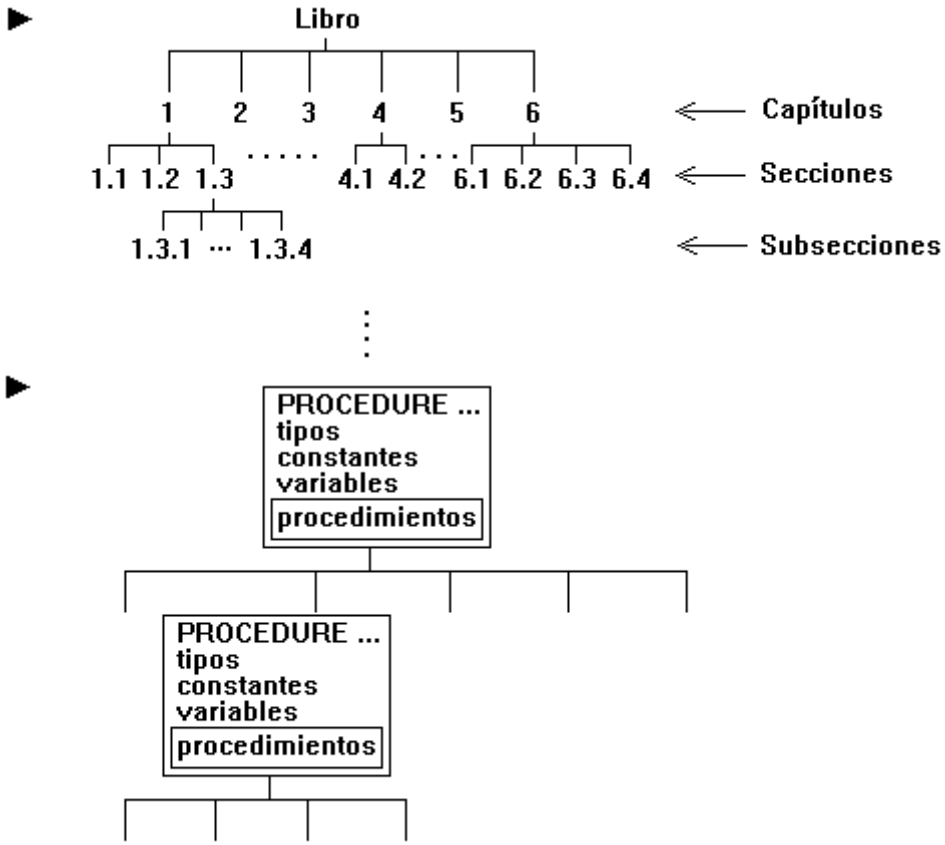
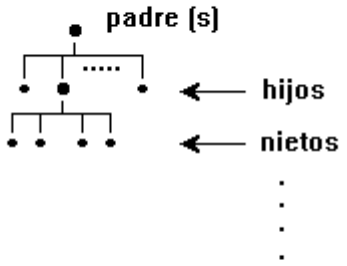
Árboles

Ejemplos de estructuras arborescentes: con forma de árbol



Regla de Alcance: los objetos visibles en un procedimiento son aquellos declarados en él mismo o en cualquier ancestro de él (cualquier procedimiento que se encuentre en el camino [único] desde él hasta el origen del árbol (=la raíz)).

- Arboles genealógicos



- Expresiones :
  Fórmulas

Normalmente se escriben en forma lineal:

x + 3 * y - 2 \ z

Pero para entender cómo son evaluadas se construye (implícitamente) un árbol:

- las reglas de precedencia aseguran la existencia de un único árbol para cada expresión en forma lineal;
- la forma de árbol representa directamente la estructura de la fórmula; pero es difícil de escribir (con la tecnología más comúnmente usada). La forma lineal es más fácil de escribir, pero más fácil de entender

forma lineal = sintaxis concreta
forma de árbol = sintaxis abstracta

Abstracción de la Noción de Árbol

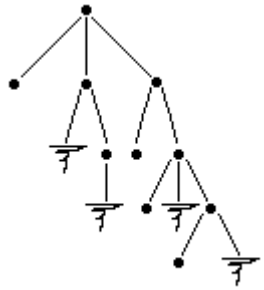
- Tratamos de definir: árbol de elementos de tipo α
- Un árbol de (elementos de tipo) α es:

-- o bien vacío
-- o bien consta de:

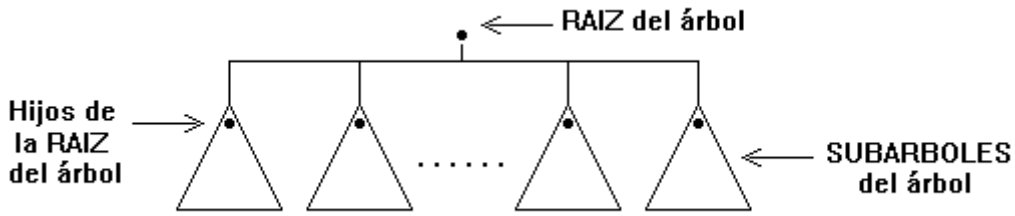
  - un elemento de tipo α
  - más un número de árboles de α

Ejemplo:

(árbol no vacío)

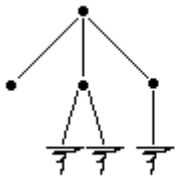


Algunas definiciones para árboles no vacíos:



- los elementos se llaman habitualmente NODOS del árbol
- los árboles también pueden definirse como grafos con ciertas propiedades especiales

Árboles n-arios y finitarios

- árbol n-ario: si no es vacío, tiene exactamente n subárboles n-arios
- árbol finitario: si no es vacío, tiene una cantidad finita de subárboles finitarios
- Ejemplo:
  - los árboles binarios son 2-arios (obvio)
  - 
    es finitario pero no n-ario para ningún n
  -
  - obviamente los árboles n-arios son todos finitarios

Pero: Todo árbol finitario puede representarse como un árbol binario, según método a ver más tarde.

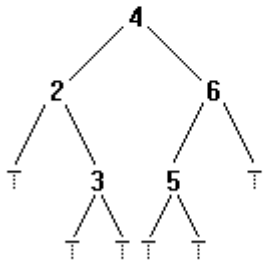
Árboles Binarios -- Definición Inductiva

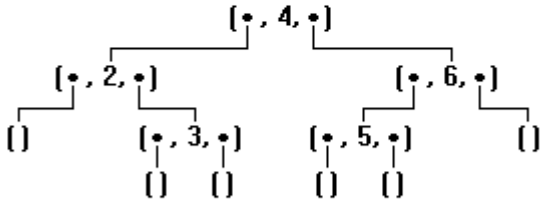
Definimos α AB (árboles binarios de α):

$$\frac{}{[\,]: \alpha\ AB}$$

$$\frac{a: \alpha \quad izq: \alpha\ AB \quad der: \alpha\ AB}{(izq, a, der): \alpha\ AB}$$

► Ej:





[[[ ], 2, [[ ], 3, [ ]]], 4, [[[ ], 5, [ ]], 6, [ ]]]

En la notación gráfica (bidimensional) podríamos definir:

$$\frac{}{\overline{\overline{\overline{}}}: \alpha\ AB}$$ (árbol binario vacío)

$$\frac{a: \alpha \quad izq: \alpha\ AB \quad der: \alpha\ AB}{\begin{array}{c} a \\ izq \quad der \end{array} : \alpha\ AB}$$ (árboles binarios no vacíos)

Funciones (recurrentes) sobre árboles binarios

- Recurrencia primitiva:

$$f : \alpha\ AB \rightarrow X$$

$$\begin{cases} f(()) = x_o \\ f((izq, a, der)) = c\ (a, f(izq), f(der)\ ) \end{cases}$$

- Recurrencia (más) general:

  En los casos con llamadas recurrentes:
  $f(t) = c\ (\ f(t_1), \ldots, f\ (t_n)\ )$
  alcanza que cada t_1 sea <u>más chico</u> que t (por ejemplo: en cantidad de nodos).

- Ejemplos:
  - Tamaño:

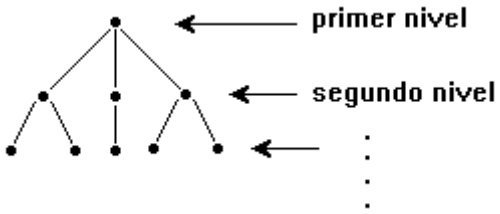
    Cantidad de nodos
    $|\ t\ |$: N para cada árbol binario t.

    $|\ -\ | : \alpha\ AB \rightarrow N$

    $|\ ()\ | : 0$

    $|\ (izq, a, der)\ | = 1 + |\ izq\ | + |\ der\ |$

  - Profundidad:
    - Antes: <u>niveles</u> de un



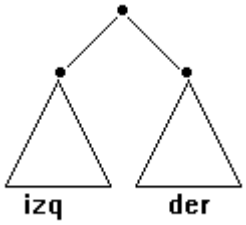
árbol

(Usando la analogía con árboles genealógicos, nivel = generación)

Profundidad de un árbol = cantidad de niveles que tiene
= cantidad de nodos en el camino más largo de la raíz a una hoja.

La profundidad del árbol binario vacío es 0.

La profundidad de un árbol de la forma:



es 1 + max (p_i, p_d) donde p_i es la profundidad de izq y p_d es la profundidad de der.

Obtenemos:

prof ( ) = 0
prof (izq, a, der) = 1 + max (prof (izq), prof (der))

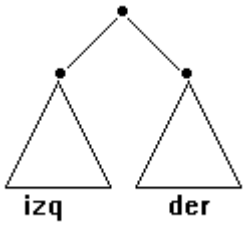
Máximo de un árbol binario de naturales

Max: Nat AB $\rightarrow$ Nat
(precondición: el argumento debe ser un árbol no vacío)
Max ((), n, ()) = n
Max ((), n, der) = max (n, Max (der))
Max (izq, n, ()) = max (n, Max (izq))
Max (izq, n, der) = max3 (n, Max (izq), Max (der))

(en las tres últimas ecuaciones suponemos izq y der no vacíos).

Recorridas de árboles binarios:

- En general, son procedimientos que visitan todos los nodos de un árbol binario efectuando cierta acción sobre cada uno de ellos.
- La forma de estos procedimientos depende del orden que se elija para visitar los nodos.
- Obviamente, recorrer un árbol vacío es trivial. Para recorrer un árbol no vacío:



hay tres órdenes naturales, según la raíz sea visitada antes que los subárboles, entre las recorridas de los subárboles o después de recorrer los subárboles.

Primer orden: Pre orden	se visita la raíz; se recorre el subárbol izquierdo;

	se recorre el subárbol derecho;
Segundo orden: Recorrida e orden Orden simétrico	se recorre el subárbol izquierdo; se visita la raíz; se recorre el subárbol derecho;
Tercer orden Post-orden	se recorre el subárbol izquierdo; se recorre el subárbol derecho; se visita la raíz;

- Ejemplo: formar la lista de los elementos de un árbol binario que se obtiene por recorrerlo en orden (linealización, secuenciamiento, listado):

en-orden ( ) = [ ]
en-orden (izq, a, der) = en-orden (izq)
++ (a. enorden (der))
(concatenación de listas)

= en-orden (izq)

++ [a] ++ enorden (der).

Árboles Binarios Ordenados (de búsqueda)

Consideremos árboles binarios de naturales
(más generalmente, de cualquier tipo sobre el cual pueda definirse un orden lineal).

Un árbol binario de naturales está ordenado si cada nodo es mayor que todo nodo de su subárbol izquierdo y menor que todo nodo de su subárbol derecho.

Ejemplo:

- Definición inductiva:

  un árbol binario ordenado (de naturales) es:

  - o bien vacío
  - o bien consta de un natural n y dos árboles binarios ordenados, izq y der, tales que todo elemento de izq es menor que n y todo elemento de der es mayor que n.

  (En esta definición no consideramos elementos repetidos - La generalización a este caso es obvia).

- Recordar búsqueda binaria en vectores ordenados, por bipartición

Los árboles binarios de búsqueda representan directamente la estructura de la búsqueda binaria:

?esMiembro: Nat x Nat AB → Bool

?esMiembro (n, ()) = false

?esMiembro (n, (izq, m, der)) =

case

n < m → ?esMiembro (n, izq)

| n = m → true

| n > m → ?es Miembro (n, der)

end

Otra propiedad interesante de los árboles binarios ordenados

-- su linealización en (segundo) orden es una lista ordenada.

-- lo cual da una idea para un algoritmo de ordenamiento de listas:

1. dada la lista a ser ordenada, formar con sus elementos un árbol binario ordenado.
2. luego, listar este árbol en orden.

(TreeSort)

-- Para resolver el primero de estos dos pasos, debe escribirse un algoritmo de inserción de un elemento en un árbol binario ordenado, de modo que el orden sea respetado.

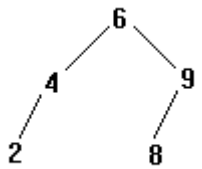
Representación de Árboles Binarios en C/C++

Caben las mismas consideraciones que para las listas:

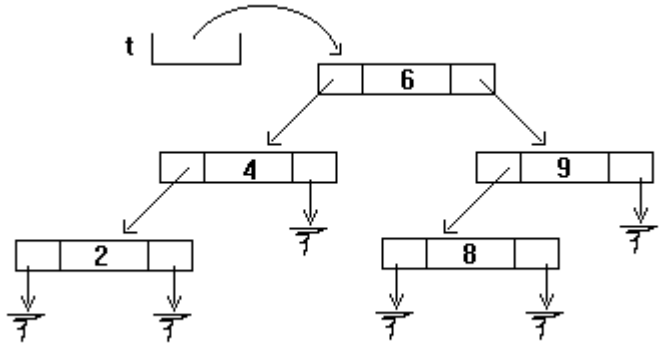
- hay árboles de tamaño arbitrario
- interesa representarlos como estructuras dinámicas

Por ello, la representación natural es con punteros:

el árbol:



se representará:



Definimos entonces:

```
typedef ABNodo* AB;

typedef struct ABNodo{

                int elem;
                AB izq, der;
};
```

Veamos ahora la función profundidad :

```
int ABProf (AB t){ //ABPRof(AB t) es lo mismo que ABPRof(ABNodo* t)

      if (t == NULL)
            return 0;
      else
            return 1 + max (ABProf (t->izq), ABProf (t->der));

}
```

Procedimiento de recorrida de un árbol binario en orden: (se aplica a cada nodo un procedimiento P que suponemos dado--Por ejemplo P podría ser: desplegar el elemento contenido en cada nodo).

```
void ABRecorridaP (AB t){

      if (t== NULL)
            //acción
      else{
            ABRecorridaP (t->izq);
            P (t->elem);
            ABRecorridaP (t->der)
      }
}
```

Suponemos aquí que P actúa sólo sobre los elementos (información en los nodos) y no sobre los punteros (estructura del árbol).

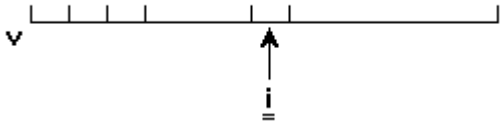
Dejamos abierta la posibilidad de una acción final, a realizar para cada subárbol vacío.

¿Cómo se escriben algoritmos iterativos sobre árboles?

Consideremos el caso de recorrida en primer orden.

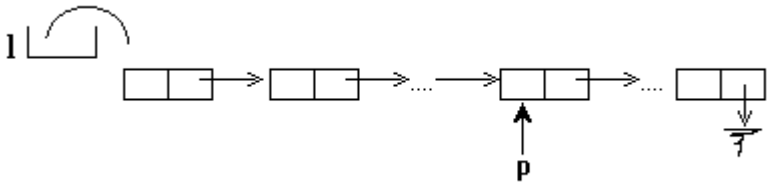
- recordemos las ideas de los algoritmos que recorren estructuras lineales (vectores, listas):

  Dado un vector v a recorrer:



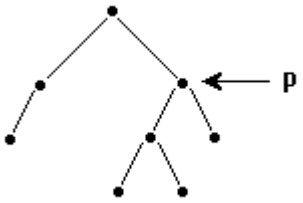
  utilizamos un índice i que señala al elemento a ser visitado.

  En el caso de las listas, usamos un puntero:



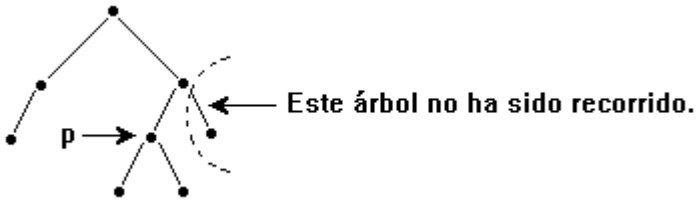
  De hecho, también necesitamos saber cuál es el resto de la estructura que queda por recorrer.
  En el caso de estructuras lineales, este resto queda trivialmente determinado, por la propia estructura.

¿Qué pasa en el caso de los árboles binarios?

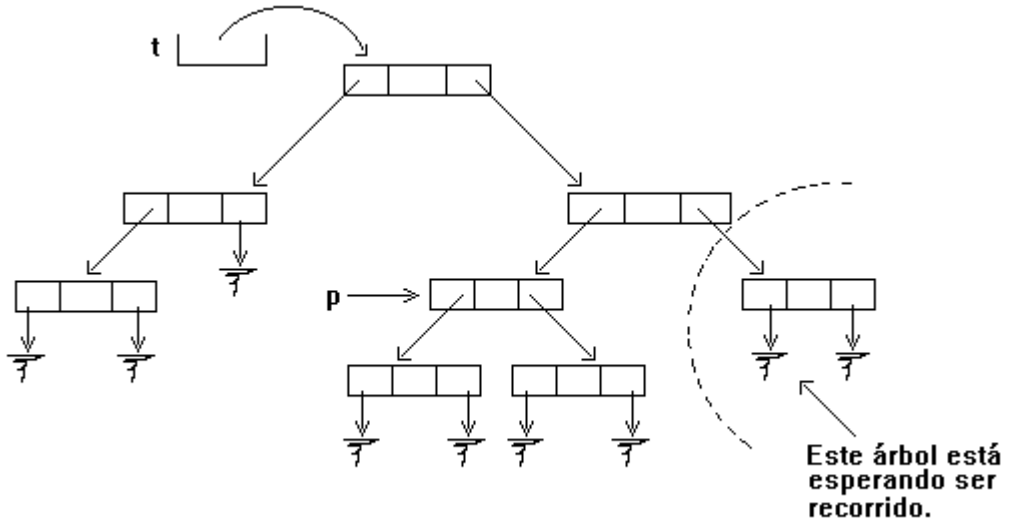


1. Debemos utilizar un puntero (P) para indicar el siguiente nodo a visitar.
2. Dado que estamos considerando la recorrida en pre-orden, entonces aplicaremos la acción que nos interese al nodo corriente y luego continuaremos la recorrida, primero por el subárbol izquierdo y luego por el derecho.

Por lo tanto: al continuar hacia el subárbol izquierdo, dejamos pendiente la recorrida del subárbol derecho.
La situación al mover el puntero P es como sigue:



Veamos el árbol representado con punteros, de acuerdo a las definiciones dadas antes:

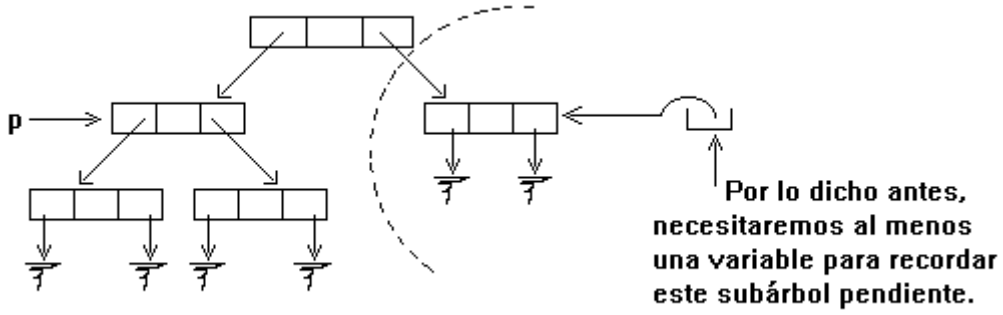


¡¡¡ Y NO HAY MANERA DE VOLVER A EL DESDE DONDE ESTAMOS !!!

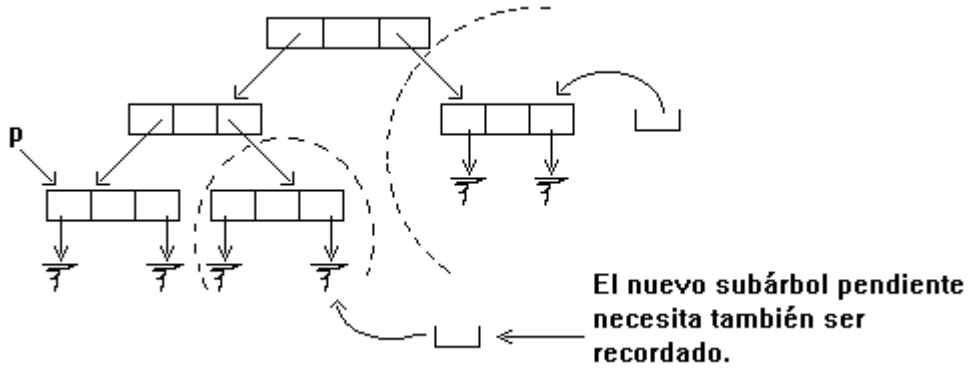
Conclusión: hay que guardar en alguna estructura de datos los punteros a los subárboles que van quedando "pendientes".

¿Cuál es la estructura de datos que necesitamos?

Consideremos un caso más simple:



Al avanzar p aparecerá otro subárbol pendiente.



-- De hecho tenemos que recordar todos los sucesivos subárboles derechos que van apareciendo durante la recorrida.

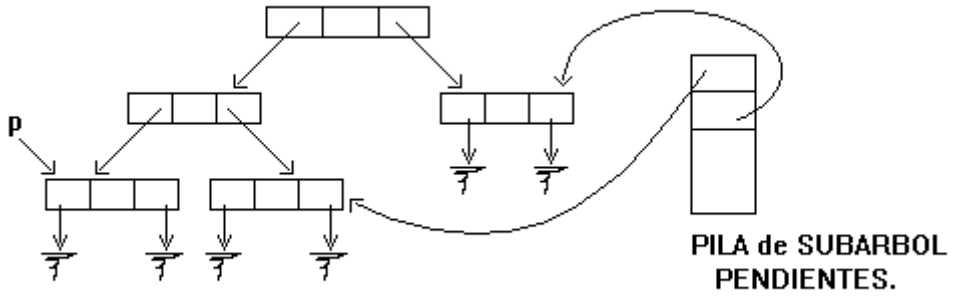
(*)Cada nuevo subárbol derecho que aparece debe ser recorrido antes que los aparecidos anteriormente

Por lo tanto: podemos apilar los punteros a los subárboles pendientes.

Dicho de otra forma: estructuramos los punteros a subárboles pendientes en una pila.

El tope será el primer subárbol pendiente a recorrer. Debido a la observación marcada (*) arriba, cada nuevo subárbol derecho que aparezca deberá ser colocado como nuevo tope de la pila.

Conclusión: la estructura de datos a ser utilizada para recorrer un árbol binario es como sigue:



- Con esta idea puede completarse la versión iterativa del algoritmo de recorrida en pre-orden. (Ejercicio).
- Se concluye que, en general, la programación iterativa de algoritmos en árboles es MUCHO más compleja que la recurrente.
- Dado que los programas iterativos son en general más eficientes, se hace interesante estudiar métodos de transformación de algoritmos recurrentes en iterativos equivalentes.

Si estos métodos pueden ser automatizados, entonces pueden ser aplicados directamente por un compilador. En tal caso, podemos construir programas simples (recurrentes) sin tener que preocuparnos por la pérdida de eficiencia.

(Los algoritmos recurrentes sobre árboles son de mayor nivel de abstracción que los iterativos).

Inserción en un árbol binario ordenado:

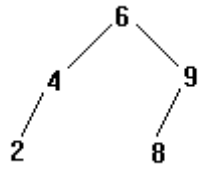
```
void ABOIns (int x, AB &t){
/* Precondición: t está ordenado
Postcondición: x es insertado en t, manteniendo el orden */

      if (t== NULL) {
```

```
            t= New ABNodo;
            t->elem = x;
            t->izq =t->der = NIL;
      }else if( x < t->elem )
            ABOIns (x, t->izq);
      else if( x > t->elem )
            ABOIns (x, t->der);
```

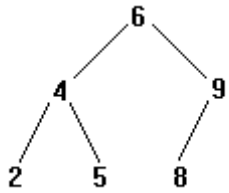
}

Notar que se inserta siempre una nueva hoja:

Ej: dado

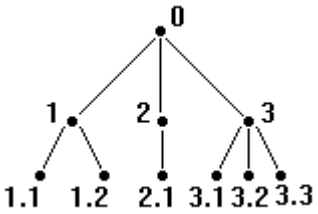


insertar 5 resulta en:



Representación de Finitarios como Binarios

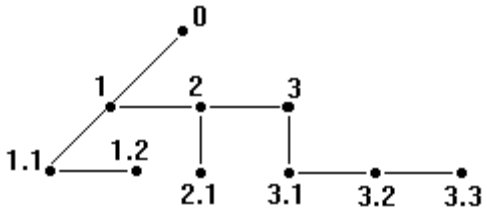
Dado un árbol finitario:



puede representarse la misma información en un árbol binario.

La regla de la transformación es:

> cada nodo apunta por izquierda a su primer hijo, los hermanos se encadenan por derecha.



Arbol Binario - Arbol Binario de Búsqueda

El TAD Arbol Binario

El tipo abstracto de datos Arbol Binario de elementos de tipo T especifica la noción de un conjunto finito de nodos tal que:

- o bien es vacío
- o consiste de un elemento de T (llamado **raíz**) y dos árboles binarios de tipo T disjuntos llamados **subárbol izquierdo y derecho** de la raíz

Se definen seis operaciones asociadas al TAD AB.

- Constructoras: **Vacio** y **CreoAB**.
  La primera operación permite construír un árbol vacio.
  Dados un elemento de tipo T y dos árboles binarios CreoAB construye un árbol no vacío.
- Predicado: EsVacio.
  Esta operación es usada para testear si un árbol es o no vacío
- Selectoras: Raiz, SubArbIzq y SubArbDer.
  Estas operaciones permiten seleccionar la raíz, el subárbol izquierdo y el derecho, respectivamente, de un árbol no vacío.

Módulo de definición para árboles binarios

```
#ifndef  ArbolBinario_H

#define  ArbolBinario_H

#include “TipoBase.h”
```

```
typedef struct NodoAB* AB;

AB  Vacio ();
/* retorna un árbol vacío */

AB  CreoAB ( T r, AB i, AB d);
/* retorna un nuevo árbol con raíz r y subárboles i y d. */

bool EsVacio ( AB  ab);

/* retorna TRUE si ab es vacío */

T Raiz ( AB ab);
/* pre: ab es no vacío. post: retorna la raíz de ab */

AB  SubArbIzq ( AB ab);
/* pre: ab es no vacío. post: retorna subárbol izquierdo de ab */

AB SubArbDer (AB ab);
/* pre: ab es no vacío. post: retorna subárbol derecho de ab */

#endif
```

Aplicación

Escribir un procedimiento que dado un árbol binario retorna el número de nodos que contiene.

Especificación

- Nodos (Vacio()) = 0
- Nodos (CreoAb(r, i, d)) = 1 + Nodos(i) + Nodos(d)

Implementación

```
int CatnNodos (  AB ab ){
/* retorna el número de nodos de ab */

//Declaración de variables

        int n;


if (Vacio(ab))

  num = 0;

else{

                num = 1 + Nodos(SubArbIzq(ab)) +
                Nodos(SubArbDer(ab));
```

```
}
return num;

}//fin cantnodos
```

Implementación del TAD AB

Veremos ahora una implementación del tipo abstracto de árboles binarios usando punteros (implementación dinámica).

```
IMPLEMENTATION MODULE ArbolBinario;
#include “ArbolBinario.h”

#include <stdlib.h>



        struct NodoAB {

                T raiz;
                AB izq;

                AB der;

        };


AB Vacio (){
    return NULL;

}

AB CreoAB ( T r,  AB i , AB d) {
  AB Temp ;

   Temp=new NodoAB;

   Temp->raiz = r;

   Temp->izq= i;
    Temp->der = d;

    return Temp;

}

boolean EsVacio (AB ab) {
```

```
        return (ab ==NULL);

}

T Raiz ( AB ab){

        return ab->raiz;

}

AB SubArbIzq ( AB ab){
  return ab->izq;

}

AB SubArbDer ( AB ab){
  return ab->der ;

}
```

El TAD Arbol Binario de Búsqueda

Los árboles binarios son frecuentemente usados para representar un conjunto de datos cuyos elementos pueden ser recuperables (y por lo tanto identificables) por medio de una clave única.

- Si un árbol está organizado de forma tal que para cada nodo n_i, todas las claves en el subárbol izquierdo de n_i son menores que la clave de n_i, y todas las claves en el subárbol derecho de n_i son mayores que la clave de n_i, entonces este árbol es un *ABB*.
- Los valores en los nodos de un ABB pueden ser números, caracteres, o estructuras complejas. Lo importante es que haya un orden total definido sobre el conjunto de los valores.

Los árboles binarios de búsqueda son también llamados árboles ordenados.
Son una subclase muy importante del TAD AB ya que permiten el ordenamiento y búsqueda de datos en forma simple y eficiente.

La especificación del TAD ABB es muy similar a la de AB. La diferencia más importante es que la constructora CreoAB no debería ser visible para el usuario del tipo abstracto ya que su uso podría permitir crear árboles desordenados.

Para construír ABBs se necesita una nueva operación:

```
AB Insertar ( T t, AB ab );
/* pre: ab es un árbol binario de búsqueda. post: retorna un ABB con t insertado
en la posición correcta. */
```

La semántica de esta operación la podemos especificar de la siguiente forma:

```
Insert(t, Vacio()) = CreoAb (i, Vacio(), Vacio())
Insert(t, CreoAb(r, i, d) =

                    if t = r

                        then

                            CreoAb(r, i, d)

                        else

                            if t < r

                                then

                                    CreoAb(r, Insertar(t,i), d)

                                else

                                    CreoAb(r, i, Insertar(t,d))

                            end

                    end
```

Notar que debido al orden impuesto sobre los elementos de un ABB no vacío el menor de ellos es aquel que está almacenado en el nodo más a la izquierda del mismo. Podemos entonces definir un procedimiento para localizar el mínimo elemento de un ABB:

```
T Minimo (AB ab){
/* pre: ab es un ABB no vacío post: retorna el mínimo elemento de ab */


    if (EsVacio(SubArbIzq(ab)))

                return  Raiz(ab);

     else

          return Minimo(SubArbIzq(ab));



}//fin Mínimo;
```

En forma similar podemos definir un procedimiento para localizar el máximo elemento de un ABB:

T Maximo (AB ab);

/* pre: ab es un ABB no vacío post: retorna el máximo elemento de ab */

Borrar un elemento

Esta operación, dada una clave i y un ABB, retorna otro ABB del que se ha borrado el item con clave i (si existe).

Solución:

Si el item a borrar se encuentra en la raíz del árbol, primero hay dos simples casos:

1. si el subárbol izquierdo es vacío entonces devolvemos el subárbol derecho
2. el caso dual

Si ninguno de los subárboles es vacío,entonces debemos encontrar un elemento con el cual reemplazar el item a borrar, preservando además la propiedad de ABB del árbol.

Dos posibles candidatos:

1. el elemento mayor del subárbol izquierdo
2. el elemento menor del subárbol derecho.