons(r, PreordenB(B))

PreordenB(CrearBosqueG) == Crear
PreordenB(Insertar(A,B)) == Concatenar(PreordenA(A), PreordenB(B))

5.2.2 Implementación de árboles de generales.

Para los árboles generales vamos a presentar únicamente dos representaciones dinámicas, una basada el listas enlazadas y otra basada en árboles binarios.

5.2.2.1 Representación basada en listas.

En esta representación un árbol consta de un registro con el valor del nodo y una lista enlazada de punteros que apuntan a los hijos.

La estructura de tipos sería la siguiente:

```
TYPE
        ARBOLG = POINTER TO ESTRUCTURA;
        LISTAG = POINTER TO NODOLISTA;
        ESTRUCTURA =    RECORD
                                raiz: ELEMENTO;
                                bosque: LISTAG;
                        END;
        NODOLISTA =     RECORD
                                cont: ARBOLG;
                                sig := LISTAG;
                        END;
```

Con esta implementación las operaciones para obtener los hijos de un nodo y el primer árbol de un bosque y el resto del bosque son directas.

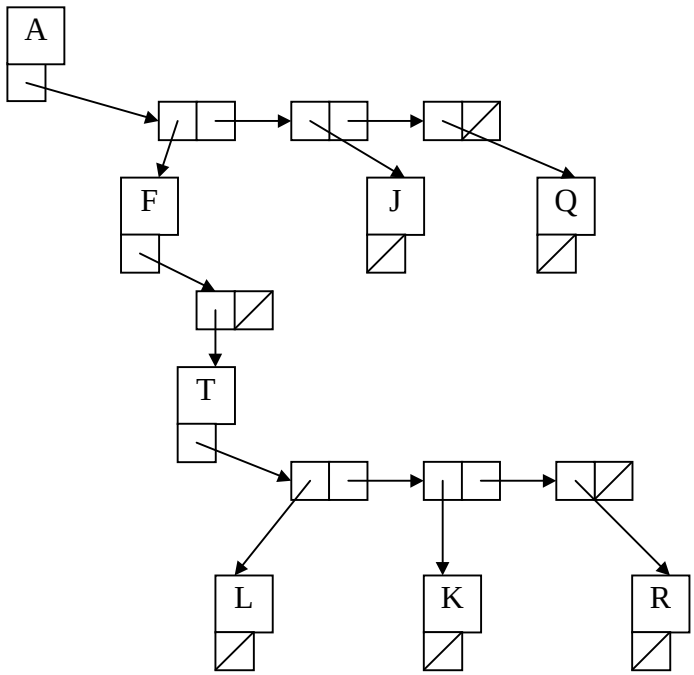


Figura 5. 4. El árbol de la figura 5.1 como árbol general mediante listas.

5.2.2.2 Representación basada en árboles binarios.

En esta representación se usa un solo tipo de nodo a diferencia de la anterior. Todos los nodos del árbol serán nodos iguales que los de un árbol binario con un campo que indique si ese nodo es el último de un bosque o no. La raíz se conecta al primer hijo del bosque a través del puntero al hijo izquierdo. Dentro del bosque, cada hijo se conecta al siguiente hermano por el puntero al hijo derecho. El último hijo del bosque tiene el puntero al hijo derecho a NIL. Básicamente estamos usando un árbol binario degenerado para representar la lista de hijos.

Con esta representación las operaciones para acceder al primer hijo y al resto del bosque también son directas. Si queremos que el acceso al padre sea también rápido podemos añadir un nuevo puntero a cada nodo que apunte al padre, pero esto complica las operaciones. Otra solución más simple se basa en el uso de hebras. El hijo derecho del último hijo del bosque apunta al nodo padre en vez de estar vacío. Para diferenciar si ese puntero apunta al siguiente hermano o al padre, se añadirá un nuevo campo al nodo que indique su función. Si apunta al padre el campo valdrá VERDADERO y se está apuntando al siguiente hermano valdrá FALSO.

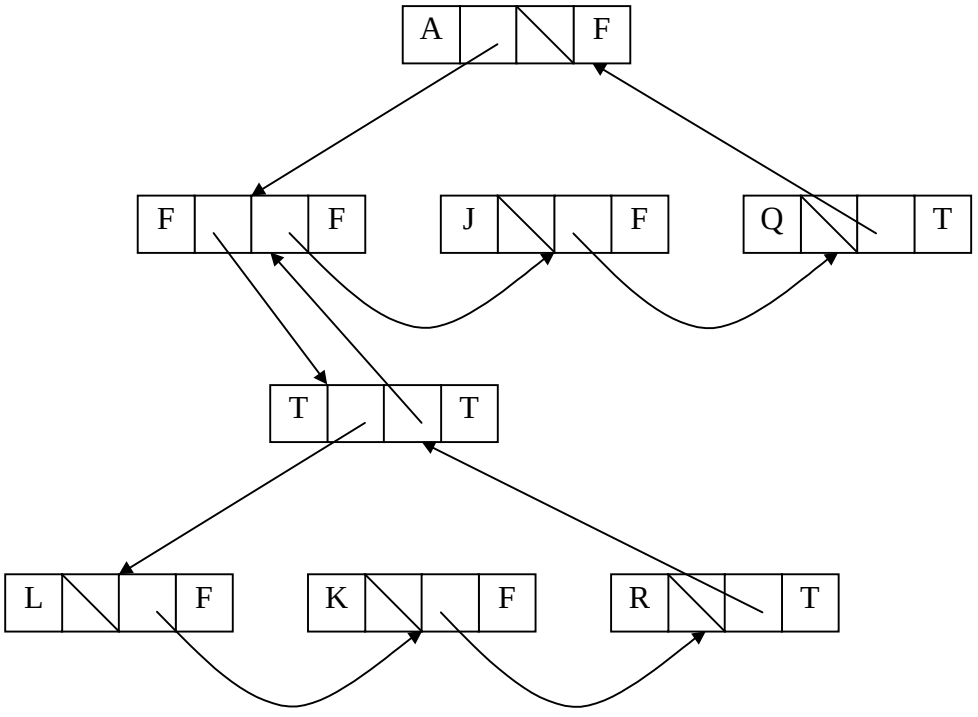


Figura 5. 5. Implementación del árbol anterior basado en árboles binarios en la que el último hermano apunta al padre.

5.3 Árboles 2-3.

Como se vio en el tema de árboles binarios, los árboles de búsqueda se usan para mantener una estructura enlazada en la que se pueda realizar la búsqueda de un elemento en un tiempo similar a la búsqueda binaria sobre un array. En estos árboles, se compara el

elemento a buscar una vez en cada nivel del árbol hasta que lo encontremos o salgamos de árbol sin hallarlo. El número de niveles para un árbol de búsqueda con *n* elementos puede estar entre *n* para árboles degenerados y $\log_2(n)$ para árboles completamente equilibrados. Esta variación de altura, que depende del orden de inserción de los elementos, nos lleva a buscar nuevos tipos de árboles que se mantengan lo más equilibrado posible para asegurar un rendimiento eficiente de la operación de búsqueda.

Ya se vieron los árboles AVL que mantenían una altura cercana a la mínima, pero sin garantizar que se alcanzara. En este apartado vamos a presentar un nuevo tipo de árbol balanceado, en el que se garantiza una altura mínima y, además servirán de introducción a otros árboles, los árboles B, que se aplican para mantener estructuras enlazadas de datos almacenados en memoria secundaria.

5.3.1 Definición de árboles 2-3.

Un árbol 2-3 permite que un nodo tenga dos o tres hijos. Esta característica le permite conservar el balanceo tras insertar o borrar elementos, por lo que el algoritmo de búsqueda es casi tan rápido como en un árbol de búsqueda de altura mínima. Por otro lado, es mucho más fácil de mantenerlo.

En un árbol 2-3, los nodos internos han de tener 2 ó 3 hijos y todas las hojas han de estar al mismo nivel. De forma recursiva se pueden definir como:

A es un árbol 2-3 de altura *h* si:

- *A* es un árbol vacío (un árbol 2-3 de altura 0), o
- *A* es de la forma (r, A_i, A_d), donde *r* es un nodo y A_i y A_d son árboles 2-3 de altura *h-1*, o
- *A* es de la forma (r, A_i, A_c, A_d), donde *r* es un nodo y A_i, A_c y A_d son árboles 2-3 de altura *h-1*.

Al poder tener los nodos tres hijos no estamos en un caso de un árbol binario. Si todos los nodos tienen dos hijos coincidirá con un árbol binario completo.

Para usar estos árboles de forma eficiente en las búsquedas, tenemos que introducir un orden entre los elementos por lo que daremos una nueva definición.

Un árbol *A* es un árbol 2-3 de búsqueda de altura *h* si:

- *A* está vacío, o
- *A* es de la forma (r, A_i, A_d), donde *r* contiene un elemento, A_i y A_d son árboles 2-3 de búsqueda de altura *h-1* y todos los elementos de A_i son menores que el elemento de *r* y todos los elementos de A_d son mayores que el elemento de *r*, o
- *A* es de la forma (r, A_i, A_c, A_d), donde *r* contiene dos elementos (r_1 y r_2), A_i, A_c y A_d son árboles 2-3 de búsqueda de altura *h-1* y todos los elementos de A_i son menores que el menor elemento de *r*, todos los elementos de A_c son mayores que el

elemento menor de *r* y menores que el elemento mayor de *r* y todos los elementos de A_d son mayores que el mayor elemento de *r*.

Esta definición implica que el número de hijos de un nodo es siempre uno más que el número de elementos que contiene ese nodo. En el caso de las hojas se permiten uno o dos elementos en el nodo.

Desde ahora nos referiremos a los árboles 2-3 de búsqueda simplemente como árboles 2-3.

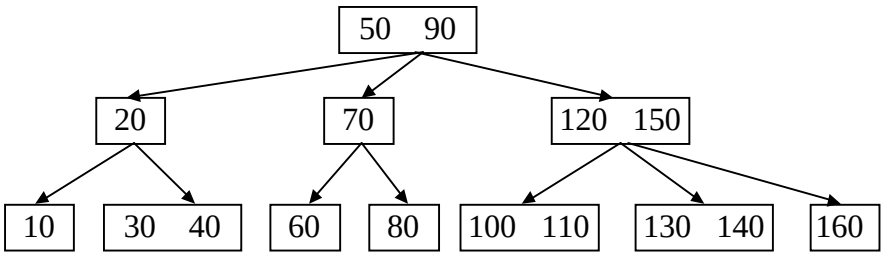


Figura 5. 6. Árbol 2-3 de búsqueda.

5.3.2 Búsqueda en árboles 2-3.

La última definición de árboles 2-3 es análoga a la de los árboles binarios de búsqueda, por lo que el algoritmo de búsqueda también lo será.

Para buscar un dato en un árbol 2-3, se empieza por la raíz y se va bajando por el árbol hasta que se encuentre el dato o se llegue a una hoja. El camino que se sigue viene dado por la comparación entre el dato buscado y el contenido del nodo inspeccionado en cada momento.

Si el árbol está vacío, el dato no está en el árbol. Si no está vacío, primero lo buscamos en la raíz. Si es alguno de los elementos de la raíz, ya lo hemos encontrado. En caso contrario, si el nodo es una hoja, el elemento no está en el árbol y el algoritmo acaba.

Si el nodo no es una hoja se sabe el subárbol por el que seguir la búsqueda comparando el dato con los elementos del nodo. Si sólo hay un elemento en el nodo y es mayor que el dato a buscar, seguimos por el hijo derecho, si es menor, por el hijo izquierdo. Si en el nodo hay dos elementos, r_1 y r_2 (con $r_1 \leq r_2$) y el dato es menor que r_1, buscamos por el hijo izquierdo. Si el dato es mayor que r_1 y menor que r_2, buscamos por el hijo central y si es mayor que r_2 buscamos por el hijo derecho.

Como en un árbol 2-3 puede haber más de un elemento en un nodo, se reduce el número de nodos inspeccionados, aunque el número de comparaciones no sea menor que en un árbol binario de búsqueda completamente equilibrado. Se puede demostrar que la complejidad de la búsqueda de $O(\log_2(n))$ en el peor caso.

5.3.3 Inserción en árboles 2-3.

A la hora de insertar un nuevo dato en un árbol 2-3 lo haremos de forma que se mantenga el equilibrio en el árbol. La capacidad de tener uno o dos elementos en cada nodo nos va a ayudar a conseguirlo.

Veamos ejemplos concretos para ilustrar el mecanismo de inserción.

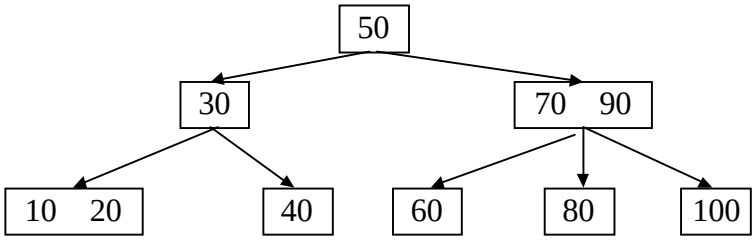


Figura 5. 7. Árbol 2-3 donde se insertarán los datos.

Supongamos que en el árbol de la figura queremos insertar el dato 39. Como en los árboles binarios de búsqueda, el primer caso consiste en localizar el nodo hoja donde debe ir el elemento siguiendo un recorrido similar al que se produce en la búsqueda de elementos.

En este caso, el 39 va en el mismo nodo que el 40. Como sólo hay un elemento en ese nodo, le añadimos el 39 y hemos acabado la inserción.

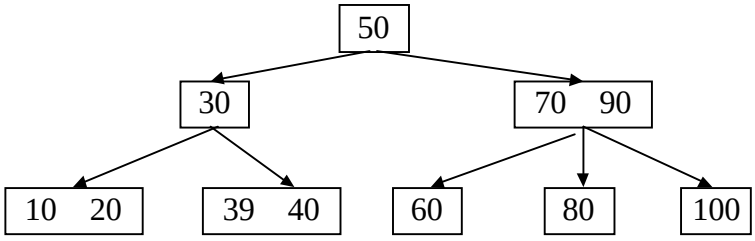


Figura 5. 8. Árbol de la figura anterior tras insertar el dato 39.

Si intentamos insertar el 38, iría en el nodo [39, 40], pero no hay sitio en él para un nuevo elemento. Lo que se hace en ese caso es dividir el nodo [38, 39, 40] en dos, uno con el [38] y otro con el [40] y el elemento central, en este caso el 39, lo insertamos en el nodo padre. El nodo padre pasa a ser [30, 39], el nodo [38] se convierte en el hijo central y el nodo [40] en el hijo derecho.

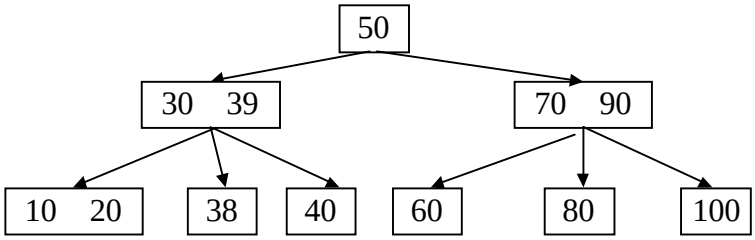


Figura 5. 9. Árbol 2-3 de la figura anterior tras insertar el dato 38.

Si el nodo padre hubiera estado ya lleno, al insertar el nuevo elemento habría que repetir la operación de división del nodo en dos e inserción en el nodo superior del elemento central. Si insertamos el dato 37 en el árbol anterior, se queda en el nodo del 38 sin ningún problema. Si, posteriormente, insertamos el dato 36, el árbol quedaría de la siguiente forma:

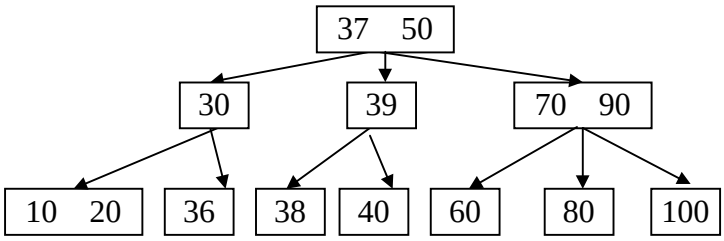


Figura 5. 10. Árbol de la figura anterior tras insertar los datos 37 y 36.

La secuencia de inserciones y divisiones se puede propagar hacia arriba hasta llegar a la raíz. Cuando la raíz pase a tener tres elementos [r_1, r_2 y r_3], se dividirá en dos nuevos nodos [r_1] y [r_3] y se creará una nueva raíz, que contendrá sólo el elemento [r_2]. De esta forma, cuando un árbol 2-3 crece en altura, lo hace por la arriba, creando una nueva raíz con un sólo elemento.

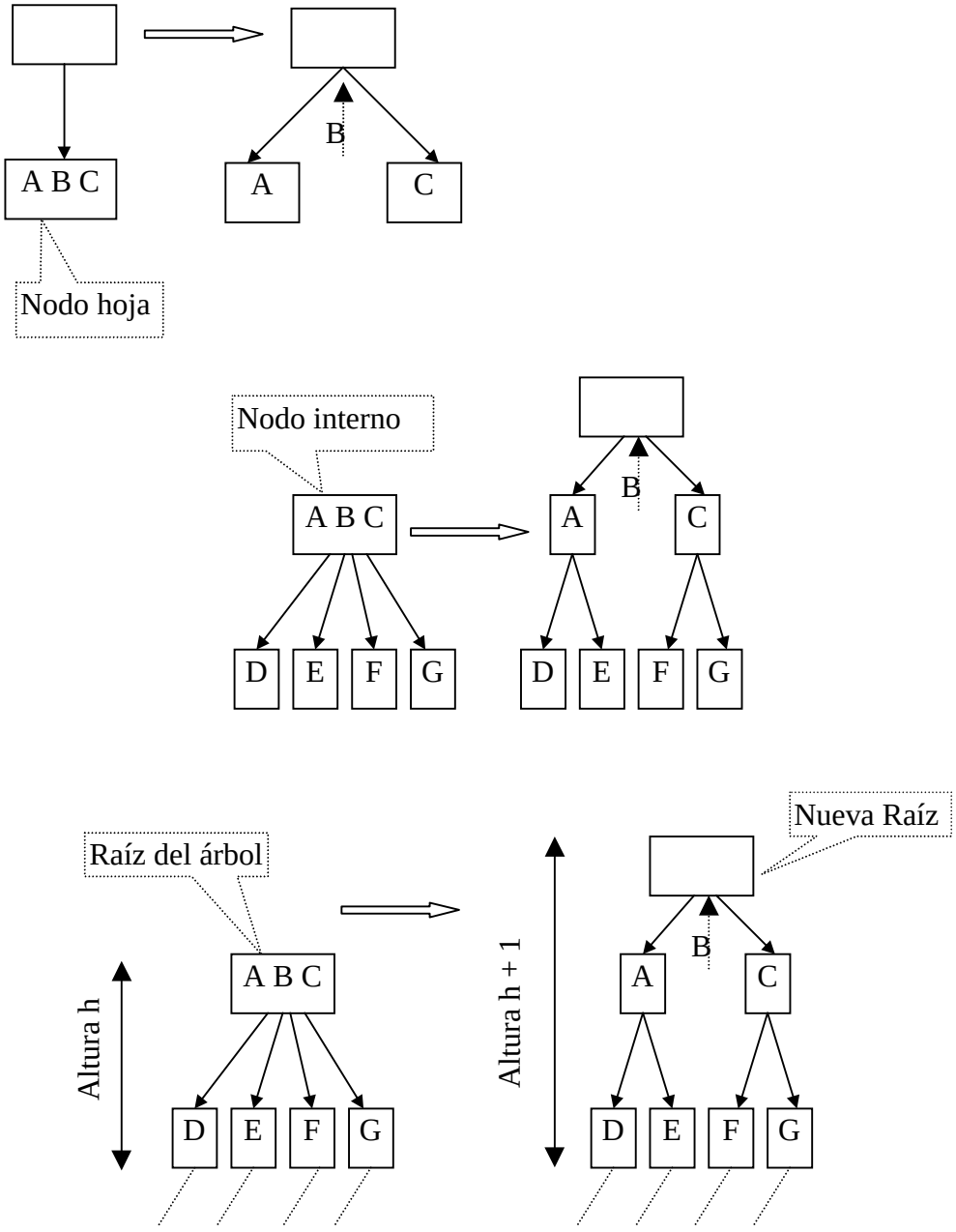


Figura 5. 11. División de nodos tras la inserción de un elemento.

5.3.4 Eliminación en árboles 2-3.

La estrategia de eliminación de datos de un árbol 2-3 es la complementaria a la de inserción. En la inserción se produce una cadena de divisiones e inserciones hasta que un nodo no necesite dividirse o se llegue a la raíz.

En el caso de la eliminación de datos, los nodos que se quedan vacíos tras la extracción de un elemento se fusionan con uno de sus hermanos para formar uno solo. Como el nodo padre ha perdido un hijo, también ha de tener un elemento menos, por lo que uno de los elementos del nodo padre pasa al nuevo nodo. La cadena de fusiones seguirá hasta que un nodo no se quede vacío o se llegue a la raíz.

El proceso siempre se va a empezar en un nodo hoja. Por tanto, si el elemento a borrar está en un nodo interior, se intercambia su valor con el su sucesor en inorden[1]. El sucesor en inorden es el menor elemento del subárbol que queda a la derecha del elemento a borrar. Este elemento siempre estará en un nodo hoja y su nueva posición en el árbol respeta el orden. En cambio, la nueva posición del elemento a borrar no está en orden, pero no importa porque, precisamente, ese elemento va a ser eliminado. Es desde esa hoja desde donde se empieza el algoritmo de eliminación.

Si en la hoja en la que empezamos la eliminación hay otro elemento, ahí acaba el proceso, pero si era el único elemento del nodo, éste se queda vacío, una situación que no está permitida en los árboles 2-3. Para arreglarlo, se fusionan el nodo que se ha quedado vacío con uno de sus hermanos. Como el nodo padre ha perdido un hijo, también tiene que tener un elemento menos, por lo que se pasa un elemento del padre al nuevo nodo. El elemento que se pasa es el antecesor común a los dos nodos fusionados.

Una vez fusionados los nodos, hay dos situaciones posibles. Si el nodo hermano ya estaba lleno (tenía dos elementos), no puede almacenar otro más. Se divide el nuevo nodo en dos, que serán hijos del nodo padre y se pasa el elemento medio al nodo padre. Así, el nodo padre acaba con el mismo número de elementos y de hijos. Simplemente, se han distribuido los elementos entre los nodos hijos. Hay que hacer notar que no se puede pasar directamente un valor de uno de los hermanos al nodo que se ha quedado vacío porque no se mantendría el orden del árbol.

Sin embargo, si el nodo hermano no estaba lleno (sólo tenía un elemento), admite el nuevo elemento y el nodo padre se queda con un elemento menos. Si el nodo padre se queda vacío al perder un elemento, se ha de repetir el algoritmo de fusión en el árbol tantos niveles hacia arriba como haga falta hasta encontrar un nodo que no se quede vacío o se llegue a la raíz. Si se llega a la raíz y se queda vacía, podemos asegurar que sólo tiene un hijo. La raíz se elimina y su único hijo pasa a ser la nueva raíz.

Para ilustrar el mecanismo de eliminación, borraremos del árbol de la figura los elementos 70, 100 y 80.

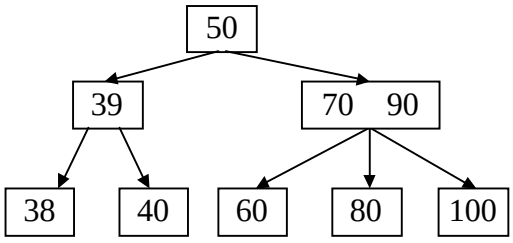


Figura 5. 12. Árbol 2-3 de donde se borrarán los datos.

Como el elemento 70 no está en una hoja, lo intercambiamos con su sucesor en inorden, el 80. El nodo quedaría entonces [80, 90] con tres hijos, [60], [70] y [100]. Al borrar el 70, su nodo se queda vacío. Se intenta escoger para fusionarse un hermano con dos

[1] También se puede intercambiar por al antecesor en inorden.

elementos, para que el nodo padre no pierda elementos y evitar que se propaguen la fusiones hacia arriba. Como este caso no tenemos ningún hermano que pueda ceder un elemento, escogemos uno cualquiera, por ejemplo el [60]. En la fusión se baja un elemento del nodo padre y el árbol queda finalmente:

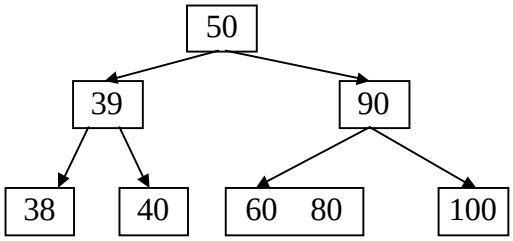


Figura 5. 13. Árbol de la figura anterior tras borrar el dato 70.

Ahora se pretende borrar el elemento 100, que ya está en un nodo hoja. Este nodo también se queda vacío, por lo que hay que repetir el proceso de fusión. Como el nodo hermano está lleno, al bajar un elemento del padre, hay que dividirlo y repartir los elementos. Los nuevos nodos son el [60] y el [90] y el 80 es el elemento que pasa al nodo padre. El árbol queda:

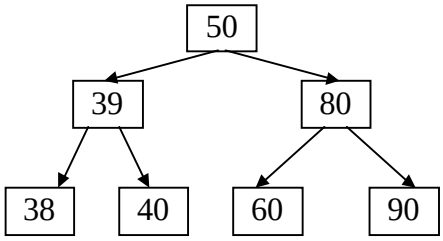


Figura 5. 14. Árbol de la figura anterior tras borrar el dato 100.

Como el elemento 80 está en un nodo intermedio, el primer paso cuando se va a borrar es intercambiarlo por su sucesor en inorden, el 90. Al eliminar el 80 de la hoja donde se ha colocado, ésta se queda vacía, por lo que hay que fusionarla con su hermano. Su hermano, que no está lleno, acepta el elemento que baja del nodo padre que se queda vacío. El árbol queda:

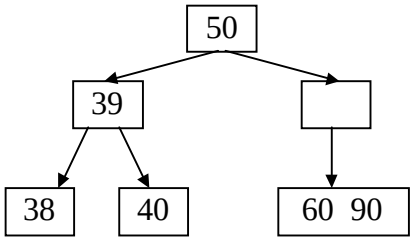


Figura 5. 15. Árbol 2-3 con un nodo vacío durante el borrado de un dato.

El nodo intermedio que se ha quedado vacío ha de fusionarse con el hermano ([39]). Como en el paso anterior, el hermano pasa a tener dos elementos y el padre, en este caso la raíz, se queda vacío:

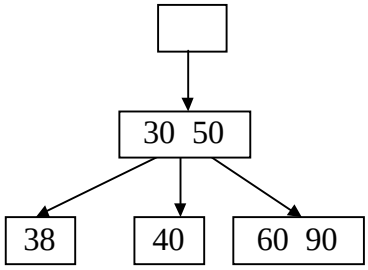


Figura 5. 16. Árbol 2-3 con la raíz vacía durante el borrado de un dato.

Finalmente, se borra el nodo raíz vacío y su único hijo pasa a ser la nueva raíz. El árbol ha perdido altura:

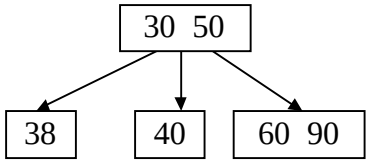


Figura 5. 17. Estado final del árbol tras borrar el dato 80.

5.3.5 Especificación de árboles 2-3.

La especificación del tipo ARBOL23 es muy similar a la de otros árboles con propiedades que hemos visto anteriormente, como los árboles binarios de búsqueda y los árboles AVL. Algunos de los generadores del tipo serán ocultos, esto es, no se ofrecerán como operaciones al usuario, sino que servirán para construir la especificación y, en su lugar se ofrecerán las operaciones de inserción y borrado de elementos que asegurarán que el resultado mantenga las propiedades de los árboles 2-3 respecto al orden y forma del árbol.

Una diferencia respecto a los anteriores tipos es la definición de tres operaciones generadoras en lugar de dos. ÁrbolVacío es la operación que genera un árbol sin elementos, como en los tipos anteriores y hay una operación (ConsÁrbol) que genera un árbol 2-3 cuya raíz tiene un solo elemento y dos hijos y otra (Cons3Árbol) que genera un árbol 2-3 cuya raíz tiene dos elementos y tres hijos. Estas dos últimas operaciones son los generadores que se mantienen ocultos al usuario.

Las operaciones de inserción y borrado hacen uso de una operación auxiliar llamada Equilibrar. Si no se utilizara esa operación, el resultado de insertar un elemento en un árbol podría producir un árbol con un hijo de una altura superior a la de sus hermanos en un nivel. En el caso del borrado se podría producir un árbol con un hijo de una altura inferior a la de sus hermanos en un nivel.

La operación de Equilibrar no sirve para equilibrar cualquier árbol, sino que admite como entrada un árbol 2-3 en el que uno de sus hijos tiene una altura mayor o menor que la de sus hermanos en un nivel y su resultado es un árbol 2-3 donde todos los hijos tienen la misma altura. Si el árbol de entrada ya estaba equilibrado, el resultado es el mismo árbol.

tipo ARBOL23;
dominios ITEM, BOOLEAN;

generadores
ÁrbolVacío: ⟶ ARBOL23;
ConsÁrbol: ITEM × ARBOL23 × ARBOL23 ⟶ ARBOL23;
Cons3Árbol:ITEM × ITEM × ARBOL23 × ARBOL23 × ÁRBOL23 ⟶ ARBOL23

constructores
ÁrbolIzq, ÁrbolDch: ARBOL23 ⟶ ARBOL23;
ÁrbolCentral: ARBOL23 —/→ ARBOL23;
Insertar: ITEM × ARBOL23 ⟶ ARBOL23;
Borrar: ITEM × ARBOL23 ⟶ ARBOL23;

selectores
EsÁrbolVacío: ARBOL23 ⟶ BOOLEAN;
Raíz: ARBOL23 —/→ ITEM;
PrimerValor, SegundoValor: ARBOL23 —/→ ITEM;

auxiliares
NúmValores: ARBOL23 ⟶ CARDINAL;
MinVal, MaxVal: ARBOL23 —/→ ITEM;

precondiciones A: ARBOL23;
pre ÁrbolCentral(A): NOT EsÁrbolVacío(A) AND NúmValores(A) = 2;
pre Raíz(A): NOT EsÁrbolVacío(A) AND NúmValores(A) = 1;
pre PrimerValor(A): NOT EsÁrbolVacío(A) AND NúmValores(A) = 2;
pre SegundoValor(A): NOT EsÁrbolVacío(A) AND NúmValores(A) = 2;
pre MinVal(A): NOT EsÁrbolVacío(A);
pre MaxVal(A): NOT EsÁrbolVacío(A);

ecuaciones i, j: ITEM; l, m, r: ARBOL23;
EsÁrbolVacío(ÁrbolVacío) == TRUE;
EsÁrbolVacío(ConsÁrbol(i, l, r)) == FALSE;
EsÁrbolVacío(Cons3Árbol(i, j, l, m, r)) == FALSE;

MinVal(ConsÁrbol(i, j, l, r)) == SI EsÁrbolVacío(l) ENTONCES i
SI NO MinVal(l);
MinVal(Cons3Árbol(i, j, l, m, r)) == SI EsÁrbolVacío(l) ENTONCES i
SI NO MinVal(l);

MaxVal(Cons3Árbol(i, j, l, r)) == SI EsÁrbolVacío(r) ENTONCES j
SI NO MaxVal(r);
MaxVal(Cons3Árbol(i, j, l, m, r)) == SI EsÁrbolVacío(r) ENTONCES j
SI NO MaxVal(r);

NúmValores(ÁrbolVacío) == 0
NúmValores(ConsÁrbol(i, l, r)) == 1

```
NúmValores(Cons3Árbol(i, j, l, m, r)) ==    2

Raíz(ConsÁrbol(i, l, r)) == i;

PrimerValor(Cons3Árbol(i, j, l, m, r)) == i;

SegundoValor(Cons3Árbol(i, j, l, m, r)) == j;

ÁrbolIzq(ÁrbolVacío) == ÁrbolVacío;
ÁrbolIzq(ConsÁrbol(i, l, r)) == l;
ÁrbolIzq(Cons3Árbol(i, j, l, m, r)) == l;

ÁrbolDch(ÁrbolVacío) == ÁrbolVacío;
ÁrbolDch(ConsÁrbol(i, l, r)) == r;
ÁrbolDch(Cons3Árbol(i, j, l, m, r)) == r;

ÁrbolCentral(Cons3Árbol(i, j, l, m, r)) == m;

Insertar(i, ÁrbolVacío) == ConsÁrbol(i, ÁrbolVacío, ÁrbolVacío);
Insertar(i, ConsÁrbol(j, ÁrbolVacío, ÁrbolVacío)) ==
        SI i<j ENTONCES
                Cons3Árbol(i, j, ÁrbolVacío, ÁrbolVacío, ÁrbolVacío)
        SI NO
                Cons3Árbol(j, i, ÁrbolVacío, ÁrbolVacío, ÁrbolVacío);
Insertar(i, Cons3Árbol(j, k, ÁrbolVacío, ÁrbolVacío, ÁrbolVacío)) ==
        ConsÁrbol(    Medio(i, j, k),
                        ConsÁrbol(Mínimo(i, j), ÁrbolVacío, ÁrbolVacío),
                        ConsÁrbol(Máximo(i, k), ÁrbolVacío, ÁrbolVacío));
Insertar(i, ConsÁrbol(j, l, r)) ==
        SI i<j ENTONCES
                Equilibrar(ConsÁrbol(j, Insertar(i, l), r))
        SI NO
                Equilibrar(ConsÁrbol(j, l, Insertar(i, r)));
Insertar(i, Cons3Árbol(j, k, l, m, r)) ==
        SI i<j ENTONCES
                Equilibrar(Cons3Árbol(j, k, Insertar(i, l), m, r))
        SI NO SI i<k ENTONCES
                Equilibrar(Cons3Árbol(j, k, l, Insertar(i, m), r))
        SI NO
                Equilibrar(Cons3Árbol(j, k, l, m, Insertar(i, r)));

Borrar(i, ArbolVacío) == ArbolVacío;
Borrar(i,ConsArbol(j,l,r)) ==
        SI i=j ENTONCES
                SI EsÁrbolVacío?(l) ENTONCES
                        ÁrbolVacío
                SI NO
                        Equilibrar(Cons2Arbol(MaxVal(l),Borrar(MaxVal(l),l),r))
        SINO SI i<j ENTONCES
```

```
                Equilibrar(Cons2Arbol(j,Borrar(i,l),r))
        SINO
                Equilibrar(Cons2Arbol(j,l,Borrar(i,r)))
Borrar(i,Cons3Arbol(j,k,l,m,r)) ==
        SI i=j ENTONCES
                SI EsÁrbolVacío?(l) ENTONCES
                        ConsArbol(k,ArbolVacío,ArbolVacío)
                SI NO
                Equilibrar(Cons3Arbol(MaxVal(l),k,Borrar(MaxVal(l),l),m,r))
        SINO SI i=k ENTONCES
                SI EsÁrbolVacío?(l) ENTONCES
                        ConsArbol(j,ArbolVacío,ArbolVacío)
                SI NO
                        Equilibrar(Cons3Arbol(j,MinVal(r),l,m,Borrar(MinVal(r),r)))
                SINO
                        SI i<j ENTONCES
                                Equilibrar(Cons3Arbol(j,k,Borrar(i,l),m,r))
                        SINO SI i<k ENTONCES
                                Equilibrar(Cons3Arbol(j,k,l,Borrar(i,m),r))
                        SINO
                                Equilibrar(Cons3Arbol(j,k,l,m,Borrar(i,r)));

Equilibrar(ÁrbolVacío) == ÁrbolVacío;
Equilibrar(ConsÁrbol(i, l, r)) ==
        SI Altura(l)=Altura(r) ENTONCES
                ConsÁrbol(i, l, r)
        SI NO SI Altura(l)>Altura(r) ENTONCES
                SI NúmValores(l) = 1 ENTONCES
                        Cons3Árbol(Raíz(l), i, ÁrbolIzq(l), ÁrbolDch(l), r)
                SI NO
                        ConsÁrbol(SegundoValor(l),
                                ConsÁrbol(PrimerValor(l), ÁrbolIzq(l), ÁrbolCentral(l)),
                                ConsÁrbol(i, ÁrbolDch(l), r))
        SI NO
                SI NúmValores(r) = 1 ENTONCES
                Cons3Árbol(i, Raíz(r), l, ÁrbolIzq(r), ÁrbolDch(r))
                SI NO
                        ConsÁrbol(PrimerValor(r),
                        ConsÁrbol(i, l, ÁrbolIzq(r)),
                        ConsÁrbol(SegundoValor(r), ÁrbolCentral(r), ÁrbolDch(r)))

Equilibrar(Cons3Árbol(i, j, l, m, r)) ==
        SI Altura(l) = Altura(m) = Altura(r) ENTONCES
                Cons3Árbol(i, j, l, m, r)
        SI NO SI Altura(m) = Altura(r) ENTONCES
                SI Altura(l) > Altura(m) ENTONCES (* Inserción *)
                        ConsÁrbol(i, l, ConsÁrbol(j, m, r))
                SI NO (* Borrado *)
                        SI NúmValores(m) = 1 ENTONCES
```

```
                    ConsÁrbol(j,
                        Cons3Árbol(   i, Raíz(m),
                                      l, ÁrbolIzq(m), ÁrbolDch(m)), r)
                SI NO
                    Cons3Árbol(PrimerValor(m), j,
                        ConsÁrbol(i, l, ÁrbolIzq(m)),
                        ConsÁrbol(    SegundoValor(m),
                                      ÁrbolCentral(m), ÁrbolDch(m)),
                        r)
        SI NO SI Altura(l) = Altura(r) ENTONCES
            SI Altura(m) > Altura(l) ENTONCES (* Inserción *)
                ConsÁrbol(Raíz(m),
                    ConsÁrbol(i,l,ÁrbolIzq(m)),
                    ConsÁrbol(j,ÁrbolDch(m),r))
            SI NO (* Borrado *)
                SI NúmValores(l) = 1 ENTONCES
                    SI NúmValores(r) = 1 ENTONCES
                        ConsÁrbol(j,
                                Cons3Árbol(Raíz(l), i,
                                    ÁrbolIzq(l), ÁrbolDch(l), m),
                                r)
                    SI NO
                        Cons3Árbol(i, PrimerValor(r), l,
                                ConsÁrbol(j, m, ÁrbolIzq(r)),
                                ConsÁrbol(SegundoValor(r),
                                        ÁrbolCentral(r), ÁrbolDch(r)))
                SI NO
                    Cons3Árbol(SegundoValor(l, j,
                        ConsÁrbol(    PrimerValor(l),
                                      ÁrbolIzq(l), ÁrbolCentral(l)),
                        ConsÁrbol(    i, ÁrbolDch(l), m),
                        r)
        SI NO SI Altura(m) = Altura(l) ENTONCES
            SI Altura(r) > Altura(m) ENTONCES (* Inserción *)
                ConsÁrbol(j, ConsÁrbol(i, l, m), r);
            SI NO (* Borrado *)
                SI NúmValores(m) = 1 ENTONCES
                    ConsÁrbol(i, l,
                        Cons3Árbol(   Raíz(m), j,
                                      ÁrbolIzq(m), ÁrbolDch(m), r))
                    SI NO
                        Cons3Árbol(i, SegundoValor(m), l,
                                ConsÁrbol(PrimerValor(m),
                                        ÁrbolIzq(m), ÁrbolCentral(m)),
                                ConsÁrbol(     j, ÁrbolDch(m), r))
```

fin

5.3.6 Implementación de árboles 2-3.

En este apartado vamos a presentar dos opciones para la estructura con la que implementar de forma dinámica. La primera opción se basa en un registro con un campo por cada uno de los valores posibles en un nodo:

```
TYPE
    ARBOL23 = POINTER TO NODO;
    NODO =    RECORD
                  NumValores: CARDINAL;
                  PrimerValor, SegundoValor: ELEMENTO;
                  Izquierdo, Medio, Derecho: ARBOL23;
              END;
```

El campo NumValores puede declararse de tipo BOOLEAN, ya que solo tiene dos valores posibles. Se puede tomar a TRUE cuando un solo valor y FALSE cuando tenga dos o al revés.

La otra opción se basa en un registro variante donde se separan claramente las dos formas que puede tener el nodo en función de que tenga un elemento o dos:

```
TYPE
    ARBOL23 = POINTER TO NODO;
    NODO =    RECORD
                  CASE NumValores: CARDINAL OF
                  1:
                      Raiz: ELEMENTO;
                      Izquierdo, Derecho: ARBOL23;
                  2:
                      PrimerValor, SegundoValor: ELEMENTO;
                      Izquierdo, Medio, Derecho: ARBOL23;
                  END;
              END;
```

En este caso también NumValores puede declararse de tipo BOOLEAN.

5.4 Árboles B (Balanceados).

Los árboles B tienen por objetivo agrupar en cada nodo más de un elemento de manera que el acceso a un elemento cualquiera tenga lugar visitando un número de nodos inferior al caso del árbol binario de búsqueda simple.

Esto es útil cuando el árbol se halla almacenado en un dispositivo de acceso lento, como puede ser el disco duro, y acceder a un nodo implica acceder a un sector distinto del disco. En estos casos se hace coincidir un sector físico con el tamaño de un nodo, de manera que si un elemento ocupar *e* bytes y un sector físico tiene un tamaño *f*, en cada nodo se almacenarán $\left\lfloor \frac{f}{e} \right\rfloor$ elementos. Así aprovechamos al máximo el acceso al disco.

Un árbol B de orden *n* es un árbol de búsqueda de orden *n* que cumple las siguientes propiedades[2]:

a) cada nodo tiene un máximo de 2*n* elementos y un mínimo de *n* elementos, excepto la raíz, que puede tener un mínimo de un elemento.

b) un nodo es o bien un nodo hoja sin hijos o bien un nodo intermedio con *m* elementos y *m + 1* hijos ($n \leq m \leq 2n$). Si e_1, e_2, ..., e_m denotan los elementos del nodo y P_0, P_1, .., P_m los hijos, entonces:

- P_0 contiene los elementos menores que e_1,
- P_i contiene los elementos mayores que e_i y menores que e_{i+1} y
- P_m contiene todos los elementos mayores que e_m.

c) todas las ramas tienen la misma longitud.

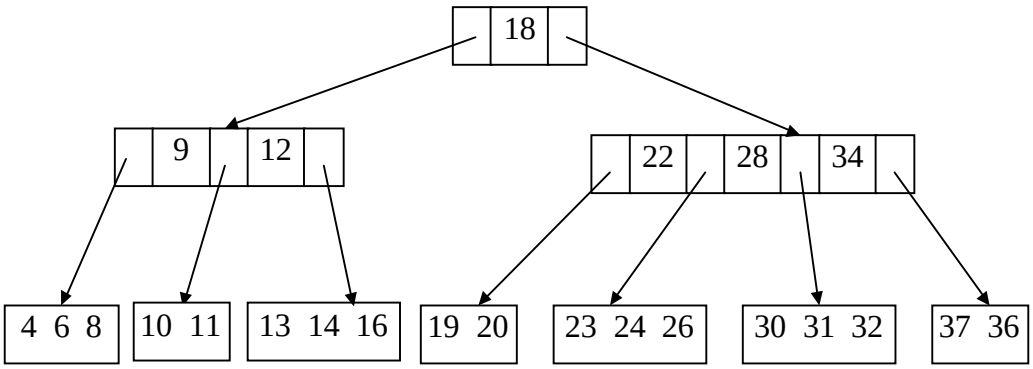


Figura 5. 18. Árbol B de orden 2.

Como se puede ver por la definición, los árboles 2-3 son árboles B de orden 1.

5.4.1 Inserción en árboles B.

La inserción de un elemento en un árbol de este tipo es análoga a la inserción en los árboles 2-3. Los nuevos datos siempre se insertan en un nodo hoja. Si la hoja la que se inserta el nuevo elemento tiene menos de 2*n* elementos, el nuevo elemento se inserta ordenadamente en esa hoja y ahí acaba el proceso de inserción. Pero si la hoja donde debe ir el nuevo elemento ya tiene 2*n* elementos, no podemos actuar de la misma manera, porque obtendríamos un nodo con 2*n* + 1 elementos. Para evitarlo, dividimos el nodo en dos, uno con los elementos e_1, .., e_n y otro con los elementos e_{n+2}, .. e_{2n}. El elemento e_{n+1} lo insertamos en el nodo padre siguiendo la misma técnica.

[2] Otros autores definen los árboles B de grado N como aquellos donde el número de elementos de un nodo está entre N -1 y N DIV 2. El resto de la definición es similar.

Si el nodo padre ya contenía 2*n* elementos, hay que volver a dividir. Este proceso puede repetirse hasta llegar al nodo raíz y hacer que éste se quede con 2*n* + 1 elementos. En ese caso, generamos una nueva raíz que contendrá sólo el elemento desplazado de la antigua raíz. De esta forma, un árbol B crece por la raíz, de manera que, siempre que la altura se incrementa en 1, la nueva raíz sólo tiene un elemento.

Veamos con un ejemplo cómo es la inserción en un árbol B de orden 2. Supongamos que queremos insertar la secuencia de elementos: 28, 47, 97, 2, 77.

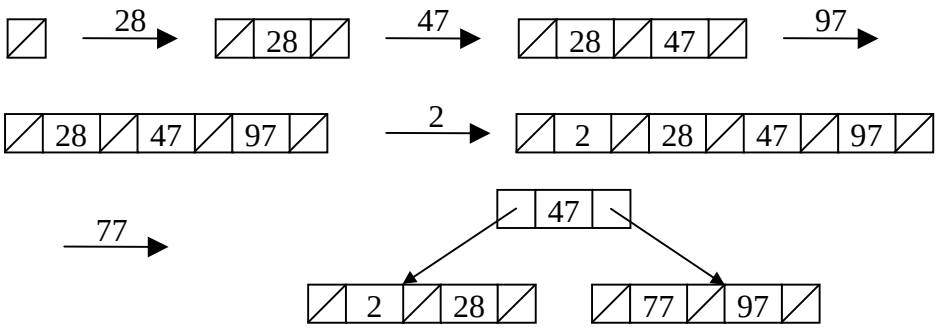


Figura 5. 19. Inserción de elementos en un árbol B de orden 2.

Veamos otro ejemplo: la inserción de los elementos 45 y 51 en un árbol que contiene ya otros elementos:

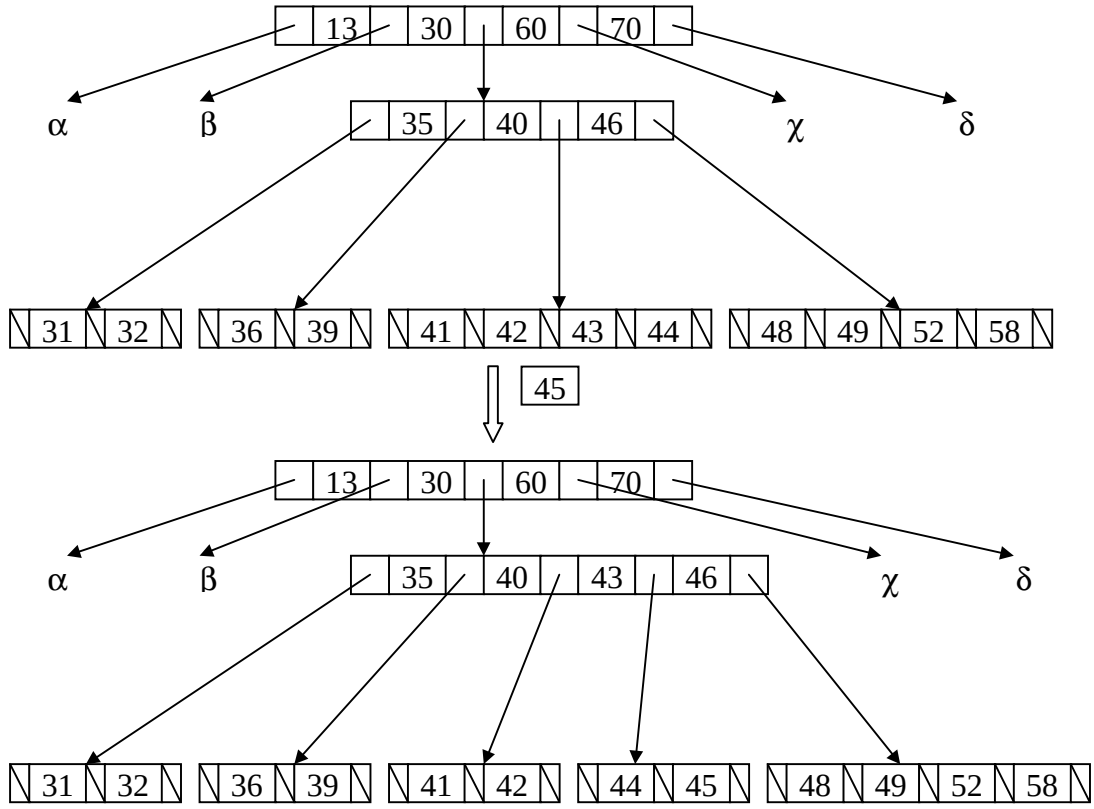


Figura 5. 20. Inserción del elemento 45 en un árbol B de orden 2.

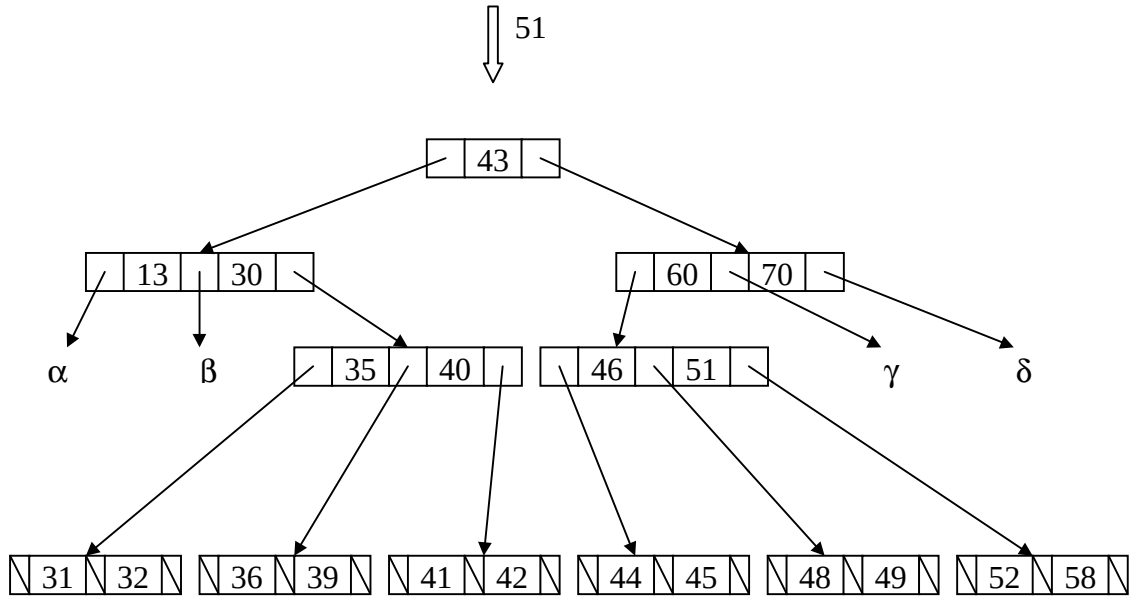


Figura 5. 21. Inserción del elemento 51 en el árbol de la figura anterior.

El nodo hoja en el que debería ir el elemento 51 (el que tiene [48, 49, 52, 58]) ya está lleno, por lo que lo dividimos en dos e insertamos en el nodo padre el elemento central,

precisamente el 51. A su vez , el nodo padre ya está lleno, por lo que lo volvemos a partir en dos e intentamos insertar el elemento central, en este caso el 43, en el nodo padre. El nodo donde intentamos insertar el 43 es la raíz y también se encuentra lleno, por lo que lo dividimos en dos y creamos un nuevo nodo raíz con el elemento medio, que vuelve a ser el 43.

5.4.2 Eliminación en árboles B.

La operación de eliminación también es análoga a la de los árboles 2-3. Siempre eliminaremos un elemento de un nodo hoja. Si el elemento a eliminar está en un nodo intermedio, lo sustituiremos por el elemento sucesor en inorden, que será el menor elemento del árbol situado inmediatamente a su derecha. El sucesor se encuentra, con seguridad, en un nodo hoja. También puede sustituirse por el elemento anterior en inorden, que sería el máximo elemento del árbol situado inmediatamente a su izquierda.

El problema aparece cuando intentamos eliminar un elemento de un nodo hoja que sólo tiene n elementos. Como el nodo no se puede quedar con menos de n elementos, hemos de conseguir de algún sitio otro elemento para reemplazarlo. Lo que hacemos es generar un nuevo nodo con el nodo de donde hemos eliminado, uno de los nodos vecinos (el derecho o el izquierdo) y el elemento común a ambos del nodo padre. Podemos encontrarnos con dos situaciones tras la fusión:

a) el nodo vecino tiene más de n elementos, con lo que el nodo que resulta de la fusión tiene al menos $2n + 1$ elementos. Entonces, el elemento mediana del nuevo nodo se inserta en el nodo padre en la posición del elemento que se ha tomado prestado y se crean dos nuevos nodos, uno con los elementos menores y otro con los elementos mayores. Ambos nodos tendrán al menos n elementos cada uno.

b) el nodo vecino tiene n elementos, con lo que el nodo que resulta de la fusión tiene $2n$ nodos, por lo que no tenemos que dividirlo. Sin embargo, si el nodo padre tenía n elementos, al perder uno, se queda con n-1. Para conseguir otro elemento para este nodo aplicamos este mismo proceso reiteradamente hasta que tengamos al menos n elementos en el nodo sin necesidad de coger otro de otro nodo o el nodo que pierde el elemento sea la raíz. Como la raíz no tiene que cumplir la restricción del número de elementos no hace falta aplicarle el proceso.

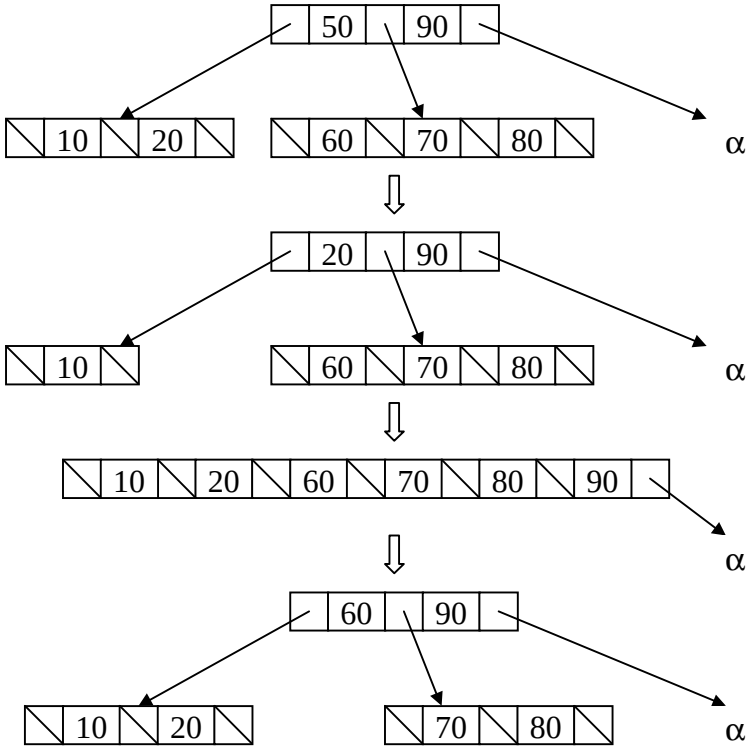


Figura 5. 22. Eliminación del elemento 50 de un árbol B de orden 2.

En el ejemplo de la figura 5.10, como el elemento a eliminar (50) no se encuentra en una hoja, lo sustituimos por el mayor del árbol inmediatamente a su izquierda, el 20. Al quitar el 20, ese nodo se queda con un solo elemento, por lo que hemos de fusionarlo con el nodo vecino y el elemento común del padre, el 20. El nodo resultante tiene $2n + 1$ elementos, por lo que lo dividimos en dos nuevos nodos y pasamos el elemento central al nodo padre.

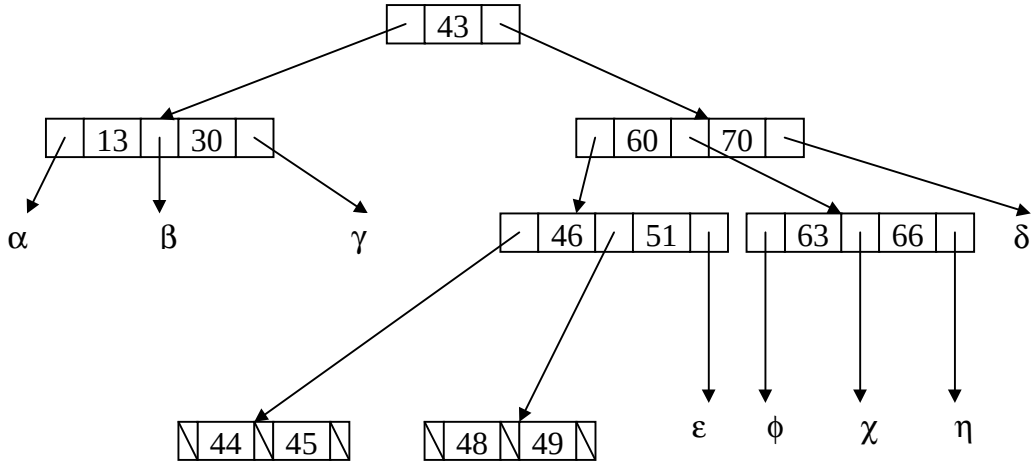


Figura 5. 23. Árbol B de orden 2 en el que se va a eliminar el elemento 43.

En el árbol de la figura 5.11 vamos a eliminar el elemento 43. Si lo sustituimos por el siguiente elemento en inorden, sacaremos el elemento 44 del nodo hoja en el que está. Como

ese nodo se queda con un solo elemento, hay que reorganizar los nodos. Fusionamos en un nodo el 45, los nodos del árbol vecino y el elemento común de la raíz. Con esos elementos formamos el nodo [45, 46, 48, 49], que no hay que separarlo. Pero entonces el padre se queda con un solo elemento, por lo que tenemos que reorganizar otra vez. Formamos un nodo con el elemento 51, los elementos del nodo vecino y el elemento común del padre. El nodo formado es el [51, 60, 63, 66]. Este nodo no hay que dividirlo, pero otra vez el nodo padre se queda con un solo elemento. Hay que volver a reorganizar. Se crea un nuevo nodo con el nodo restante, los elementos del nodo vecino y el elemento de la raíz. El nuevo nodo está formado por los elementos [13, 30, 44, 70] y pasa a ser la raíz.

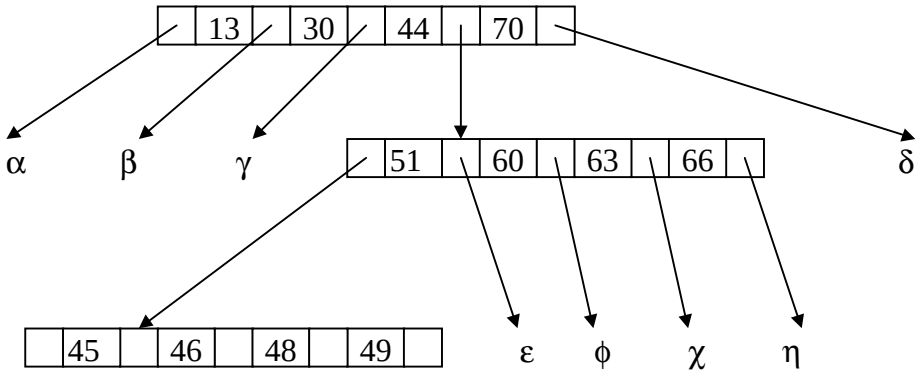


Figura 5. 24. Árbol de la figura anterior tras la eliminación del elemento 43.

Como puede observarse, cada nodo, excepto la raíz, tiene una carga mínima del 50 %. Hay otras variedades de árboles balanceados que aumentan la carga mínima por nodo. Los nodos B* aseguran una carga mínima del 66 %. Los árboles compactos aseguran una carga mínima del 90 %. Evidentemente, en estos árboles los algoritmos de inserción y borrado son más complejos y provocan un número mayor de propagaciones, por lo que son más costosos.

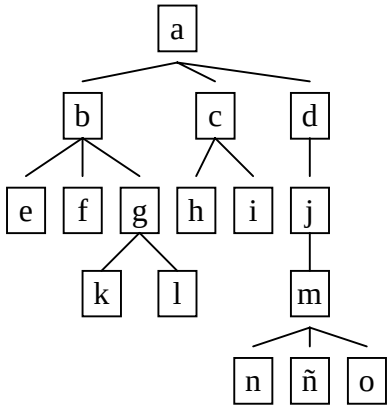
Por otro lado, tenemos los árboles B^+, que sólo almacenan las claves de los elementos insertados. Junto con cada clave hay un puntero al elemento completo. Además todos los elementos están encadenados entre sí, formando una secuencia ordenada, que facilita su recorrido secuencial a partir de cualquier posición.

5.5 Ejercicios.

1.- En la especificación algebraica de los árboles de orden N, algunas operaciones se han definido usando técnicas de inmersión para aplicarlas a los bosques de hijos. Estas técnicas de inmersión consisten en añadir un nuevo parámetro que haga el papel de contador en la programación imperativa. Definir de nuevo dichas operaciones sin usar técnicas de inmersión sino definiendo las operaciones para bosques sobre los constructores de dichos tipos.

Nota: tener en cuenta en qué operaciones el orden es significativo para el resultado de la función y si la nueva especificación lo tiene en cuenta o no.

2.- En un árbol general, el recorrido en inorden se puede definir recorriendo el primer hijo en inorden, seguido de la raíz y de los demás hijos en inorden. Indicar los recorridos en preorden, inorden y postorden del siguiente árbol general:



3.- Dibujar el árbol general a que equivale la siguiente forma canónica:

* Arbol(a, Insertar(A_1, Insertar(A_2, Insertar(A_3, Bosque_vacío))))

donde:

A_1=Arbol(b, Bosque_vacío)

A_2=Arbol(c, Insertar(A_{21}, Insertar(Arbol(f, Bosque_vacío), Bosque_vacío)))

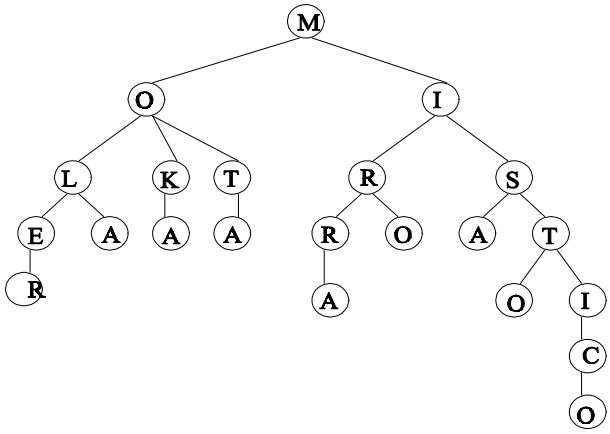
A_3=Arbol(d, Insertar(Arbol(g, Bosque_vacío), Insertar(Arbol(h, Bosque_vacío), Bosque_vacío)))

A_{21}=Arbol(e, Insertar(A_{211}, Insertar(A_{212}, Insertar(Arbol(k, Bosque_vacío), Bosque_vacío))))

A_{211}=Arbol(i, Insertar(Arbol(l, Bosque_vacío), Bosque_vacío))

A_{212}=Arbol(j, Insertar(Arbol(m, Bosque_vacío), Insertar(Arbol(n, Insertar(Arbol(ñ, Insertar(Arbol(o, Bosque_vacío), Bosque_vacío)), Bosque_vacío)), Bosque_vacío)))

4.- Suponiendo que tenemos una implementación de árboles generales, hacer un procedimiento en Modula-2 que escriba todos los caminos que van desde la raíz a cada una de las hojas, suponiendo que cada nodo contiene un carácter. P. ej., en el árbol siguiente



deberían obtenerse las palabras: MOLER, MOLA, MOKA, MOTA, MIRRA, MIRO, MISA y MÍSTICO.

5.- La estructura de datos cuad-árbol se emplea en informática para representar figuras planas en blanco y negro. Se trata de un árbol en el cual cada nodo, o bien tiene exactamente cuatro hijos, o bien es una hoja. En este último caso puede ser o bien una hoja Blanca, o una hoja Negra.

El árbol asociado a una figura dibujada dentro de un plano (que supondremos un cuadrado de lado 2^k), se construye de la forma siguiente:

- Se subdivide el plano en cuatro cuadrantes.

- Los cuadrantes que están completamente dentro de la figura corresponden a hojas Negras, y los que están completamente fuera de la región, corresponden a hojas Blancas.

- Los cuadrantes que están en parte dentro de la figura, y en parte fuera de ésta, corresponden a nodos internos; para estos últimos se aplica recursivamente el algoritmo. Como ejemplo veamos la siguientes figura y su árbol asociado:

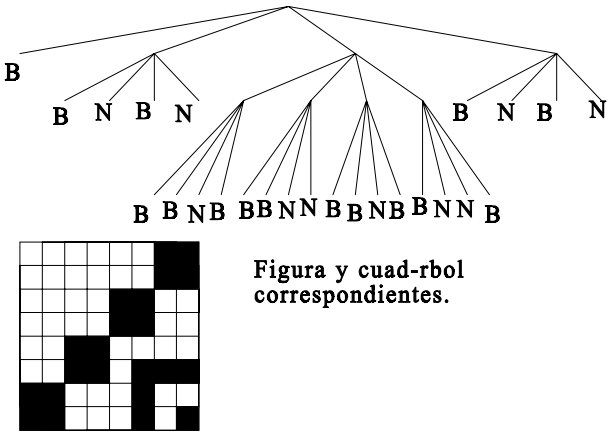


Figura y cuad-rbol correspondientes.

a) Especificar una operación que aplicada sobre una figura, la convierta en su correspondiente cuad-árbol.

b) Especificar una operación que aplicada sobre un cuad-árbol, lo convierta en su correspondiente figura.

Suponer que existe un tipo *figura* con las operaciones que se necesiten, y con la estructura que más convenga.