4 ÁRBOLES. ÁRBOLES BINARIOS.

Hasta ahora nos hemos dedicado a estudiar TADes que de una u otra forma eran de naturaleza lineal, o unidimensional. En los tipos abstractos de datos lineales existen exactamente un elemento previo y otro siguiente (excepto para el primero y el último, si los hay); en las estructuras no lineales, como conjuntos o árboles, este tipo de secuencialidad no existe, aunque en los árboles existe una estructura jerárquica, de manera que un elemento tiene un solo predecesor, pero varios sucesores.

Una exploración algo amplia en el campo de la ciencia de la computación nos lleva a situaciones en que las representaciones lineales son inadecuadas, tanto en sentido conceptual como práctico. Un paso importante lo representan los árboles binarios, y el siguiente vendrá dado con el estudio de la noción general de árbol. En capítulos posteriores, lo extenderemos hasta llegar a los grafos.

Un árbol impone una estructura jerárquica sobre una colección de objetos. Ejemplos claros de utiización de árboles se presentan tanto dentro como fuera del área de computación (indices de libros, árboles genealógicos, etc.); en Informática constituyen una de las estructuras más utilizadas, con aplicaciones que van desde los árboles sintácticos utilizados para la representación y/o interpretación de términos de un lenguaje o expresiones aritméticas, pasando por los arboles de activación de procedimientos recursivos, hasta la representación de datos que se desea mantener ordenados con un tiempo de acceso relativamente bajo. En general, se usarán árboles siempre que se quiera representar información jerarquizada, cuando esta converja en un solo punto.

4.1 ARBOLES Y SUS PROPIEDADES

Intuitivamente, podemos visualizar árboles como una forma de organizar información de forma jerárquica, con un único punto de entrada y una serie de caminos que van abriéndose en cada punto hacia sus sucesores.

Desde un punto de vista formal (teoría de conjuntos), un árbol se puede considerar como una estructura $A = (N,\prec)$, constituida por un conjunto, N, cuyos elementos se denominan nodos, y una relación de orden parcial transitiva, $\prec$, definida sobre N, y caracterizada por la existencia de

- Un elemento minimo (anterior a todos los demás) único, la raiz.

$$\exists!\ r \in N \ / \ (\forall\ n_{\neq r} \in N,\ r \prec n).$$

- Un predecesor único para cada nodo p distinto de la raiz, es decir, un nodo, q, tal que $q \prec p$ y para cualquier nodo q' con las mismas caracteristicas se cumple $q' \prec q$.

$$\forall\ n \in N \ / \ n \neq r \rightarrow (\exists!\ m \in N, ((m \prec n) \cap (\forall\ s_{\neq m} \in N \ / \ s \prec n \rightarrow s \prec m))).$$

Los ascendientes de un nodo se definen como el conjunto formado por su predecesor único y los ascendientes de éste, cuando estos existan: $Asc(n) = \{m: m \prec n\}$. *Antepasado y ancestro* suelen ser utilizados como sinónimos de ascendiente.

Los descendientes de un nodo se definen como el conjunto formado por todos aquellos nodos que lo tienen en su conjunto de ascendientes: $Desc(n) = \{m: n \prec m\} = \{m: n \in Asc(m)\}$.

Esta estructura se puede considerar una estructura recursiva teniendo en cuenta que cada nodo del árbol, junto con todos sus descendientes, y manteniendo la ordenación original, constituye también un árbol o subárbol del árbol principal, caracteristica esta que permite definiciones simples de árbol, más apropiadas desde el punto de vista de la teoria de tipos abstractos de datos, y, ajustadas, cada una de ellas, al uso que se vaya a hacer de la noción de árbol.

Las dos definiciones más comunes son las de árbol general y la de árbol de orden N, que se pueden dar en los términos siguientes:

> Un *árbol general* con nodos de un tipo T es un par (r, L_A) formado por un nodo r (la raiz) y una lista (si se considera relevante el orden de los subárboles) o un conjunto (si éste es irrelevante) L_A (bosque), posiblemente vacio, de árboles generales del mismo tipo (subárboles de la raiz). Vemos que aquí no existe el árbol vacío, sino la secuencia vacía de árboles generales.
>
> Un *árbol de orden N* (con $N \geq 2$), con nodos de tipo T, es un árbol vacio () o un par (r, L_A) formado por un nodo r (la raiz) y una tupla L_A (bosque) formada por N árboles del mismo tipo (subárboles de la raiz). Este último caso suele escribirse explicitamente de la forma $(r, A_1, ..., A_N)$.

Los nodos se clasifican dependiendo de su posición dentro del árbol en:
Raíz: Elemento mínimo de un árbol.
Nodo intermedio: Cualquier nodo predecesor de una hoja, y sucesor de la raíz.
Nodo terminal u hoja: Nodo que no tiene sucesores.

También los podemos dividir en:
Nodo interno: Cualquier nodo del árbol.
Nodo externo: Son los árboles vacíos que penden de los nodos que no tienen todos sus hijos, (en los árboles de orden N). Se representa por □.

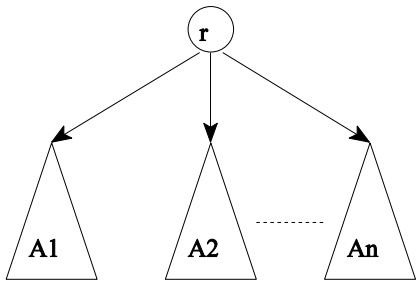
Conceptos más importantes son:
Padre: Predecesor máximo de un nodo.
Hijo: Cualquiera de los sucesores directos de un nodo
Hermano: Cualquier otro nodo hijo de un mismo padre.

A cada par de nodos (A, B) tal que B es hijo de A se le denomina *eje*; si $n_l, n_2, ...n_k$, es una sucesión de nodos de un árbol tal que n_i es el padre de n_{i+l} (el par (n_i, n_{i+l}) es un eje) para $l \leq i < k$, entonces la secuencia se denomina *camino del nodo n_l al nodo n_k*. A cada camino se le suele asociar una *longitud*, que se define como el número de ejes que lo componen. Serán *ramas* aquellos caminos cuyo primer nodo sea el raiz y cuyo último nodo es una hoja.

Altura de un nodo en un árbol es el número de nodos del camino más largo de ese nodo a una hoja.
Altura del árbol es la altura de la raíz, o 0 si el árbol es vacío.
Profundidad de un nodo es la longitud del camino único que va desde la raiz hasta ese nodo.

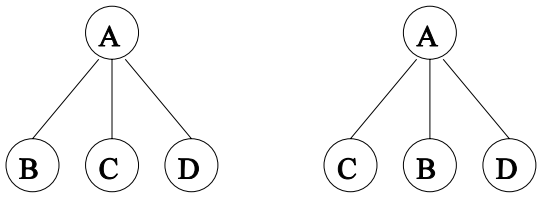
Se denomina *grado de un nodo* al número de hijos de dicho nodo.
El *grado de un árbol* es el mayor grado de los nodos que contiene.
El *nivel de un nodo* se asigna en función al criterio siguiente:
La raiz tiene nivel 1.
Si un nodo tiene nivel N, sus hijos tendrán nivel N+l.
El nº de niveles de un árbol es igual a la altura de su raíz, o a 0, si el árbol es vacío.

Conceptos menos utilizados son:
Longitud del camino externo: Suma de los niveles de cada nodo externo.
Longitud del camino interno: Suma de los niveles de cada nodo interno.



Se utilizan distintas notaciones gráficas para representar árboles. Entre ellas la más usual es el diagrama de árbol invertido, que consiste en utilizar un "árbol invertido", representando la raíz en la parte superior, y, debajo de ella, de izquierda a derecha, cada uno de sus subárboles:

Los hijos de un nodo de orden N pueden estar ordenados o desordenados. Los árboles resultantes son ordenados y desordenados respectivamente. Los siguientes árboles ordenados no son equivalentes:



O sea, podemos ordenar o no a los hijos de un nodo, lo cual dará lugar a diferentes conceptos de igualdad.

En árboles de orden N se define:
Árbol lleno: es un árbol con todos sus niveles llenos. También podemos definirlo como el árbol en el que la longitud del camino más largo y la del más corto desde cualquier nodo, son iguales.
Árbol Completo es un árbol con todos sus niveles llenos salvo quizás el último, que deberá estar completo, (sin "huecos") de izquierda a derecha.
Árbol perfectamente balanceado: Es aquél en el que el número de nodos de cada subárbol de cada nodo interno, no varía en más de uno. La palabra *equilibrado* suele emplearse como sinónimo de *balanceado*.
Árbol balanceado en altura como el árbol cuyos subárboles tienen alturas que difieren a lo más en una unidad y también son equilibrados en altura.
Árbol degenerado: Es aquél en el que cada nodo sólo tiene un subárbol. Equivale a una lista.

4.2 ARBOL BINARIO

El árbol binario es el caso más simple de árbol de orden N, cuando N vale 2. Su especificación se puede hacer considerando un valor constante, el árbol nulo, y un constructor de árboles a partir de un elemento y dos árboles.

4.2.1 Especificación algebraica de árboles binarios

tipo ArbolB;
dominios ArbolB, Elemento, Lógico
generadores
Crear: $\longrightarrow$ ArbolB
Arbol_binario: Elemento × ArbolB × ArbolB $\longrightarrow$ ArbolB
constructores
Hijo izq: ArbolB $\not\longrightarrow$ ArbolB
Hijo dch: ArbolB $\not\longrightarrow$ ArbolB
selectores
Raíz: ArbolB $\not\longrightarrow$ Elemento
Es_Vacio: ArbolB $\longrightarrow$ Lógico
precondiciones a: ArbolB
Raiz(A): not Es_Vacío(a)
Hijo_izq(a): not Es_Vacío(a)
Hijo_dch(a): not Es_Vacío(a)
ecuaciones r: Elemento; i, d: ArbolB
Es_Vacío(Crear) == V
Es_Vacío(Arbol_binario(r, i, d)) == F
Raiz(Arbol_binario(r, i, d)) == r
Hijo_izq(Arbol_binario(r, i, d)) == i
Hijo_dch(Arbol_binario(r, i, d)) == d
fin

La especificación anterior desempeña un papel básico, como ocurre con la de listas respecto a los demás tipos lineales, que consiste en facilitar la construcción de otras especificaciones de árboles, en los que se recurre al esquema de árbol binario y sus operaciones para la expresión de operaciones de inserción, extracción, rotación, etc. más complejas, y permitir la formalización de conceptos y nociones ligadas a la de árbol binario.

4.2.2 Propiedades de los árboles binarios

De la especificación anterior se deducen las siguientes formas canónicas para los términos del tipo ArbolB:

Crear
Arbol_binario(r, I, D)

siendo I y D dos términos del mismo tipo también en forma canónica.

A partir de la especificación, y como se hizo en casos anteriores, se pueden definir los distintos conceptos relacionados con la noción de árbol binario:

● Un primer grupo corresponde a las nociones de igualdad de forma, número de nodos y altura.

Igual_Forma: ArbolB × ArbolB ⟶ Lógico

ecuaciones

Igual_Forma(Crear, a) == Es_Vacío(a)

Igual_Forma(Arbol_binario(r, i, d), a)==

SI Es_Vacío(a) ENTONCES

F

SI NO

Igual_Forma(i, Hijo_izq(a)) and Igual_Forma(d, Hijo_dch(a))

Número_Nodos: ArbolB ⟶ ℕ

ecuaciones

Número_Nodos(Crear) == 0

Número_Nodos(Arbol_binario(r, i, d)) == 1 + Número_Nodos(i) + Número_Nodos(d)

Altura: ArbolB ⟶ ℕ

ecuaciones

Altura(Crear) == 0;

Altura(Arbol_binario(r, i, d)) == 1 + Máximo(Altura(i), Altura(d));

● Un segundo grupo aparece al considerar una relación de igualdad entre los valores del tipo Elemento. Entonces se puede hablar de igualdad de árboles y de la relación de existencia de un elemento en un árbol.

Está: Elemento × ArbolB ⟶ Lógico

ecuaciones

Está(e, Crear) == F

Está(e, Arbol_binario(r, i, d)) == (e = r) or Está(e, i) OR Está(e, d)

Igual: ArbolB × ArbolB ⟶ Lógico

ecuaciones

Igual(Crear, a) == Es_Vacío(a)

Igual(Arbol_binario(r, i, d), a)==

SI Es_Vacío(a) ENTONCES

F

SI NO

(r=Raíz(a)) and Igual(i,Hijo_izq(a)) and Igual(d,Hijo_dch(a))

Como vemos, aquí sí nos importa el orden en que se disponen los subárboles.

● Un tercer grupo aparece al considerar las posibles formas de recorrer los nodos de un árbol. Con ayuda de las especificaciones de listas funcionales podemos especificar distintas operaciones tipicas de recorrido de árboles como funciones de árboles binarios en listas de nodos.

Nivel: $\mathbb{N} \times$ ArbolB ⟶ Lista

ecuaciones

```
Nivel(0, a) == Crear
Nivel(suc(n), Crear) == Crear
Nivel(suc(n), Arbol_binario(r, i, d)) ==
        SI n=0 ENTONCES
                Insertar(r, Crear)
        SI NO
                Concatenar(Nivel(n, i), Nivel(n, d))
```

Preorden, Inorden, Postorden: ArbolB ⟶ Lista

ecuaciones

```
Preorden(Crear) == Crear
Preorden(Arbol_binario(r, i, d)) ==
        Construir(r, Concatenar(Preorden(i), Preorden(d)))
Inorden(Crear) == Crear
Inorden(Arbol_binario(r, i, d)) ==
        Concatenar(Inorden(i), Construir(r, Inorden(d)))
Postorden(Crear) == Crear
Postorden(Arbol_binario(r, i, d)) ==
        Concatenar(Concatenar(Postorden(i), Postorden(d)), Construir(r, Crear))
```

● Se pueden especificar predicados (para árboles de elementos comparables) que determinen si dichos elementos están distribuidos en el árbol con arreglo a alguna de las ordenaciones o recorridos; para ello empleamos el predicado Está_Ordenada, que se supone definido para listas de nodos.

Preordenado, Inordenado, Postordenado: ArbolB ⟶ Lógico

ecuaciones

```
Preordenado(a) == Está_Ordenada(Preorden(a))
Inordenado(a) == Está_Ordenada(Inorden(a))
Postordenado(a) == Está_Ordenada(Postorden(a))
```

● Predicados para árbol lleno, árbol completo y árbol equilibrado en altura.

Es_Lleno, Es_Completo, Es_Equilibrado: ArbolB ⟶ Lógico

ecuaciones

```
Es_Lleno(Crear) == V
Es_Lleno(Arbol_binario(r, i, d)) ==   (Altura(i) = Altura(d)) and
                                      Es_Lleno(i) and
                                      Es_Lleno(d)

Es_Completo(Crear) == V
Es_Completo(Arbol_binario(r, i, d)) ==
        (Altura(i) = Altura(d)) and Es_Lleno(i) and Es_Completo(d))
                or
        (Altura(i) = 1+Altura(d)) and Es_Completo(i) and Es_Lleno(d))
```

Es_Equilibrado(Crear) == V
Es_Equilibrado(Arbol_binario(r, i, d)) == (-1 ≤ Altura(i) - Altura(d) ≤ 1) and
Es_Equilibrado(i) and Es_Equilibrado(d)

Utilizando estas definiciones se pueden demostrar, por inducción estructural, una serie de propiedades típicas de los árboles binarios:

★ Para a: ArbolB y j: ℕ se cumple
$0 < j \leq Altura(a) \rightarrow Longitud(Nivel(j, a)) \leq 2^{j-1}$

Demostración:

☆ Para a == Arbol_binario(r, Crear, Crear), la implicación es trivialmente cierta.

☆ (h. i.) $Longitud(Nivel(j, i)) \leq 2^{j-1}$, y $Longitud(Nivel(j, d)) \leq 2^{j-1}$

☆ (p. i.) Para a == Arbol_binario(r, i, d), suponemos que se cumple para i y d.

Longitud(Nivel(Suc(j), Arbol_binario(r, i, d)))
== SI j = 0 ENTONCES
Longitud(Insertar(r, Crear))
SI NO
Longitud(Concatenar(Nivel(j, i), Nivel(j, d)))
== SI j = 0 ENTONCES
1
SI NO
Longitud(Nivel(j, i)) + Longitud(Nivel(j, d))

y por la h. i.

$Longitud(Nivel(j, i)) + Longitud(Nivel(j, d)) \leq 2^{j-1} + 2^{j-1} = 2^j$

podemos concluir pues, que, en cualquiera de los dos casos:

$Longitud(Nivel(Suc(j), Arbol\ binario(r, i, d))) \leq 2^{Suc(j)-1}$

★ En particular, para a: ArbolB y j: ℕ se cumple:
$(Es\_Lleno(a)\ and\ (0 < j \leq Altura(a))) \rightarrow Longitud(Nivel(j, a)) = 2^{j-1}$

Esta propiedad es fácil de demostrar de forma parecida a la anterior.

4.3 FUNCIONES DE ORDEN SUPERIOR.

Vamos a ver en este epígrafe un nuevo tipo suministrado por MODULA-2: el tipo PROCEDURE. A diferencia de los tipos conocidos hasta ahora, las variables declaradas de tipo PROCEDURE, no están destinadas a contener datos propiamente dichos, sino que su cometido es simplemente 'apuntar' a procedimientos, o lo que es lo mismo, sirven para poder designar con un alias (otro nombre), a procedimientos ya creados.

P.ej, supongamos que tenemos el tipo:

```
TYPE
        FUNCION_REAL = PROCEDURE(REAL) : REAL;
```

y la variable

```
VAR
        p : FUNCION_REAL;
```

Entonces, dado que las funciones trigonométricas están declaradas de la forma:

```
PROCEDURE Sin(x : REAL) : REAL;
PROCEDURE Cos(x : REAL) : REAL;
PROCEDURE Tan(x : REAL) : REAL;
```

pues podemos hacer asignaciones de la forma:

```
        p := Sin; ó p := Cos; ó p := Tan;
```

dado que tanto el tipo FUNCION_REAL, y los procedimientos Sin, Cos, y Tan, tienen argumentos del mismo tipo, y producen resultados también del mismo tipo.

Posteriormente, podemos hacer cosas tales como:

```
Opcion := RdChar();
CASE Opcion OF
        | 'S' : p := Sin;
        | 'C' : p := Cos;
        | 'T' : p := Tan;
END;
WrStr("Introduzca argumento: "); RdReal(x);
WrStr("El resultado es: "); WrReal(p(x), 7, 3); WrLn;
```

Supongamos ahora que tenemos una lista de tipo base CARDINAL, y se desea construir un procedimiento Duplica, que toma la lista, y devuelve una lista en la que cada elemento se ha multiplicado por dos. P.ej.: Duplica([1, 3, 14, 6]) = [2, 6, 28, 12].

Así, se tendría:

```
PROCEDURE Duplica(l : LISTA) : LISTA;
BEGIN
        IF Es_vacia(l) THEN
                RETURN Crear;
        ELSE
                RETURN Construir(2*Cabeza(l), Duplica(Cola(l)));
        END
END Duplica;
```

Sin embargo, si escribimos el procedimiento de la siguiente forma:

```
TYPE
        MONADICO = PROCEDURE(ITEM) : ITEM;

PROCEDURE Map(Func : MONADICO; l : LISTA) : LISTA;
BEGIN
```

```
        IF Es_vacia(l) THEN
                RETURN Crear();
        ELSE
                RETURN(Construir(Func(Cabeza(l)), Map(Func, Cola(l)));
        END;
END Map;

PROCEDURE Multiplica_por_2(n : ITEM) : ITEM;
BEGIN
        RETURN 2*n;
END Multiplica_por_2;
```

podríamos obtener el mismo resultado mediante una llamada de la forma:

```
Map(Multiplica_por_2, [1, 3, 14, 6]);
```

donde [] representa una lista.

Así, conseguimos un método general de procesamiento de todos y cada uno de los elementos que componen la lista. Gracias a que a la función **Map** se le pasa como parámetro una función Func() que se aplicará a todos y cada uno de los elementos de la lista, dependiendo del comportamiento de Func(), pues así procesaremos de una forma u otra a los elementos de la lista.

Se le suele dar el nombre de **Map** (aplicación), puesto que cada elemento origen se aplica en uno destino. En definitiva, Map es una operación de *recorrido* que aplica una función pasada como parámetro a todo elemento visitado.

En contraposición a este método, se tienen las funciones de reducción, que procesan todos y cada uno de los elementos de una lista, y dan como resultado un solo elemento, dependiendo de lo que se quiera hacer: el menor, el mayor, la media, la suma, el producto, etc.. A esta operación se la suele llamar **Reduce**.

Reduce exige que la función que se le pasa como argumento sea al menos de 2 parámetros: uno que contendrá el valor reducido hasta el momento de una nueva aplicación , y otro el valor procesado en ese momento, en esa aplicación (la operación se aplica una vez para cada elemento). También es posible hacer operaciones Reduce que pasen más información, p.ej. la posición, la longitud total, etc., lo cual permitiría el cálculo de la media y de la mediana.

Así, para proceder al cálculo del mayor valor de una lista, se tendría:

```
PROCEDURE Mayor(l : LISTA) : ITEM;
VAR
        Cont : CARDINAL;
        MAYOR : ITEM;
BEGIN
        FOR Cont := 1 TO Longitud(l) DO
                IF Cont := 1 THEN
                        MAYOR := Elemento(l, 1);
                ELSIF MAYOR < Elemento(l, Cont) THEN
```

```
                    MAYOR := Elemento(l, Cont);
                END;
        END;
END Mayor;
```

Sin embargo, con un procedimiento general de reducción, se tendría:

```
TYPE
        FUNC = PROCEDURE(ITEM, ITEM) : ITEM;

PROCEDURE Reduce(Func : FUNC; l : LISTA; Inicial : ITEM) : ITEM;
BEGIN
        IF Es_vacia(l) THEN
                RETURN Inicial;
        ELSE
                RETURN Func(Cabeza(l), Reduce(Func, Cola(l), Inicial));
        END;
END Reduce;

PROCEDURE Mayor(a, b : ITEM) : ITEM;
BEGIN
        IF a > b THEN
                RETURN a
        ELSE
                RETURN b
        END;
END Mayor;
```

Y la llamada sería de la forma:

```
Reduce(Mayor, [1, 3, 13, 6], 0);
```

que devolvería el valor **13.**

NOTA: Una implementación de **Reduce** quizás más intuituiva sería:

```
PROCEDURE Reduce(Func: FUNC; l : LISTA; Inicial : ITEM) : ITEM;
BEGIN
        IF Es_vacia(l) THEN
                RETURN Inicial
        ELSE
                RETURN Reduce(Func, Cola(l), Func(Cabeza(l), Inicial));
        END;
END Reduce;
```

Como se ha dicho, Reduce lo que hace es compactar todos los valores de una estructura, a un único valor.

Tal y como se da aquí, posee algunos inconvenientes:

- Si queremos compactar los valores a una estructura diferente de la de dichos valores, necesitaremos una función Reduce diferente. Esto ocurre por ejemplo cuando se intenta convertir un árbol en una lista.
- Por otro lado, hay funciones que no permite calcular, como pueda ser la mediana (valor central

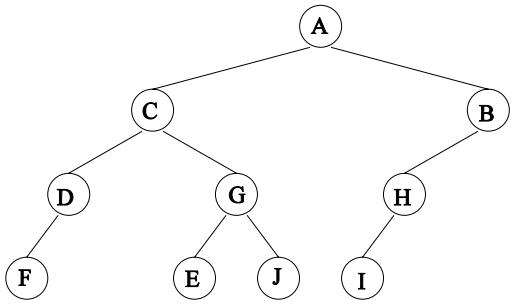
de una lista), o la moda, a pesar de que ambos valores dependen de todos los elementos de la lista.
- Hay cálculos que requieren parámetros diferentes a los aquí expuestos; p.ej., el número de repeticiones de un valor en una estructura, la posición que ocupa un elemento en una estructura, etc..

4.4 PROCESAMIENTO DE ÁRBOLES.

Podemos estudiar el procesamiento de árboles desde dos puntos de vista: en amplitud, y en profundidad.

4.4.1 Procesamiento en amplitud.

Con esta técnica se pretende visitar los nodos de un árbol, por niveles: primero los del nivel 1, luegos los del nivel 2, luegos los del 3, y así sucesivamente. En el siguiente ejemplo, los nodos se visitarían en el siguiente orden:



A, C, B, D, G, H, F, E, J, I.

P.ej., podemos devolver una lista con los elementos en este orden,mediante la operación:

Amplitud : ArbolB ⟶ *Lista*

auxiliares

Genera : ArbolB × ℕ × ℕ ⟶ Lista

ecuaciones n, m : ℕ r : Elemento a, i, d : ArbolB

Amplitud(a) == Genera(a, 1, Altura(a));

Genera(a, n, m) == SI n ≤ m ENTONCES
Concatena(Nivel(n, a), Genera(a, suc(n), m))
SI NO
Crear

donde la operaciones Nivel y Altura ya fueron definidas en puntos anteriores.

4.4.2 Procesamiento en profundidad.

Hay tres métodos para visitar un árbol en profundidad: *preorden, inorden, y postorden.*

En inorden, se visita la raíz entre la visita de los subárboles izquierdo y derecho, o sea, se visita primero el subárbol izquierdo, después la raíz, y por último el subárbol derecho. En el caso de árboles generales o de orden N, se visita primero el árbol más a la izquierda, a continuación la raíz, y por último el resto de subárboles en secuencia.

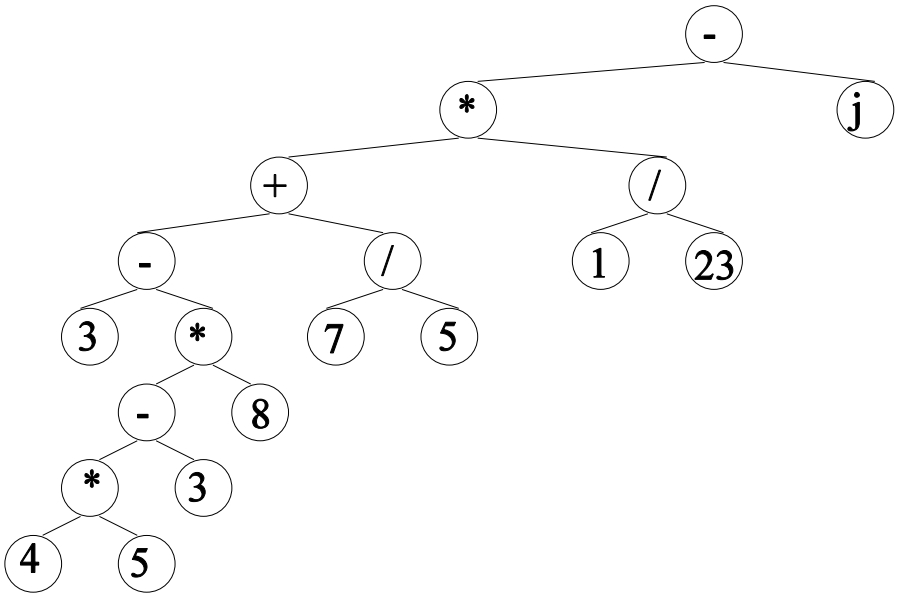
En preorden, la raíz se visita antes de visitar los subárboles izquierdo y derecho en

preorden.

Por último, en postorden, primero se visitan el subárbol izquierdo, luego el derecho (ambos en postorden), y por último la raíz.

Siguiendo el diagrama de antes, la secuencia de nodos visitados sería la siguiente para cada tipo de recorrido:

PREORDEN: A, C, D, F, G, E, J, B, H, I.
INORDEN: F, D, C, E, G, J, A, I, H, E.
POSTORDEN: F, D, E, J, G, C, I, H, E, A.



Este tipo de recorridos es el que suele utilizar cuando se quieren implementar operaciones **Map** y **Reduce** con árboles.

Los árboles binarios se emplean a menudo para la representación de expresiones aritméticas, dado que una operación con dos operandos la podemos representar como un árbol cuya raíz sea el operador, y sus subárboles sean los operandos. P.ej., la operación: (3-(4*5-3)*8+7/5)*1/23-j, equivaldría, rellenando paréntesis, a: (((3-(((4*5)-3)*8))+(7/5))*(1/23))-j, y su árbol sería el de la figura.

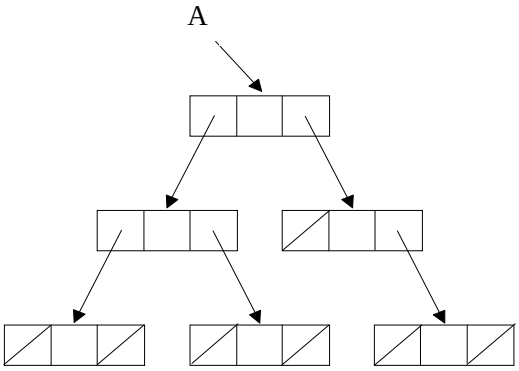
El recorrido del tal árbol en inorden equivale a la notación polaca inversa (*RPN-Reverse Polish Notation*); en inorden equivale a la notación infija a que estamos acostumbrados; y en preorden equivale a una evaluación impaciente, (en el momento en que hay una operación y dos operandos, se opera).

4.5 IMPLEMENTACIÓN DE ÁRBOLES

4.5.1 Representaciones dinámicas

Hay diferentes posibilidades en función de que se consideren estructuras:
· con enlaces simples o dobles.
· con cabecera o sin cabecera.

4.5.1.1 Estructura dinámica de registros con enlaces a los hijos.

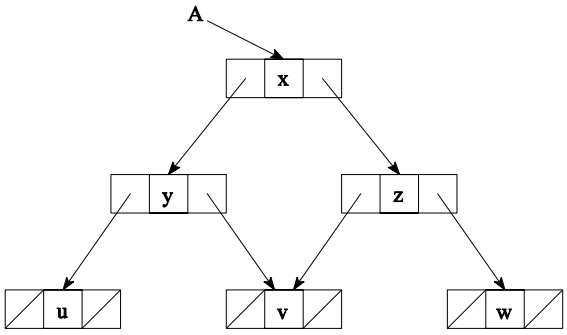



```
TYPE
        ARBOL = POINTER TO NODO;
        NODO =      RECORD
                        valor: ITEM;
                        izq, dch: ARBOL
                    END;
```

La situación de la siguiente figura se presentará con árboles construidos de la siguiente forma:

D := Arbol_binario(v, Crear, Crear);
C := Arbol_binario(y, Arbol_binario(u, Crear, Crear), D);
B := Arbol_binario(z, D, Arbol_binario(w, Crear, Crear));
A := Arbol_binario(x, B, C);

Sin embargo, este tipo de construcciones no suele ser deseable, ya que un cambio en un nodo produce como efecto lateral ese mismo cambio en otro nodo. P.ej., si implementamos la función Map con este árbol, y la llamamos con la función Multiplica_por_2, resulta que los elementos compartidos los multiplicará por 4 (en este caso), o por 2^n, siendo **n** el número de veces que es compartido.



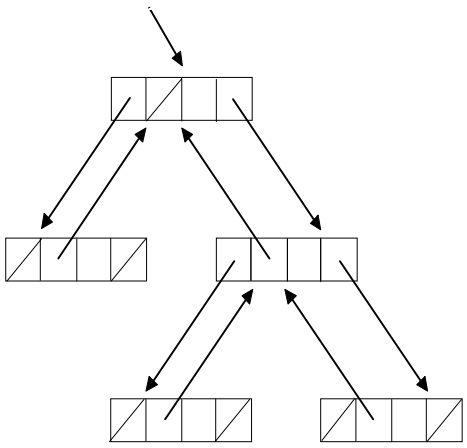
· Permite obtener una implementación fácil y rápida para todas las operaciones.
· Permite aplicar de forma casi directa las operaciones de la misma forma recursiva con que las hemos visto en la especificación algebraica.

4.5.1.2 Estructura de registros dinámicos con enlaces a los hijos y al padre

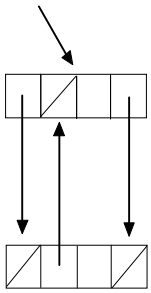
Permite efectuar retrocesos en el árbol sin necesidad de salir de puntos profundos en recursividad (backtracking). La implementación sería:

```
TYPE
        ARBOL = POINTER TO NODO;
        NODO =      RECORD
                        valor: ITEM;
                        ant, izq, dch: ARBOL
                    END;
```

A



B



El árbol B, demuestra de nuevo un caso de compartición estructural, que debe evitarse en condiciones normales.

· Presenta la ventaja de permitir recorridos fáciles de forma no recursiva.
· Permite obtener caminos ascendentes a partir de cualquier nodo.

El siguiente pseudocódigo corresponde a un recorrido en postorden no recursivo:

```
actual := raiz
BAJA
REPETIR HASTA QUE actual = NIL
        SI BAJA ENTONCES
                SI actual^.izq ≠ NIL ENTONCES
                        actual := actual^.izq
                SI NO SI actual^.dch ≠ NIL ENTONCES
                        actual := actual^.dch
                SI NO
                        Visitar actual
                        anterior := actual
                        actual := actual^.padre
                        SUBE
                FIN SI
        SI NO SI SUBE
                SI anterior = actual^.izq ENTONCES
                        SI actual^.dch ≠ NIL ENTONCES
                                BAJA
                                actual := actual^.dch
                        SI NO
                                Visitar actual
                                anterior := actual
                                actual := actual^.padre
                                SUBE
                        FIN SI
                SI NO SI anterior := actual^.dch ENTONCES
```

```
                    Visitar actual
                    anterior := actual
                    actual := actual^.padre
                    SUBE
                FIN SI
            FIN SI
FIN REPETIR
```

4.5.1.3 Estructura de registros dinámicos con enlaces a los hijos

Esta estructura también se conoce como: *enhebrada*

Aquí, cada nodo va a constar de cinco campos:
- Valor que se desea almacenar.
- Puntero izquierdo.
- Cometido del puntero izquierdo.
- Puntero derecho.
- Cometido del puntero derecho.

En condiciones normales, los punteros izquierdo y derecho apuntan a los subárboles izquierdo y derecho. Ahora bien, cuando alguno de estos subárboles no existe, se reutilizan estos punteros para otras cosas. La existencia o ausencia de subárboles viene determinada por los campos de cometido de punteros, que estarán a Verdad cuando el subárbol es vacío, y a Falso en caso contrario.

Si el subárbol izquierdo de un nodo está vacío, reutilizaremos el puntero correspondiente para apuntar a su predecesor en inorden.

Si el subárbol derecho de un nodo está vacío, reutilizaremos el puntero correspondiente para apuntar a su sucesor en inorden.

En definitiva, se trata de una estructura de registros dinámicos con enlaces a los dos hijos, en la que los enlaces que están a NIL se utilizan con el propósito de facilitar los recorridos por el árbol, en inorden:

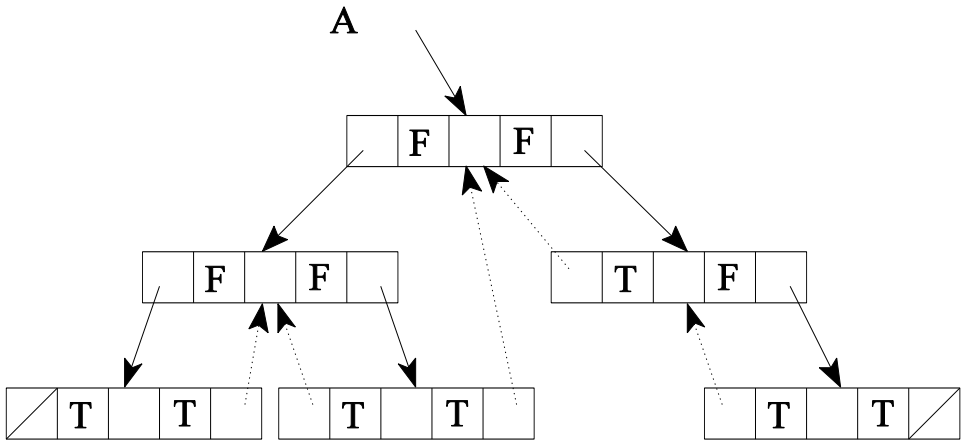
- ○ Cuando se trata de un hijo izquierdo, para enlazar con el nodo anterior en inorden.
- ○ Cuando se trata de un hijo derecho, para enlazar con el nodo siguiente también en inorden.

Esto obliga a introducir en los registros un par de campos con valor lógico que sirvan para distinguir el papel que desempeñan los enlaces en cada momento (si actúan como enlaces a los hijos o como *hebras* hacia nodos superiores).

Nótese que surge la pregunta: ¿a qué apunta el puntero izquierdo del primer nodo visitado en inorden?, ¿y el puntero derecho del último nodo, también en inorden?. Podemos hacerlo a nuestro gusto:

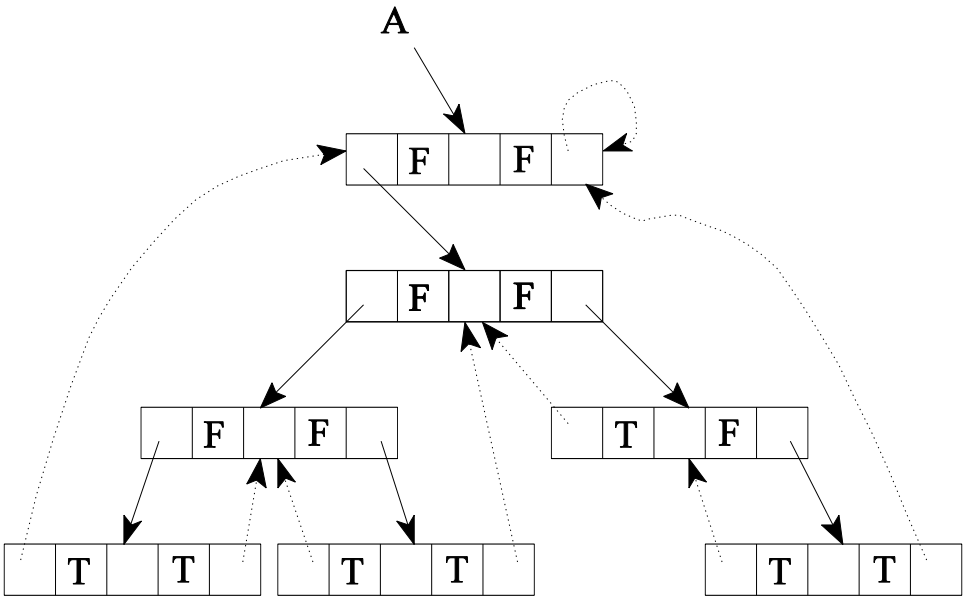
- ▶ Podemos hacer que apunten a sí mismos.

▶ Podemos hacerlos NIL:



▶ Se puede utilizar un registro de cabecera, del mismo tipo, con el campo valor vacío, el puntero izquierdo enlazando con el árbol en sí, y el derecho consigo mismo. Los campos de cometido estarían a Falso. Los punteros que nos hemos preguntado antes, enlazarían con esta estructura.

Esta estructura es interesante para resolver el ejericio 16 de los propuestos al final del tema.



TYPE

```
ARBOL = POINTER TO NODO;
NODO =      RECORD
                valor: ITEM;
                hebra_izq, hebra_dch: BOOLEAN;
                izq, dch: ARBOL
            END;
```

Esta representación se adapta mal a la construcción de árboles binarios mediante ArbolBinario, HijoIzq e HijoDch, pues implicaría la creación de copias, lo cual por otro lado puede no ser un problema si se desea evitar a toda costa la compartición estructural. La necesidad de copias aparece debido a los encadenamientos existentes, lo que hace que la estructura no sea recursiva totalmente (un subárbol posee enlaces con nodos extraños a él).

Se comporta bien con inserción de nodos, y con los distintos recorridos; la eliminación de nodos es sumamente compleja, pues posee una extensa casuística.

Como se ha dicho, una de las ventajas de esta implementación, es la eliminación de la recursividad a la hora de hacer recorridos; si hay una gran profundidad, la recursividad puede llegar a un punto en el que se acabe la memoria. El uso de árboles enhebrados permite efectuar recorridos sin necesidad de recursividad.

Suponiendo que los nodos sin predecesor ni sucesor en el recorrido en inorden, apuntan a NIL, podemos hacer una función que devuelva el sucesor de un nodo en inorden:

```
PROCEDURE Sucesor(a : ArbolB) : ArbolB;
VAR
        b : ArbolB;
BEGIN
        IF a^.hebra_d THEN
                RETURN a^.dch;
        ELSE
                b := a^.dcha
                WHILE NOT (b^.hebra_i) DO
                        b := b^.izq;
                END;
                RETURN b;
        END;
END;
```

El recorrido en inorden consiste simplemente en busca el primer elemento, y a partir de él sus sucesores:

```
PROCEDURE Inorden(a : ArbolB);
VAR
        b : ArbolB;
BEGIN
        b := a;
        WHILE NOT (b^.hebra_i) DO
                b := b^.izq;
        END;
```

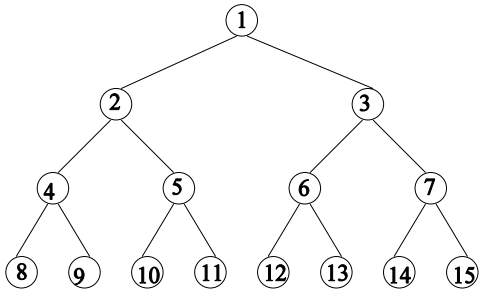
```
        WHILE b^.dch # NIL DO
                b := Sucesor(b);
        END;
END Inorden;
```

4.5.2 Representaciones estáticas

4.5.2.1 Dispersa. en amplitud.

Se almacenan los subárboles siguiendo el recorrido en amplitud, de manera que cada nodo ocupa una posición en la tabla, dependiendo de la posición que ocupa en el árbol, suponiendo que éste está Lleno (se respetan las posiciones no ocupadas).

La posición que ocupará cada nodo puede apreciarse en la figura adjunta.



Características:

- Acceso directo a cualquier subárbol de un determinado nivel utilizando la correspondencia entre niveles del árbol y segmentos de la tabla
nivel i: segmento[2^{i-1}, 2^{i}-1]

- El paso de un subárbol situado en la posición j a su hijo izquierdo o derecho, que estarán en las posiciones 2j y 2j+1 respectivamente, y a su padre, que estará en la posición j DIV 2

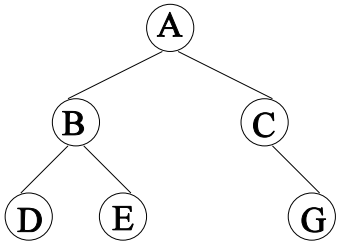
- Nos ahorramos la existencia de punteros; sin embargo, los nodos vacíos también ocupan espacio. Se empleará pues para árboles casi llenos, o completos.

- Será necesario un valor especial o campo adicional para indicar los nodos vacíos.

```
CONS
        ALTURA_MAX = -----
        MAX = 2**ALTURA_MAX-1;
TYPE
        ARBOL = POINTER TO ESTRUCTURA;
        ESTRUCTURA =        RECORD
                                nodos: ARRAY [1..MAX] OF PAR;
                            END;
      PAR =     RECORD
                        libre: BOOLEAN;
                        valor: ITEM;
                END
```

La estructura se ha mantenido como registro por si se desea introducir algún campo adicional de cabecera.

El árbol siguiente se representaría como sigue:



Posición	Valor	Es_vacío
1	A	F
2	B	F
3	C	F
4	D	F
5	E	F
6	--	V
7	G	F

Por supuesto, la construcción de árboles mediante ArbolBinario, HijoIzq e HijoDch implica copia de estructuras,aunque puede ser interesante en ciertos casos de árboles binarios de búsqueda en los que se desea mucha velocidad de acceso.

Es muy cómoda para representar árboles completos que no se modifican, y sólo se consultan.

Aparte de los problemas típicos de las estructura estáticas, aquí se tiene el inconveniente de que el desperdicio de espacio es enorme en los árboles poco balanceados.

4.5.2.2 Compacta. En preorden

Aquí se pretende hacer una mezcla entre una representación compacta, y una con cursores. Cada nodo posee un cursor a la raíz de su árbol derecho, mientras que la raíz del izquierdo se almacena justo a continuación.

Hay un mejor aprovechamiento de la tabla para niveles no completos; sin embargo, pueden ser necesarias costosas reorganizaciones de elementos. Véase que no existen huecos entre elementos, y que no se emplea ningún valor especial para denotar huecos (porque no existen), por lo que es necesario un campo Longitud para saber cual es el primer campo vacío.

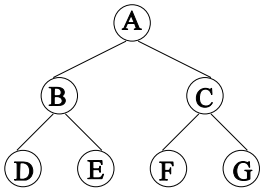
La definición del tipo sería:

```
CONS
        MAX = -----
TYPE
        ARBOL = POINTER TO ESTRUCTURA;
```

```
ESTRUCTURA =        RECORD
                            Longitud : [0..MAX];
                            nodos: ARRAY [1..MAX] OF PAR;
                    END;
PAR =    RECORD
                    derecho : [0..MAX]
                    valor : ITEM;
         END;
```

Esta estructura facilita la localización de hijos, aunque por ser una estructura estática, la construcción de hijos con ArbolBinario resulta muy costosa, ya que se requiere una enorme actualización de enlaces, y movimiento de nodos.

El recorrido en preorden es trivial; para cualquier otro el recorrido debe ser recursivo.



Posición	Valor	Posición hijo dcho.
1	A	5
2	B	4
3	D	0
4	E	0
5	C	7
6	F	0
7	G	0

Por ejemplo, el siguiente algoritmo efectúa un recorrido en inorden siguiendo este tipo de estructura:

```
Algoritmo Inorden.
Entrada: a: ARBOL; inicio, final : [0..MAX]
    Si f < i Entonces Retornar Fin Si
    Si a^.nodos[i].derecho = 0 Entonces
      Inorden(a, i+1, f)
    Si No
      Inorden(a, i+1, a^.nodos[i].derecho - 1)
    Fin Si
    Visitar a^.nodos[i].valor
    Si a^.nodos[i].derecho ≠ 0 Entonces
      Inorden(a, a^.nodos[i].derecho, f)
    Fin Si
Salida: --
```

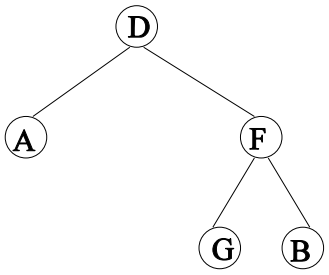
4.5.2.3 Cursores.

Al igual que se podía efectuar una implementación de listas en base a cursores, los árboles también son susceptibles de ser almacenados en una estructura similar.

La única diferencia entre el almacenamiento estático con cursores, y el dinámico con punteros, es que en el primero el número de nodos ya está predeterminado, es constante, y se hallan todos uno detrás de otro en secuencia, por lo que pueden ser denotados por su posición, también llamada cursor, en lugar de por un puntero. Asimismo, es necesario mantener una lista de nodos libres.

En definitiva, esta nueva alternativa consiste en emplear una tabla de registros de tres campos:
- Valor del nodo.
- Posición de la raíz del subárbol izquierdo.
- Posición de la raíz del subárbol derecho.

A continuación aparece un ejemplo de almacenamiento de un árbol:



	Valor	Hijo izq.	Hijo dch.
1	D	3	2
2	F	7	4
3	A	0	0
4	B	0	0
5			
6			
7	G	0	0

La definición del tipo es:

```
TYPE
    MAX = 100;
    CURSOR = [0..MAX];
    NODO =          RECORD
                        valor : ITEM;
                        izquierdo, derecho : CURSOR;
                    END;
    ARBOL = POINTER TO ESTRUCTURA;
    ESTRUCTURA =    RECORD
                        Libre : CURSOR;
                        Raíz : CURSOR;
                        Nodos : ARRAY [1..MAX] OF NODO;
                    END;
```

La posición de la raíz vendrá dada evidentemente por el valor del campo Raíz.

Este tipo de estructura tiene los problemas típicos en cuanto a la limitación en el número de nodos, desperdicio de espacio, acceso a posiciones en lugar de a direcciones, etc..

4.6 IMPLEMENTACIÓN DEL TAD ÁRBOL BINARIO.

Como es usual se presentan dos posibles implementaciones: dinámica, usando variables puntero, o estática, basadas en arrays; además de múltiples variantes en cada una de estas modalidades. Cualquiera de ellas tendrá, sin embargo, un mismo módulo de definición:

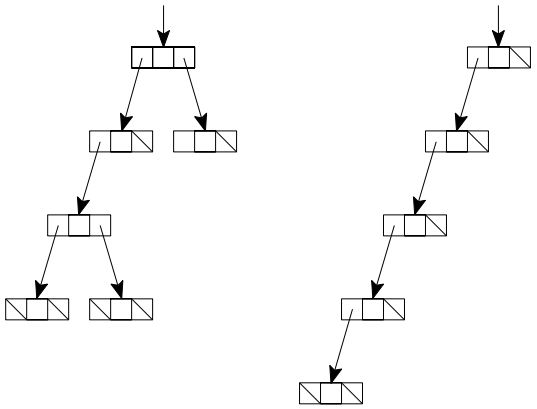
```
DEFINITION MODULE ArbolB;
TYPE
        ITEM = << cualquier tipo >>;
        ArbolB;
        ERROR = (SinError, ArbolVacio, SinMemoria);
VAR
        error : ERROR;
PROCEDURE Crear(): ArbolB;
PROCEDURE EsVacio(a: ArbolB): BOOLEAN;
PROCEDURE ArbolBinario(i: ITEM; l, r: ArbolB): ArbolB;
PROCEDURE Raiz(a: ArbolB): ITEM;
PROCEDURE HijoIzq(a: ArbolB): ArbolB;
PROCEDURE HijoDch(a: ArbolB): ArbolB;

END ArbolB.
```

4.6.1 Implementación dinámica.

Cada nodo en un árbol está compuesto por un dato del tipo ITEM, un puntero al subárbol izquierdo y otro al derecho. Esto sugiere la utilización de un registro con tres campos:

```
TYPE
        ARBOL = POINTER TO NODO;
        NODO =  RECORD
                    Valor: ITEM;
                    Izquierdo, Derecho: ARBOL
          END;
```

donde el tipo ITEM ha sido importado del módulo en que fue definido. Veamos algunos ejemplos de árboles (árbol no equilibrado en altura, y degenerado):



Una vez definido el tipo podemos implementar los procedimientos de acceso, asegurando que estos satisfagan las especificaciones algebraicas.

Con esta definición de tipo, el procedimiento Crear sólo tiene que devolver un puntero a NIL.

```
PROCEDURE Crear(): ArbolB;
(* Postcondición: devuelve un árbol vacío *)
BEGIN
        RETURN NIL
END Crear;
```

El procedimiento EsVacio se limitará, por tanto, a comprobar que el árbol a es un puntero nulo.

```
PROCEDURE EsVacio(a: ArbolB): BOOLEAN;
(* Postcondición: devuelve TRUE si a es vacío; en otro caso FALSE *)
BEGIN
        RETURN (a=NIL)
END EsVacio;
```

Es fácil comprobar que es TRUE la expresión booleana EsVacio(Crear()), mientras que EsVacio(ArbolBinario(i, l, r)) es FALSE, verificando los dos axiomas de su especificación.

El procedimiento ArbolBinario toma dos árboles y un valor, y produce un nuevo árbol insertando el valor en la raíz y usando estos árboles como sus subárboles derecho e izquierdo. Lo único que debe hacerse es crear un nuevo nodo para dicha raíz.

```
PROCEDURE ArbolBinario(i: ITEM; l, r: ArbolB): ArbolB;
(* Postcond.: devuelve un árbol binario que contiene i como valor de su raíz y l y r como subárboles izq.
y dch. respectivamente *)
VAR temp: ArbolB;
BEGIN
        NEW(temp);
        WITH temp^ DO
          valor := i;
          izquierdo := l;
          derecho := r
        END;
        RETURN temp
END ArbolBinario;
```

El número de celdas de memoria utilizadas se incrementa en uno con cada llamada a ArbolBinario.

El procedimiento Raiz devuelve el valor contenido en la raíz del árbol. Debemos asegurar que se detectan las situaciones especificadas en las precondiciones y, en caso de producirse, debe aplicarse el correspondiente tratamiento de errores; el mecanismo empleado en esta implementación consisten en la generación de un mensaje de error, y la terminación del programa. en caso de ser una situación correcta, simplemente se devuelve el contenido del campo valor del nodo primero, o raíz.

Para comprobar que se verifica el axioma Raiz(ArbolBinario(i,l,r))=i, vemos que ArbolBinario asigna i al campo valor del nodo situado en la raíz del árbol que construye, y Raiz devuelve el contenido del campo valor del árbol que le es pasado como argumento.

```
PROCEDURE Raiz(a: ArbolB): ITEM;
(* Postcond.: devuelve el contenido de la raíz del árbol binario a *)
BEGIN
        IF EsVacio(a) THEN
          WrStr("Error: árbol vacío");
          HALT
        ELSE
          RETURN a^.valor
        END
END Raiz;
```

La función selectora HijoIzq devuelve el subárbol izquierdo del árbol que le es pasado, dejando el árbol que recibe sin cambio. De los axiomas para el procedimiento sabemos que un intento de obtener el subárbol izquierdo de un árbol vacío debe resultar en una situación de error, y, como es usual, es lo primero que se comprueba. En otro caso, se devuelve el puntero almacenado en el campo izquierdo del árbol.

```
PROCEDURE HijoIzq(a: ArbolB): ArbolB;
(* Postcondición: devuelve el subárbol izquierdo de a *)
BEGIN
        IF EsVacio(a) THEN
          WrStr("Error: árbol vacío");
          HALT
        ELSE
          RETURN a^.izquierdo
        END
END HijoIzq;
```

HijoIzq devuelve el campo izquierdo del nodo situado en la raíz del árbol, luego el resultado de HijoIzq(ArbolBinario(i, l, r) es el árbol l, lo que verifica el axioma correspondiente.

El procedimiento HijoDch tiene un comportamiento idéntico al anterior, con la única diferencia de que actúa con el subárbol derecho en lugar de con el izquierdo.

```
PROCEDURE HijoDch(a: ArbolB): ArbolB;
(* Postcondición: devuelve el subárbol derecho de a *)
BEGIN
        IF EsVacio(a) THEN
          WrStr("Error: árbol vacío");
          HALT
        ELSE
          RETURN a^.derecho
        END;
END HijoDch;
```

-----·o·-----

Con esta implementación la cantidad de espacio utilizado es proporcional al número de operaciones ArbolBinario realizadas, ya que es el único procedimiento que hace una llamada a NEW.

Desventajas (las típicas de una implementación dinámica):
- La memoria necesaria se toma en tiempo de ejecución.
- Todos los accesos a la información deben hacerse de forma indirecta.

A continuación se adjunta la representación completa, empleando un tratamiento de errores mediante variable global.

```
IMPLEMENTATION MODULE ArbolB;
```

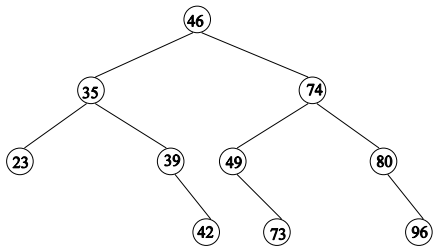
```
FROM IO IMPORT WrStr;
FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available;

TYPE
        ArbolB = POINTER TO NODO;
        NODO =  RECORD
                        valor: ITEM;
                        izquierdo, derecho: ArbolB
                END;

PROCEDURE Crear(): ArbolB;
(* Postcondición: devuelve un árbol vacío *)
BEGIN
        error := SinError;
        RETURN NIL;
END Crear;

PROCEDURE EsVacio(a: ArbolB): BOOLEAN;
(* Postcondición: devuelve TRUE si a es vacío; en otro caso FALSE *)
BEGIN
        error := SinError;
        RETURN (a=NIL);
END EsVacio;

PROCEDURE ArbolBinario(i: ITEM; l, r: ArbolB): ArbolB;
(*      Postcondición: devuelve un árbol binario que contiene i como valor de su
        raíz y l y r como subárboles izquierdo y derecho respectivamente *)
VAR temp: ArbolB;
BEGIN
        IF NOT Available(SIZE(NODO)) THEN error := SinMemoria; RETURN NIL; END;
        error := SinError;
        NEW(temp);
        WITH temp^ DO
          valor := i;
          izquierdo := l;
          derecho := r
        END;
        RETURN temp
END ArbolBinario;

PROCEDURE Raiz(a: ArbolB): ITEM;
(* Postcondición: devuelve el contenido de la raíz del árbol binario a *)
VAR
        basura : ITEM;
BEGIN
        IF EsVacio(a) THEN error := ArbolVacio; RETURN basura; END;
        error := SinError;
        RETURN a^.valor;
END Raiz;

PROCEDURE HijoIzq(a: ArbolB): ArbolB;
(* Postcondición: devuelve el subárbol izquierdo de a *)
BEGIN
        IF EsVacio(a) THEN error := ArbolVacio; RETURN NIL; END;
        error := SinError;
```

```
        RETURN a^.izquierdo;
END HijoIzq;

PROCEDURE HijoDch(a: ArbolB): ArbolB;
(* Postcondición: devuelve el subárbol derecho de a *)
BEGIN
        IF EsVacio(a) THEN error := ArbolVacio; RETURN NIL; END;
        error := SinError;
        RETURN a^.derecho;
END HijoDch;

BEGIN
        error := SinError;
END ArbolB.
```

4.7 ÁRBOL BINARIO DE BÚSQUEDA.

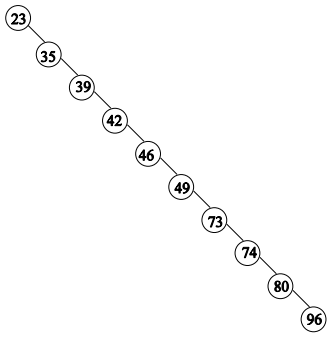
El árbol binario de búsqueda (ABB) toma su nombre del método de búsqueda dicotómico visto para listas ordenadas, que usaba un array para mantener un conjunto de items (número de items limitado, y operaciones de inserción y borrado de elementos costosas). Una implementación dinámica sería muy ineficiente. Otra solución consistiría en utilizar un árbol para mantener la información ordenada, con operaciones de inserción y extracción muy eficientes.

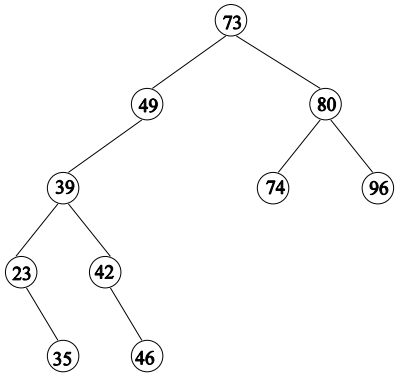
El TAD ABB es un árbol binario en el que los nodos están ordenados. El orden impuesto a los nodos es tal que para cada nodo, todos los valores de su subárbol izquierdo son menores que su valor, y todos los valores del derecho son mayores. Evidentemente, el tipo base requiere una relación de orden total. La inserción de los números 46, 35, 74, 39, 42, 49, 23, 73, 80 y 96 produce:



La localización de un valor en el árbol se realizará recorriendo el árbol, y tomando el árbol derecho o el izquierdo dependiendo de que el valor buscado sea mayor o menor que el de la raíz del árbol, respectívamente. Esto nos permite mantener la eficiencia y flexibilidad de una implementación dinámica con órdenes de complejidad para la búsqueda similares a los de la implementación estática con elementos ordenados. Sin embargo, no todos los árboles binarios mantienen esta eficiencia; dependiendo de la forma en que se realice la inserción de los elementos obtendremos representaciones muy dispares. Así, el árbol anterior también se podría haber representado como sigue:

Así podemos ver como una misma colección de elementos puede tener muchas representaciones posibles como árboles de búsqueda, con alturas muy diferentes. En el último gráfico





podemos observar como la rama izquierda es mucho mayor que la derecha. En el siguiente ejemplo, nos resulta un árbol degenerado incluso, y como cada nodo tiene vacío su subárbol izquierdo, resulta un árbol con altura igual a su número de nodos, siendo esencialmente una lista lineal.

En general, todo árbol binario con n nodos tiene una altura máxima de n y una mínima de $\lceil \lg_2(n+1) \rceil$. Esto es fácil de probar, bien mediante las propiedades triviales de los árboles o directamente por inducción sobre n.

Así pues debemos anticipar que el hecho de tener un ABB no es suficiente para asegurar un tiempo de búsqueda óptimo; el árbol debe tener además una altura mínima para asegurar que el camino más largo desde la raíz es tan corto como sea posible.

Teorema: El número de comparaciones necesarias para encontrar un valor particular en un árbol binario búsqueda balanceado en altura con n nodos es menor o igual que $\lceil \log_2(n+1) \rceil$.

Esto se deduce del hecho de que para árboles binarios balanceados, la altura es menor o igual a $\lceil \log_2(n+1) \rceil$, y puesto que siempre sabemos el camino a seguir, no necesitamos retroceder en la búsqueda.

4.7.1 Especificación algebraica.

La especificación algebraica de un ArbolBB es muy parecida a la de un árbol binario. De hecho, a pesar de tratarse de un TAD diferente, supondremos que las operaciones básicas de Crear, Arbol_binario, Raíz, Hijo_izq e Hijo_dch se definen de la misma manera, y por tanto las omitiremos. Incluso podemos ver un TAD ArbolBB como un ArbolB que cumple ciertas propiedades. Con este objetivo, podemos crear un predicado Es_ArbolBB que indica cuando un ArbolB cumple la propiedad de ser binario de búsqueda.

Es_ArbolBB : ArbolB ⟶ Lógico

ecuaciones

Es_ArbolBB(Crear) == V

Es_ArbolBB(Arbol_binario(r, i, d)) == ((not (Es_Vacio(i))) -> (r > Máximo(i))) and
((not (Es_Vacio(d))) -> (r < Mínimo(d))) and
Es_ArbolBB(i) and
Es_ArbolBB(d)

donde Máximo y Mínimo se deben especificar de la siguiente forma:

Mínimo, Máximo : ArbolB $\not\longrightarrow$ Elemento

precondiciones a : ArbolB

Máximo(a) : not Es_Vacío(a)

Mínimo(a) : not Es_Vacío(a)

ecuaciones r : Elemento i, d : ArbolB

```
Mínimo(Arbol_binario(r, i, d)) ==   SI Es_Vacío(i) and Es_Vacio(d) ENTONCES
                                        r
                                    SI NO SI Es_Vacio(i) ENTONCES
                                        MIN(r, Mínimo(d))
                                    SI NO SI Es_Vacio(d) ENTONCES
                                        MIN(r, Mínimo(i))
                                    SI NO
                                        MIN(r, MIN(Mínimo(i), Mínimo(d)))
Máximo(Arbol_binario(r, i, d)) ==   SI Es_Vacío(i) and Es_Vacio(d) ENTONCES
                                        r
                                    SI NO SI Es_Vacio(i) ENTONCES
                                        MAX(r, Máximo(d))
                                    SI NO SI Es_Vacio(d) ENTONCES
                                        MAX(r, Máximo(i))
                                    SI NO
                                        MAX(r, MAX(Máximo(i), Máximo(d)))
```

Sin embargo, nosotros supondremos que un ArbolBB es un TAD diferente al ArbolB normal visto en el punto anterior; por tanto no necesitaremos más este predicado. La principal diferencia es que el procedimiento Arbol_binario no es visible al usuario del TAD ya que su uso podría dar lugar a árboles no ordenados; en su lugar necesitamos un procedimiento Insertar que, dado un elemento **e** del tipo Elemento y un ArbolBB, devuelva un ArbolBB con e insertado en la posición correcta. Los axiomas para describir este procedimiento serán:

Insertar : Elemento × ArbolBB $\longrightarrow$ ArbolBB

ecuaciones

Insertar(e, Crear) == Arbol_binario(e, Crear, Crear)

```
Insertar(e, Arbol_binario(r, i, d)) ==   SI e = r ENTONCES
                                             Arbol_binario(r, i, d)
                                         SI NO
                                             SI e < r ENTONCES
```

```
                                        Arbol_binario(r, Insertar(e, i), r)
                                SI NO
                                        Arbol_binario(r, i, Insertar(e, r))
```

Debido a la forma en que los elementos se sitúan en el árbol, el elemento más pequeño en un árbol no vacío es siempre el situado más a la izquierda, y el mayor el situado más a la derecha.

```
    Mínimo, Máximo : ArbolBB ⇸ Elemento
precondiciones      a : ArbolBB
    Máximo(a) : not Es_Vacío(a)
    Mínimo(a) : not Es_Vacío(a)
ecuaciones
    Mínimo(Arbol_binario(r, i, d)) ==   SI Es_Vacío(i) ENTONCES
                                            r
                                        SI NO
                                            Mínimo(i)
    Máximo(Arbol_binario(r, i, d)) ==   SI Es_Vacío(d) ENTONCES
                                            r
                                        SI NO
                                            Máximo(d)
```

Nótese la diferencia entre las operaciones Mínimo y Máximo cuando se refieren al tipo ArbolB o al tipo ArbolBB. En este último caso, debido a la propia constitución del árbol, no tenemos porqué buscar en todos los nodos, sino que basta con moverse a un lado o a otro del árbol, desechando en un solo paso aproximadamente la mitad de los nodos (dependiendo del balanceo del mismo).

El procedimiento Eliminar tiene como entrada un ArbolBB y un elemento y devuelve un ArbolBB sin dicho elemento. Si el elemento a eliminar se encuentra en la raíz hay dos casos básicos: si el árbol izquierdo es vacío se devuelve el árbol derecho e, igualmente, si el subárbol derecho es vacío se devuelve el izquierdo. Si ninguno de los subárboles es vacío, debemos tomar un elemento que ocupe la posición de la raíz, asegurando que el árbol sigue siendo un ArbolBB; hay dos candidatos para el reemplazo: el mayor del subárbol izquierdo y el menor del derecho, que es el sucesor de la raíz en inorden, y es el que decidimos usar arbitrariamente. Esto nos lleva a la siguiente especificación:

```
    Eliminar : Elemento × ArbolBB ⟶ ArbolBB
ecuaciones
    Eliminar(e, Crear) == Crear
    Eliminar(e, Arbol_binario(r, i, d)) ==
                SI e = r ENTONCES
                        SI Es_Vacio(d) ENTONCES
                                i
                        SI NO SI Es_Vacio(i) ENTONCES
                                d
                        SI NO
```

```
                    Arbol_binario(Mínimo(d), i, Eliminar(Mínimo(d), d))
            SI NO
                SI e < r ENTONCES
                    Arbol_binario(r, Eliminar(e, i), d)
                SI NO
                    Arbol_binario(r, i, Eliminar(e, d))
```

También tenemos la operación Está, que queda facilitada por el hecho de trabajar sobre un árbol ordenado:

Está : Elemento × ArbolBB ⟶ Lógico

ecuaciones

```
Está(e, Crear) == F
Está(e, Arbol_binario(r, i, d)) ==    SI e = r ENTONCES
                                          V
                                      SI NO SI e < r ENTONCES
                                          Está(e, i)
                                      SI NO
                                          Está(e, d)
```

Con esta especificación, lo que se asegura son los siguientes teoremas:

teoremas e : Elemento a : ArbolBB

Es_ArbolBB(a) → Es_ArbolBB(Insertar(e, a))

Es_ArbolBB(a) → Es_ArbolBB(Eliminar(e, a))

o sea, que las operaciones de Insertar y Eliminar mantienen las propiedades de un árbol binario de búsqueda.

tipo ABB;

dominios ArbolB, Elemento, Lógico

operaciones

Insertar: Elemento × ABB ⟶ ABB

Eliminar: Elemento × ABB ⟶ ABB

Está: Elemento × ABB ⟶ Lógico

Mínimo, Máximo: ABB ⟶/ Elemento

Es_ABB: ABB ⟶ Lógico

precondiciones i: Elemento a: ABB

pre Mínimo(a): not Es_Vacio(a)

pre Máximo(a): not Es_Vacio(a)

ecuaciones i, j: Elemento r, l: ABB

```
Insertar(i, Crear) == ArbolBinario(i, Crear, Crear)
Insertar(e, ArbolBinario(r, i, d)) ==    SI e=r ENTONCES
                                             ArbolBinario(e, i, d)
                                         SI NO
                                             SI e < r ENTONCES
```

```
                                                    ArbolBinario(r, Insertar(e, i), d)
                                              SI NO
                                                    ArbolBinario(r, i, Insertar(e, d))
Mínimo(ArbolBinario(r, i, d)) ==        SI EsVacio(i) ENTONCES
                                                    r
                                        SI NO
                                                    Mínimo(i)
Máximo(ArbolBinario(r, i, d)) ==        SI EsVacio(d) ENTONCES
                                                    r
                                        SI NO
                                                    Máximo(d)
Eliminar(e, Crear) == Crear
Eliminar(e, ArbolBinario(r, i, d)) ==
                    SI e = r ENTONCES
                              SI EsVacio(d) ENTONCES
                                        i
                              SI NO SI EsVacio(i) ENTONCES
                                        d
                              SI NO
                                        ArbolBinario(Mínimo(d), i, Eliminar(Mínimo(d), d))
                    SI NO SI e < r ENTONCES
                              ArbolBinario(r, Eliminar(e, i), d)
                    SI NO
                              ArbolBinario(r, i, Eliminar(e, d));
Es_ABB(Crear) == V
Es_ABB(ArbolBinario(r, i, d)) ==        ((not (EsVacio(i))) → (r > Máximo(i))) and
                                        ((not (EsVacio(d))) → (r < Mínimo(d))) and
                                        Es_ABB(i) and
                                        Es_ABB(d)
Está(e, Crear) == F
Está(e ArbolBinario(r, i, d)) ==        SI e = r ENTONCES
                                                    V
                                        SI NO SI e < r ENTONCES
                                                    Está(e, i)
                                        SI NO
                                                    Está(e, d)
```

fin

4.7.2 Implementaciones del árbol binario de búsqueda.

A cotniuación vemos la impleemntación dinámica correspondiente a esta especificación. Suponemos que los elementos de tipo ITEM pueden ser comparados mediante los operadores relacionales típicos. Si no es este el caso, sería necesario incluir en los procedimientos Insertar(), Eliminar y Esta(), un parámetro adicional que permita saber cuando un ITEM es mayor, igual, o menor que otro.

```
DEFINITION MODULE ArbolBB;

FROM ArbolB IMPORT ITEM;

TYPE ABB;

PROCEDURE Crear(): ABB;
PROCEDURE Insertar(i: ITEM; A: ABB): ABB;
PROCEDURE Eliminar(i: ITEM; A: ABB): ABB;
PROCEDURE Esta(i: ITEM; A: ABB): BOOLEAN;

END ArbolBB.


IMPLEMENTATION MODULE ArbolBB;

FROM IO IMPORT WrStr;
FROM ArbolB IMPORT ArbolB, ArbolBinario, Raiz, HijoIzq, HijoDch, EsVacio;
IMPORT ArbolB;

TYPE ABB = ArbolB;

PROCEDURE Crear(): ABB;
BEGIN
        RETURN ArbolB.Crear()
END Crear;

PROCEDURE Insertar(i: ITEM; A: ABB): ABB;
BEGIN
        IF EsVacio(A) THEN
          RETURN ArbolBinario(i, ArbolB.Crear(), ArbolB.Crear())
        ELSIF i = Raiz(A) THEN
          RETURN A
        ELSIF i < Raiz(A) THEN
          RETURN ArbolBinario(Raiz(A), Insertar(i, HijoIzq(A)), HijoDch(A))
        ELSE
          RETURN ArbolBinario(Raiz(A), HijoIzq(A), Insertar(i, HijoDch(A)))
        END
END Insertar;

PROCEDURE Minimo(A: ABB): ITEM;
BEGIN
        IF EsVacio(A) THEN
          WrStr("Error: Árbol Nulo");
          HALT
        ELSIF EsVacio(HijoIzq(A)) THEN
          RETURN Raiz(A)
        ELSE RETURN Minimo(HijoIzq(A))
        END
END Minimo;


PROCEDURE Eliminar(i: ITEM; A: ABB): ABB;
BEGIN
        IF EsVacio(A) THEN
```

```
        RETURN ArbolB.Crear()
      ELSIF i = Raiz(A) THEN
        IF EsVacio(HijoIzq(A)) THEN
              RETURN HijoDch(A)
        ELSIF EsVacio(HijoDch(A)) THEN
              RETURN HijoIzq(A)
        ELSE
              RETURN
                    ArbolBinario(
                          Minimo(HijoDch(A)),
                          HijoIzq(A),
                          Eliminar(Minimo(HijoDch(A)), HijoDch(A)))
        END
      ELSIF i < Raiz(A) THEN
        RETURN ArbolBinario(Raiz(A), Eliminar(i, HijoIzq(A)), HijoDch(A))
      ELSE
        RETURN ArbolBinario(Raiz(A), HijoIzq(A), Eliminar(i, HijoDch(A)))
      END
END Eliminar;

PROCEDURE Esta(i: ITEM; A: ABB): BOOLEAN;
BEGIN
      RETURN
        (NOT EsVacio(A))
        AND
        ((i = Raiz(A)) OR Esta(i, HijoIzq(A)) OR Esta(i, HijoDch(A)))
END Esta;

END ArbolBB.
```

A continuación se presenta el A.B.B. implementado de forma directa (sin hacer uso para nada de un árbol binario simple definido con anterioridad). En este caso, el ITEM se halla formado por dos elementos: uno que contiene la clave por la que quiere mantener ordenados los ítemes, y otro la información asociada a cada una. El efecto es el mismo.

Esta implementación puede dar una idea de la enorme similitud entre un árbol binario y una tabla de dispersión. Se empleará uno u otro dependiendo de la velocidad de acceso que se pretenda:

- Si deseamos aumentar la eficiencia en los accesos, aun a costa de desperdiciar memoria, usaremos las tablas de dispersión, que permiten accesos de $\approx O(1)$.
- Si el espacio de almacenamiento es crítico, usaremos un árbol binario de búsqueda.
- Si deseamos establecer una cota al tiempo máximo de acceso a un dato, emplearemos construccines especiales como:
  - Árboles binarios de búsqueda AVL.
  - Estrategias especiales de redispersión.

```
DEFINITION MODULE ABB;

      TYPE ABB;
        CLAVE = CARDINAL;
        INFO = ARRAY [0..100] OF CHAR;

      PROCEDURE crear ():ABB;
```

```
        PROCEDURE vacio (B:ABB): BOOLEAN;
        PROCEDURE esta (K:CLAVE;B:ABB): BOOLEAN;
        PROCEDURE contenido (K:CLAVE;B:ABB): INFO;
        PROCEDURE insertar (K:CLAVE;X:INFO;VAR B:ABB);
        PROCEDURE eliminar (K:CLAVE;VAR B:ABB);
        PROCEDURE inorden (B:ABB);

END ABB.


IMPLEMENTATION MODULE ABB;

FROM IO IMPORT WrStr,WrLn,WrCard;
FROM Storage IMPORT ALLOCATE,DEALLOCATE;

TYPE ABB= POINTER TO NODO;
        NODO=  RECORD
                        clave:CLAVE;
                        info: INFO;
                        izq,dch: ABB
                END;

        PROCEDURE crear ():ABB;
          BEGIN
                RETURN NIL
          END crear;

        PROCEDURE vacio (B:ABB): BOOLEAN;
          BEGIN
                RETURN (B=NIL)
          END vacio;

        PROCEDURE esta (K:CLAVE;B:ABB): BOOLEAN;
          BEGIN
                IF B=NIL THEN RETURN FALSE
                ELSIF K=B^.clave THEN RETURN TRUE
                ELSIF K<B^.clave THEN RETURN esta(K,B^.izq)
                ELSE RETURN esta(K,B^.dch)
                END
          END esta;


        PROCEDURE contenido (K:CLAVE;B:ABB): INFO;
          BEGIN
                IF B=NIL THEN WrStr('ERROR: En contenido. Arbol vacío');HALT
                ELSIF K=B^.clave THEN RETURN B^.info
                ELSIF K<B^.clave THEN RETURN contenido (K,B^.izq)
                ELSE RETURN contenido (K,B^.dch)
                END
          END contenido;

        PROCEDURE nuevo_abb (K:CLAVE;X:INFO):ABB;
          VAR aux:ABB;
          BEGIN
                NEW(aux);
```

```
            aux^.clave:=K;
            aux^.info:=X;
            aux^.izq:=NIL;
            aux^.dch:=NIL;
            RETURN aux
    END nuevo_abb;

  PROCEDURE insertar (K:CLAVE;X:INFO;VAR B:ABB);
    BEGIN
            IF vacio(B) THEN B:=nuevo_abb(K,X)
            ELSIF K=B^.clave THEN B^.info:=X
            ELSIF K<B^.clave THEN insertar (K,X,B^.izq)
            ELSE insertar (K,X,B^.dch)
            END
    END insertar;

  PROCEDURE eliminar (K:CLAVE;VAR B:ABB);
    VAR aux: ABB;
    BEGIN
            IF NOT vacio(B) THEN
                    IF K=B^.clave THEN
                            IF B^.izq=NIL THEN
                                    aux:=B;
                                    B:=B^.dch;
                                    DISPOSE (aux)
                            ELSIF B^.dch=NIL THEN
                                    aux:=B;
                                    B:=B^.izq;
                                    DISPOSE (aux)
                            ELSE (* los dos árboles son no vacíos *)
                                    aux:=B^.dch;
                                    WHILE aux^.izq<>NIL DO
                                            aux:=aux^.izq
                                    END;
                                    B^.clave:=aux^.clave;
                                    B^.info:=aux^.info;
                                    eliminar (aux^.clave,B^.dch)
                            END
                    ELSIF K<B^.clave THEN eliminar (K,B^.izq)
                    ELSE eliminar (K,B^.dch)
                    END
            END
    END eliminar;

  PROCEDURE inorden (B:ABB);
    BEGIN
            IF NOT vacio(B)
                    THEN inorden (B^.izq);
                            WrCard (B^.clave,1); WrStr (' : ');
                            WrStr(B^.info);WrLn;
                            inorden (B^.dch)
            END
    END inorden;

END ABB.
```

Ahora veremos la enorme complejidad derivada de la implementación de los árboles enhebrados. al final veremos la casuística del procedimiento más complejo: Eliminar. Aquí sólo se usa una campo de cometido, o lo que es lo mismo, sólo trabajaremos con una hebra: la derecha, que apuntará al nodo sucesor en inorden. Como ITEM emplearemos tan sólo un valor de tipo CARDINAL, para no enrarecer el código innecesariamente.

```
DEFINITION MODULE Enhebrado;

TYPE
        ABB;
        ITEM = CARDINAL;

PROCEDURE Crear(): ABB;
PROCEDURE Insertar(i: ITEM; VAR A: ABB);
PROCEDURE Eliminar(i: ITEM; VAR A: ABB);
PROCEDURE Esta(i: ITEM; A: ABB): BOOLEAN;

END Enhebrado.


IMPLEMENTATION MODULE Enhebrado;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM IO IMPORT WrStr;

TYPE ABB = POINTER TO NODO;
          NODO =          RECORD
                                  info: ITEM;
                                  izq, dch: ABB;
                                  hebra: BOOLEAN
                          END;

PROCEDURE NuevoNodo(i: ITEM): ABB;
VAR nodoAux: ABB;
BEGIN
        NEW(nodoAux);
        WITH nodoAux^ DO
          info := i;
          izq := NIL;
          dch := NIL;
          hebra := TRUE;
        END;
        RETURN nodoAux
END NuevoNodo;

PROCEDURE crear ():ABB;
BEGIN
        RETURN NIL
END crear;

PROCEDURE Insertar(i: ITEM; VAR A: ABB);
VAR conexion: ABB;
BEGIN
```

```
        IF A = NIL THEN
          A := NuevoNodo(i)
        ELSIF i = A^.info THEN A^.info := i
        ELSIF i < A^.info THEN
          IF A^.izq = NIL THEN
                A^.izq := NuevoNodo(i);
                A^.izq^.dch := A
          ELSE
                Insertar(i, A^.izq)
          END
        ELSE (* i > A^.info *)
          IF A^.hebra THEN
                conexion := A^.dch;
                A^.dch := NuevoNodo(i);
                A^.hebra := FALSE;
                A^.dch^.dch := conexion
          ELSE
                Insertar(i, A^.dch)
          END
        END
END Insertar;


PROCEDURE Eliminar(i: ITEM; VAR A: ABB);
VAR aux, ant, act: ABB;
BEGIN
        IF A <> NIL THEN ant := A; act := A;
          WHILE ((i<act^.info) AND (act^.izq<>NIL))        (* búsqueda *)
                        OR ((i>act^.info) AND (NOT act^.hebra)) DO
                ant := act;
                IF i<act^.info THEN act := act^.izq
                ELSE act := act^.dch
                END
          END;
          IF act^.info = i THEN                                (* nodo localizado *)
                IF act^.izq=NIL THEN                        (*hijo izquierdo nulo *)
                        IF act^.hebra THEN
                                IF act=ant THEN A := NIL
                                ELSIF act=ant^.izq THEN ant^.izq := NIL
                                ELSE                                (* act = ant^.dch *)
                                        ant^.hebra := TRUE; ant^.dch := act^.dch
                                END
                        ELSE                                        (* NOT A^.hebra *)
                                IF act=ant THEN A := act^.dch
                                ELSIF act=ant^.izq THEN ant^.izq := act^.dch
                                ELSE (* act = ant^.dch *) ant^.dch := act^.dch
                                END
                        END;
                        DISPOSE(act)
                ELSIF act^.hebra THEN
                        IF act=ant THEN A := act^.izq
                        ELSIF act = ant^.izq THEN ant^.izq := act^.izq
                        ELSE ant^.dch := act^.izq                (* act = ant^.dch *)
                        END;
                        aux := act^.izq;
```

```
                        WHILE NOT aux^.hebra DO aux := aux^.dch END;
                        aux^.dch := act^.dch;
                        DISPOSE(act)
                ELSE (* búsqueda del mínimo en el árbol derecho *)
                        aux := act^.dch; ant := act;
                        WHILE aux^.izq <> NIL DO ant := aux; aux := aux^.izq END;
                        act^.info := aux^.info;
                        IF ant=act THEN
                                ant^.dch := aux^.dch;
                                IF aux^.hebra ant^.hebra := TRUE END;
                        ELSIF aux^.hebra THEN ant^.izq := NIL
                        ELSE ant^.izq := aux^.dch
                        END;
                        DISPOSE(aux)
                END
          END
        END
END Eliminar;


PROCEDURE Primero(A: ABB): ABB;
BEGIN
        IF A=NIL THEN
          WrStr("ERROR:  Árbol nulo");
          HALT
        ELSE
          WHILE A^.izq<>NIL DO
                A := A^.izq
          END;
          RETURN A
        END
END Primero;

PROCEDURE Siguiente(A: ABB): ABB;
BEGIN
        IF A=NIL THEN
          WrStr("ERROR:  rbol nulo");
          HALT
        ELSIF A^.hebra THEN
          RETURN A^.dch
        ELSE
          RETURN Primero(A^.dch)
        END
END Siguiente;

PROCEDURE InOrden(A: ABB): COLA;
VAR recorrido: LISTA; i: CARDINAL;
BEGIN
        recorrido := ListaNula();
        i := 1;
        IF A<>NIL THEN
          A := Primero(A);
          REPEAT
                Insertar(A^.info, i, recorrido);
                A := Siguiente(A);
```

```
                INC(i)
        UNTIL A=NIL;
    END;
    RETURN recorrido
END InOrden;

END Enhebrado.
```

EXPLICACIÓN DE Eliminar.

En primer lugar, sólo se actúa en caso de que el árbol no sea vacío. A continuación se busca el valor a eliminar. si no se encuentra, tampoco se hace nada. Si se encuentra, se pueden plantear los casos siguientes:

a) El subárbol izq. del nodo a borrar es vacío:

a.1) El nodo a borrar tiene una hebra a un sucesor.

a.1.α) El nodo a borrar es raíz.

Se retorna el árbol vacío.

a.1.β) El nodo a borrar está a la izq. del anterior.

El subárbol izq. del nodo anterior se pone a vacío.

a.1.γ) El nodo a borrar está a la derecha del anterior.

Se hace que la hebra del anterior, apunte a donde apunta la hebra del que se va a borrar.

a.2) El nodo a borrar no tiene hebra, o sea, sólo tiene árbol derecho.

a.2.α) El nodo a borrar es raíz.

Se retorna su subárbol dcho. tal cual.

a.2.β) El nodo a borrar está a la izq. del anterior.

El subárbol izq. del anterior, apunta al derecho del que se borra.

a.2.γ) El nodo a borrar está a la derecha del anterior.

El subárbol dch. del anterior apunta al dcho. del que se borra.

b) El subárbol izq. del nodo a borrar no es vacío, pero sí el derecho, y por tanto tiene una hebra.

b.1) El nodo a borrar es el raíz.

El árbol ppal. es el subárbol izq. del que se va a borrar, *****.

b.2) El nodo a borrar está a la izq. del anterior.

El subárbol izq. del anterior apunta al izq. del que se va a borrar, *****.

b.3) El nodo a borrar está a la dch. del anterior.

El subárbol dch. del anterior apunta al izq. del que se va a borrar, *****.

***** La hebra del nodo más grande del subárbol izq. del que se va a borrar, deberá apuntar a donde apuntaba la hebra del que se va a borrar.

c) El nodo a borrar tiene subárbol izq. y dch..

***** Encontramos el menor del subárbol dch. del nodo a borrar. Pasamos su información al nodo que se iba a borrar, y ahora el nodo a borrar es este menor.

c.1) El nodo menor es justamente la raíz del subárbol dcho del que se iba a borrar.

El subárbol dch. del nodo que se iba a borrar, apunta al subárbol dch. del nodo menor (el que se va a borrar), tanto si era una hebra como si no.

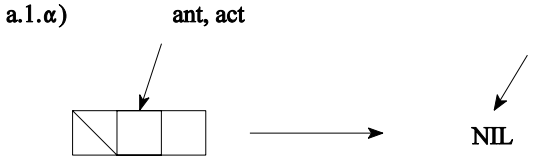
c.2) Si el elemento menor tiene una hebra (el subárbol dch. apunta a su sucesor en inorden), o sea, es un nodo hoja:

El subárbol izq. del anterior al que se va a borrar, apunta a NIL.

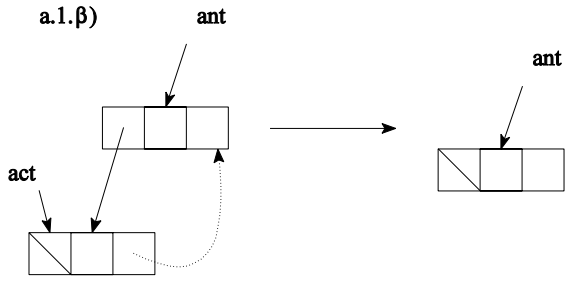
c.3) Si el subárbol dch. del elemento menor (el que se va a borrar), no es vacío (nótese que el izq. sí lo es, porque por ello es el elemento menor).

El subárbol izq. del anterior al menor, apunta al dch. del menor.

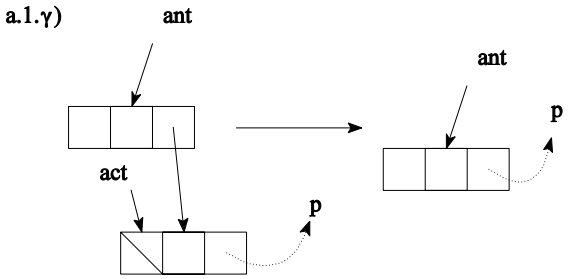
Los siguientes gráficos aclararán un poco más las cosas:



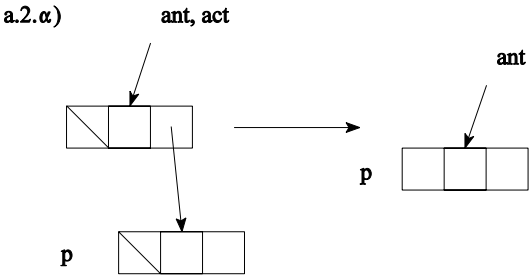
a.1.β)



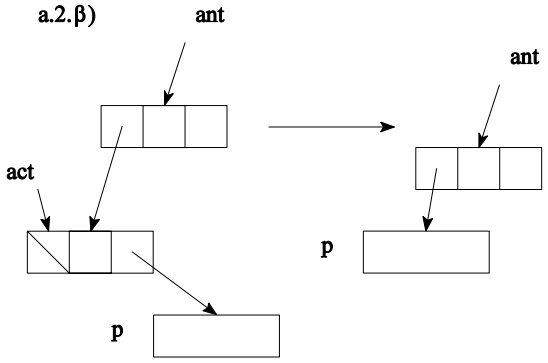
a.1.γ)



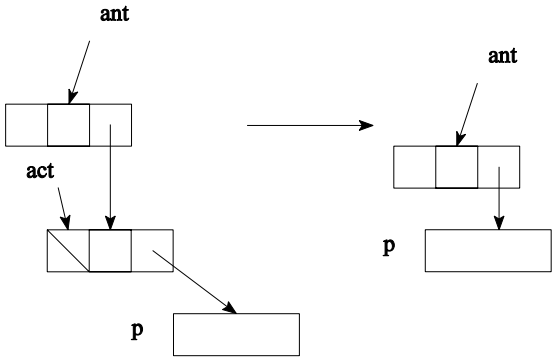
a.2.α)

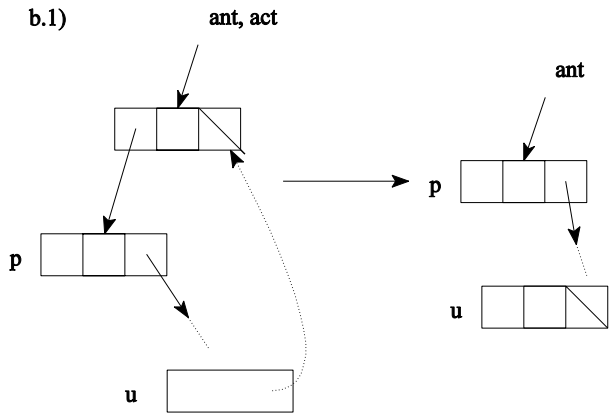


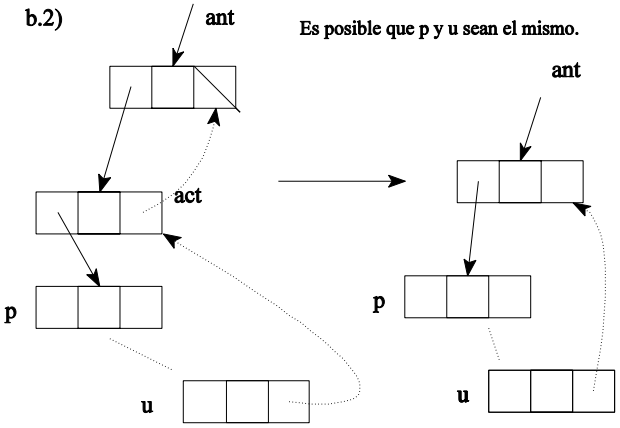
a.2.β)

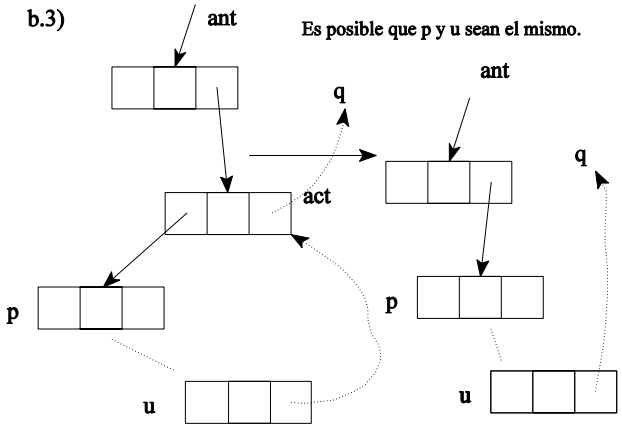


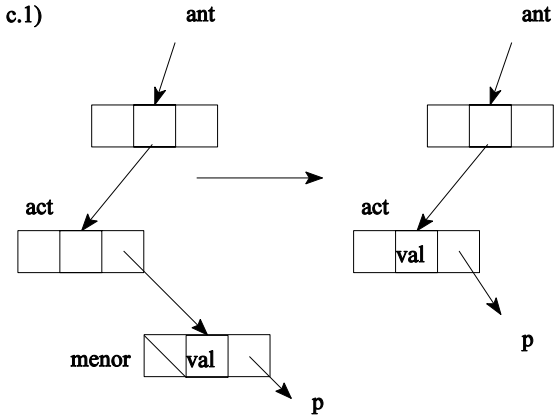
a.2.γ)

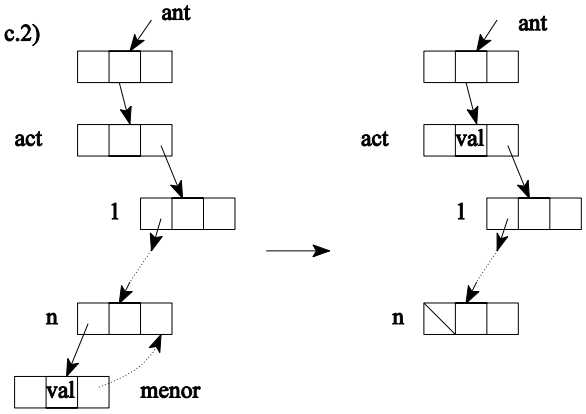


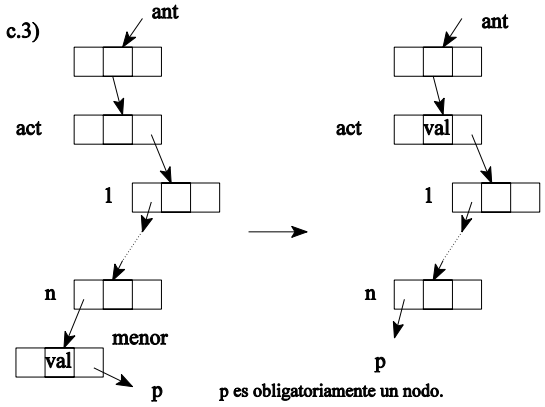

b.1)
ant, act
p
u
ant
p
u


b.2)
ant
Es posible que p y u sean el mismo.
act
p
u
ant
p
u


b.3)
ant
Es posible que p y u sean el mismo.
q
act
p
u
ant
q
p
u


c.1)
ant
act
menor
val
p
ant
act
val
p


c.2)
ant
act
1
n
val
menor
ant
act
val
1
n


c.3)
ant
act
1
n
menor
val
p
p es obligatoriamente un nodo.
ant
act
val
1
n
p

EJERCICIOS.

1.- Suponiendo que la operación Reduce utiliza funciones del tipo:
FUNCION_4 = PROCEDURE(ITEM, ITEM, CARDINAL, CARDINAL) : ITEM;
donde los parámetros son:
- Elemento reducido hasta el momento.
- Nuevo valor a reducir.
- Posición del nuevo valor a reducir.
- Nº total de valores a reducir.

Implementar la función de tipo FUNCION_4 que calculase la Moda.

2.- Mediante la operación Reduce, implementar una función para devolver la longitud de una lista.

3.- Hacer la especificación de una operación que devuelva una lista con las hojas de un árbol binario.

4.- Hacer la especificación de una operación que devuelva el nº de nodos externos de un árbol binario.

5.- Hacer la especificación algebraica de la función NIVEL_LLENO : ArbolB $\longrightarrow \mathbb{N}$, y que devuelve el número más grande del nivel Lleno en el árbol que se le pasa como parámetro.

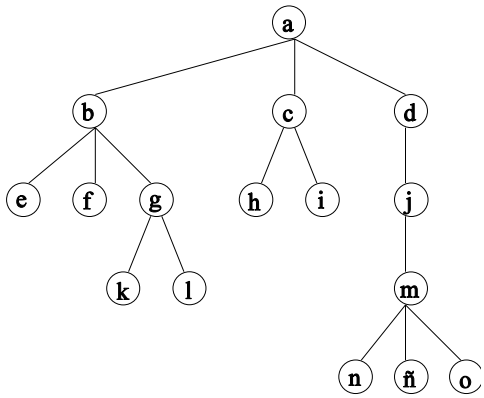
6.- Hacer la especificación algebraica de la función MAX_NODO : ArbolB $\not\longrightarrow$ Elemento, que, suponiendo que el tipo base Elemento posee una relación de orden total, devuelva el mayor valor de los almacenados en el árbol.

7.- Repetir la especificación anterior, pero en lugar de devolver el mayor valor, devolver el subárbol que tiene como raíz al nodo con dicho mayor valor.

8.- Dado un árbol binario de búsqueda, hacer un función en Modula-2 que devuelva la diferencia entre el mayor y el menor de los valores almacenados en dicho árbol. Hacerlo de forma no recursiva.

9.- Escribir una función Modula-2 para la poda de árboles, de manera que dado un árbol A, se produce un árbol A', tal que: A' es una copia de A, excepto por el hecho de que para cada subárbol S de A, si Es_Vacío(Hijo_izq(S)) ó Es_Vacío(Hijo_dch(S)), entonces S se sustituye por la hoja Arbol_binario(Raíz(S), Crear, Crear).

10.- Indicar los recorridos en preorden, inorden y postorden del siguiente árbol general:



11.- Escribir un algoritmo de recorrido en inorden de un árbol binario que representa una expresión, de manera que se visualice dicha expresión con los correspondientes paréntesis.

12.- Dibujar el árbol binario que representa las siguientes expresiones:
a) a * (b + c / ((x - 1) * (y + 4))) + X * x - 24
b) $a_0 + x * (a_1 + x * (a_2 + x * a_3))$

13.- Escribir un procedimiento en Modula-2 al que se le pasa un árbol binario representado por una estructura dinámica con punteros simples a los hijos, y devuelve el mismo árbol pero representado por una estructura estática compacta en preorden, tal y como se vió en el punto 4.5.2.2.

14.- Reconstruir un árbol binario a partir de los recorridos siguientes:
a) Preorden: 2, 5, 3, 9, 7, 1, 6, 4, 8.
Inorden: 9, 3, 7, 5, 1, 2, 6, 8, 4.
b) Inorden: 5, 6, 12, 10, 1, 9, 13, 4, 8, 2, 7, 3, 11.
Postorden: 6, 5, 10, 9, 1, 13, 12, 2, 8, 3, 11, 7, 4.

15.- Dibujar todos los posibles árboles de búsqueda que contengan sólo los cuatro números siguientes: 5, 10, 15 y 20. En general, ¿Cuántos árboles de búsqueda se pueden formar con **n** elementos diferentes?.

16.- Escribir una función en Modula-2, a la que se le pasa un nodo de un árbol enhebrado, y devuelve su padre. Hacerlo de forma no recursiva.

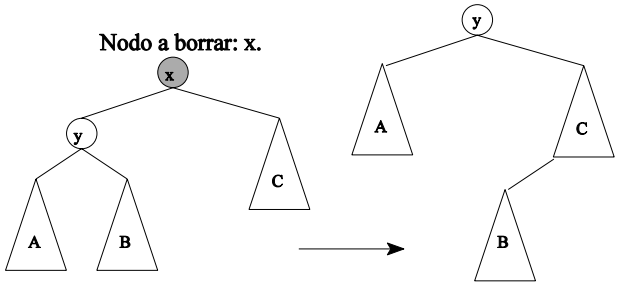
17.- Construir el árbol binario de búsqueda que se produce como resultado de introducir los siguientes elementos, según la secuencia dada:
Rafael, Antonio, Juan, Carlos, Alejandro, Miguel, Angel, José.

18.- ***Problema de la concordancia.*** Escribir un programa que acepte una lista de palabras, y devuelva dicha lista ordenada alfabéticamente, y junto con cada elemento, un contador que

especifique el número de veces que esa palabra se repite.

19.- Si en un ABB, almacenamos junto con cada nodo, el número de nodos de su hijo izq., y el número de nodos de su hijo dch., será posible encontrar el elemento k-ésimo en tiempo aproximadamente logarítmico. Implementar un ABB con esta cualidad.

20.- La operación de borrado de nodos en ABB se puede hacer moviendo subárboles enteros, de la forma:



Reescribir el procedimiento de borrado de nodos usando esta técnica.

21.- Implementar en Modula-2 un procedimiento al que se le pasan dos árboles binarios de búsqueda, y devuelve uno sólo, también binario de búsqueda, y que es el resultado de la mezcla de los dos anteriores. Hacer el ejercicio de dos formas:
a) El resultado debe ser equilibrado en altura, a ser posible balanceado.
b) Completo.
(Ver pistas al final, si se desea).

22.- Hacer la especificación de una operación que indique si un árbol es balanceado o no.

23.- Supongamos un árbol genealógico de nivel 5, en el que las hojas indican los ancestros, y la raíz, a un único hijo. Se pretende obtener la lista:tatarabuelos, bisabuelos, abuelos, padres, hijo. Evidentemente con esta estructura, sólo podemos especificar un hijo para cada par de padres. ¿Cómo sería un árbol tal?.

24.- Sea un tipo de árbol E tal que, o contiene un sólo nodo con un valor racional, o posee una raíz con la operación +, -, * ó / y obligatoriamente dos subárboles de tipo E. Véase ej. de la pág. 13. Especificar algebraicamente la operación Evalúa, a la que se le pasa un árbol E, y devuelve el resultado racional de evaluar la expresión aritmética que representa E.

25.- Una forma de representar árboles binarios, es mediante registros dinámicos con enlaces a los hijos ya al padre. Esta representación tiene la ventaja de que permite recorridos fáciles del árbol sin necesidad de recursividad. Implementar en Modula-2 un recorrido en postorden -sin

recursividad- de un árbol estructurado de esta forma.

26.- La introducción de nodos nuevos en un árbol enhebrado es mucho más fácil que su eliminación. Explicar como opera la operación Insertar nodo en un árbol tal, según la estructura vista en el punto 4.5.1.3.

27.- Supongamos que nos dan un array en Modula-2, de números a ordenar. Usando una representación estática de árboles binarios, y siguiendo el método heapsort, ordenar la lista. Los pasos son:

- Convertir la lista en un heap.
- Pasar el elemento superior a la última posición del array, y suponer que el array tiene una posición menos. El elemento de la posición última, pasa a la primera, siendo necesario reorganizar el heap.
- Hacer esto sucesivamente hasta que el árbol conste de un solo elemento.

28.- 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)))

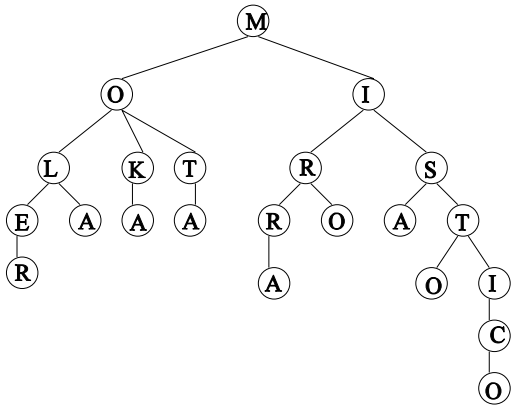
A_3=Arbol(d, Insertar(Arbol(g, Bosque_vacío), Insertar(Arbol(h, Bosque_vacío), Bosque_vacío)))

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)))

29.- 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.

30.- 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 lols 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:

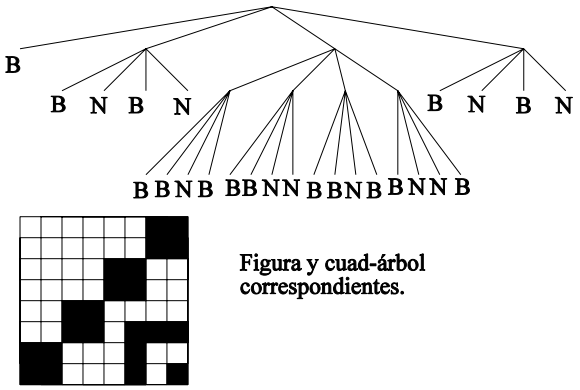


Figura y cuad-árbol correspondientes.

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.

Pista al ej. 21:
- Convertir los árboles a listas ordenadas.
- Mezclar las listas.
- Proceder a la construcción del árbol resultado.