Algoritmos y Programación II
75.41

Cátedra: Lic. Gustavo Carolo

Guía de Estudio – Árboles

Enero 2005

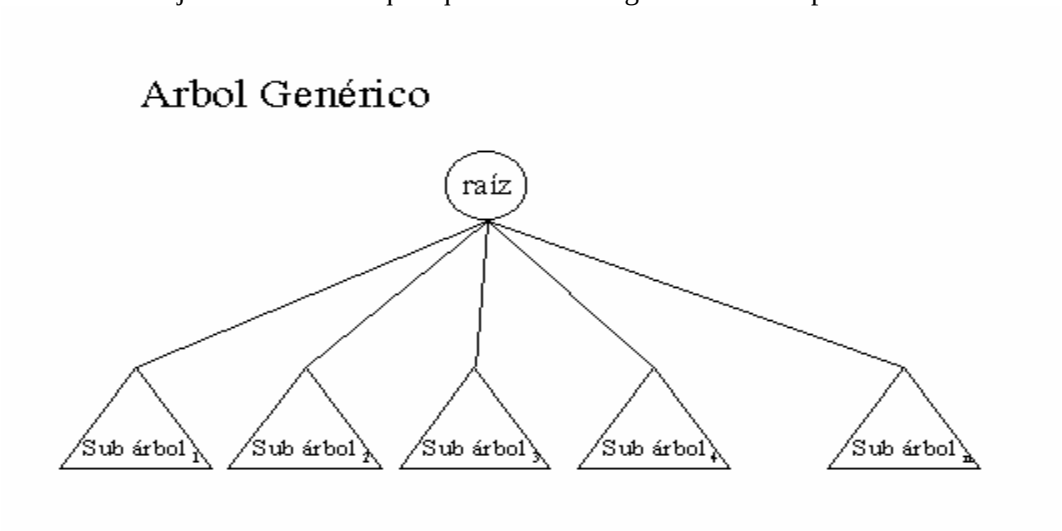
Indice

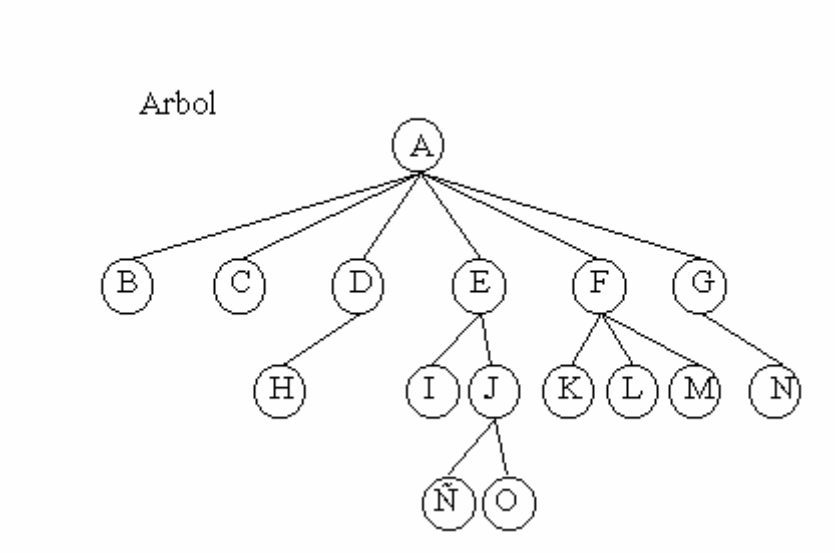
Definición

Un árbol es una colección o conjunto de nodos relacionados. La definición más natural es la recursiva. Esta formado por un nodo *raíz* la cual tiene *0 a n (sub)árboles* que se relacionan por medio de la arista o rama.

Se dice que el árbol tiene N niveles, siendo N también entonces la cantidad de nodos de la rama más larga. Una rama es una secuencia de nodos que se extiende desde la raíz del árbol hasta un nodo Hoja o Terminal.

Un nodo hoja o terminal es aquel que no tiene ningún subárbol dependiente.





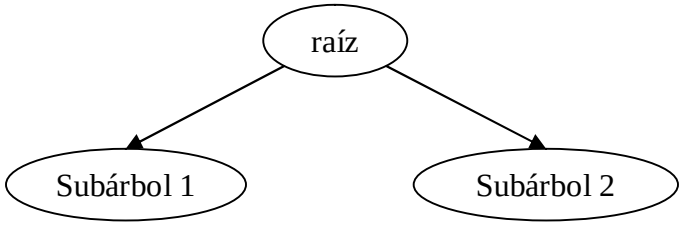
Nodo Raíz: A
Nodos Hojas o Terminales: B-C-H-I-Ñ-O-K-L-M-N
Nivel 1 del Árbol: A
Nivel 4 del Árbol: Ñ-O
Profundidad del Árbol: 4
Ramas más larga: A-E-J-Ñ // A-E-J-O

Árbol Binario

Un árbol binario es una clase especial de árbol, donde cada subnodo tiene como máximo dos hijos o subnodos relacionados. A estos dos subnodos se los conoce normalmente como "hijo izquierdo" e "hijo derecho" respectivamente del nodo.

En el árbol genérico como el de la figura que se encuentra más arriba en esta página el número de hijos de cada nodo puede variar bastante, siendo por ello difícil de implementar en un sistema computacional donde los recursos son limitados y tienen que ser usados en forma eficiente.

Un árbol binario tiene a lo sumo dos hijos, por lo cual se puede implementar mucho más fácilmente.



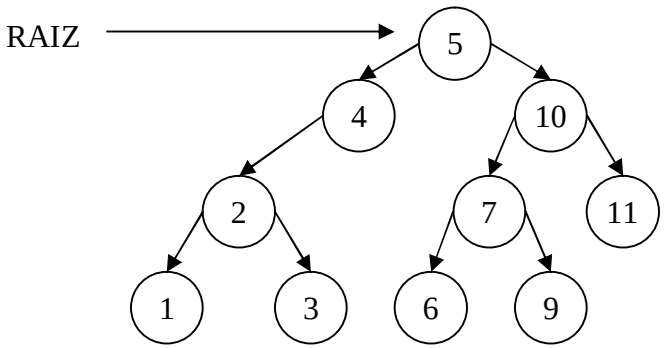
Uno de los usos principales del árbol binario es el de búsquedas binarias. Cada uno de los elementos de los nodos tiene que estar identificado por una clave.

Para la implementación de estos tipos de árboles, se decide un criterio para la ubicación y posterior búsqueda de los nodos, generalmente todas las claves menores se ubican a izquierda y las mayores a derecha.

Cuando se busca una determinada clave, se aplica el concepto "recursivo" del árbol, de la siguiente forma: Se compara la clave buscada con la clave del elemento ubicado en la raíz del árbol. Si es el elemento buscado, finalizó la búsqueda. En caso contrario, se evaluará si la clave buscada es menor que la del nodo raíz, entonces se continuará la búsqueda solo en el subárbol izquierdo que contiene las claves menores a la actual y en caso contrario se continuará la búsqueda solo en el subárbol derecho. Para los subárboles se aplica el mismo procedimiento hasta encontrarlo (de ahí el concepto recursivo).

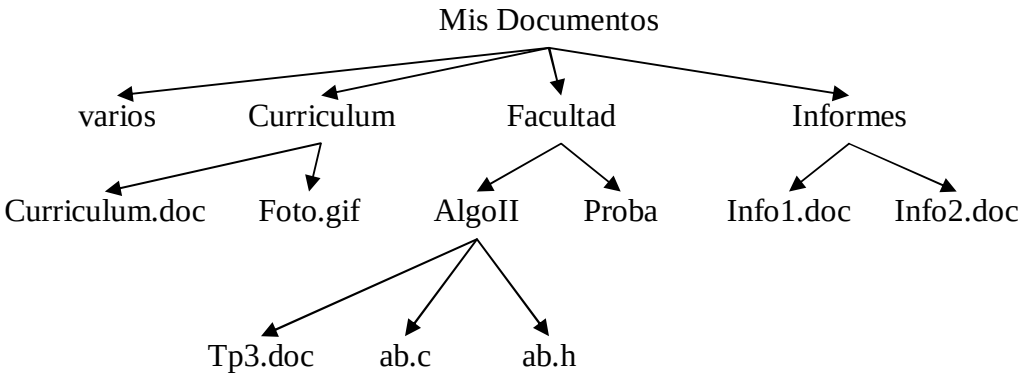
Si el árbol binario se encuentra "balanceado", es decir posee una cantidad pareja de nodos a izquierda y a derecha, la cantidad de búsquedas se reduce en forma logarítmica. Esto es, resulta más ventajosa cuanto mayor sea el número de nodos.

Árbol Binario de Búsqueda



Árbol General

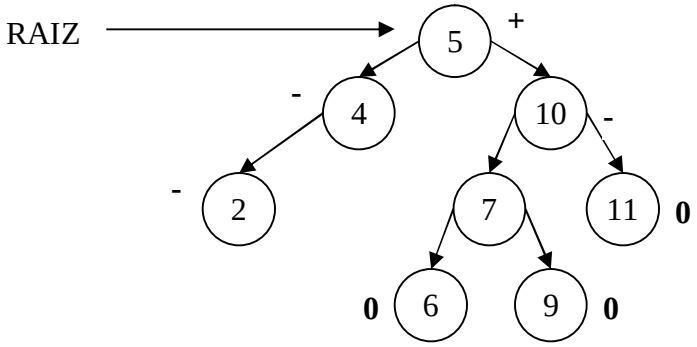
Es un tipo de árbol cuyos nodos permiten cualquier cantidad de hijos.
Un ejemplo de árbol general puede ser la estructura jerárquica de directorios de un sistema operativo.



Árbol AVL

Un árbol AVL (Adelson - Velskii y Landis) es un árbol binario de búsqueda con una condición de equilibrio.
Un árbol AVL es un árbol binario de búsqueda en el cual las alturas de los subárboles izquierdo y derecho de la raíz difieren cuando mucho en 1, y a su vez los subárboles izquierdo y derecho de la raíz son árboles AVL.

Cada nodo de un árbol AVL tiene asociado un factor de balance que indica la diferencia entre las alturas de los sub-árboles.



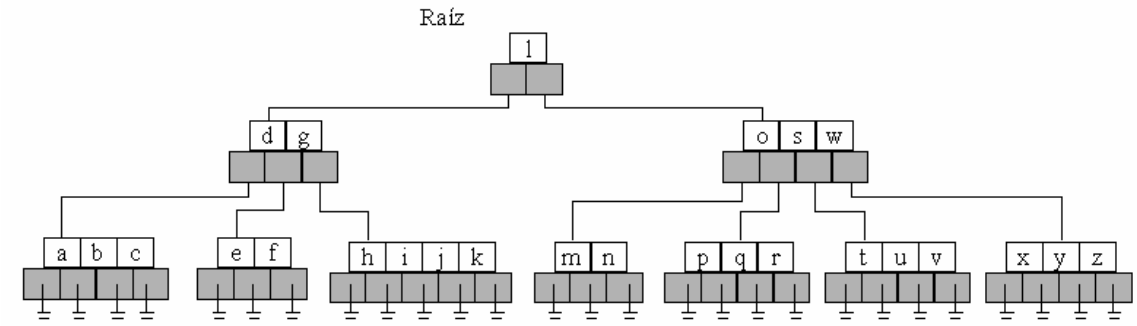
Árbol B

Aunque todos los árboles de búsqueda vistos son binarios, existe un árbol de búsqueda ampliamente utilizado que no es binario. Este es el llamado árbol-B.

Un árbol B es un árbol n-ario al cual se le asocia un número natural m llamado orden.
Un árbol B de orden m es un árbol con las siguientes propiedades estructurales:

- La raíz es una hoja o tiene entre 2 o más hijos.
- Todos los nodos internos (los nodos que no son hojas, excepto la raíz) tiene entre [m/2] y m hijos.
- Todos los nodos internos tienen una cantidad de claves igual a su cantidad de hijos - 1.
- Todas las hojas están a la misma profundidad.

Árbol B de orden 5



Implementación de Árboles Binarios con Punteros

AB.H

```
#ifndef __AB_H__
   #define __AB_H__

   #include <stdlib.h>

   #define IZQ 1
   #define DER 2
   #define PAD 3
   #define RAIZ 4

   typedef struct TNodoAB
   {
      void* elem;
      struct TNodoAB *izq, *der;
   } TNodoAB;

   typedef struct TAB
   {
      TNodoAB *raiz,*cte;
      int tamdato;
   } TAB;

   void AB_Crear(TAB *a,int tdato);

   void AB_ElemCte(TAB a,void *elem);

   void AB_ModifCte(TAB *a,void *elem);

   void AB_MoverCte(TAB *a,int mov,int *error);

   void AB_Vaciar(TAB *a);

   int AB_Vacio(TAB a);

   void AB_Insertar(TAB *a,int mov,void *elem,int *error);

#endif
```

AB.C

```
#include "ab.h"
#include <memory.h>

void AB_Crear(TAB *a,int tdato)
{
   a->tamdato = tdato;
   a->raiz = NULL;
   a->cte = NULL;
}

void AB_ElemCte(TAB a,void *elem)
{
   memcpy(elem,a.cte->elem,a.tamdato);
}

void AB_ModifCte(TAB *a,void *elem)
{
   memcpy(a->cte->elem,elem,a->tamdato);
}

TNodoAB* BuscarPadre(TNodoAB *padre,TNodoAB *hijo)
{
   TNodoAB *paux = NULL;

   if ((padre->izq == hijo) || (padre->der == hijo))
      return padre;
   else
   {
      if (padre->izq != NULL)
         paux = BuscarPadre(padre->izq,hijo);

      if ((padre->der != NULL) && (paux == NULL))
         paux = BuscarPadre(padre->der,hijo);

      return paux;
   }
}

void AB_MoverCte(TAB *a,int mov,int *error)
{
   *error = 0;
   switch (mov)
   {
      case IZQ : if (a->cte->izq != NULL)
                    a->cte = a->cte->izq;
                 else
                    *error = 1;
                 break;

      case DER : if (a->cte->der != NULL)
                    a->cte = a->cte->der;
                 else
                    *error = 1;
                 break;

      case PAD : if (a->cte != a->raiz)
                    a->cte = BuscarPadre(a->raiz,a->cte);
                 else
```

```
                  *error = 1;
               break;

     case RAIZ :  if (a->raiz != NULL)
                     a->cte = a->raiz;
                  else
                     *error = 1;
                  break;
     default :    *error = 1;
   }
}

void VaciarSub(TNodoAB *pnodo)
{
   if (pnodo != NULL)
   {
      if ((pnodo->izq == NULL) && (pnodo->der == NULL))
      {
         free(pnodo->elem);
         free(pnodo);
      }
      else
      {
         VaciarSub(pnodo->izq);
         VaciarSub(pnodo->der);
         free(pnodo->elem);
         free(pnodo);
      }
   }
}

void AB_Vaciar(TAB *a)
{
   VaciarSub(a->raiz);
   a->raiz = NULL;
   a->cte = NULL;
}

void AB_Insertar(TAB *a,int mov,void *elem,int *error)
{
   TNodoAB *paux;
   *error = 0;
   paux = (TNodoAB*) malloc(sizeof(TNodoAB));
   paux->izq = NULL;
   paux->der = NULL;
   if (paux)
   {
      paux->elem = malloc(a->tamdato);
      if (paux->elem)
      {
         memcpy(paux->elem,elem,a->tamdato);
         switch(mov)
         {
            case IZQ :   if (a->cte->izq == NULL)
                            a->cte->izq = paux;
                         else
                            *error = 1;
                         break;

            case DER :   if (a->cte->der == NULL)
                            a->cte->der = paux;
                         else
```

```
                        *error = 1;
                     break;

         case RAIZ :  if (a->raiz == NULL)
                        a->raiz = paux;
                     else
                        *error = 1;
                     break;
         default :    *error = 1;
       }
       if (*error)
       {
          free(paux->elem);
          free(paux);
       }
       else
          a->cte = paux;
    }
    else /* if (paux->elem) */
    {
       *error = 1;
       free(paux);
    }
  }
  else /* if (paux) */
     *error = 1;
}

int AB_Vacio(TAB a)
{
  if (a.raiz == NULL)
     return 1;
  else
     return 0;
}
```

Recorridos de un Árbol

Existen varias formas de recorrer un árbol binario, desde el punto de vista del orden en que se recorren los elementos:

PreOrden

Primero se procesa el nodo corriente, luego el subárbol izquierdo, y luego el subárbol derecho.

```
void PreOrden(TAB ab, int MOV)
{
   TelemAB eab;
   int error;

   AB_MoverCte(&ab,MOV,&error);
   if ( !error )
   {
      AB_ElemCte(ab,&eab);
      procesar(eab);

      PreOrden(ab,IZQ);
      PreOrden(ab,DER);
   }
}
```

InOrden

Primero se procesa el subárbol izquierdo, luego el nodo corriente, y luego el subárbol derecho.

```
void InOrden(TAB ab, int MOV)
{
   TelemAB eab;
   int error;

   AB_MoverCte(&ab,MOV,&error);
   if ( !error )
   {
      InOrden(ab,IZQ);
      AB_ElemCte(ab,&eab);
      procesar(eab);
      InOrden(ab,DER);
   }
}
```

PosOrden

Primero se procesa el subárbol izquierdo, luego el subárbol derecho, y luego el nodo corriente.

```
void PosOrden(TAB ab, int MOV)
{
   TelemAB eab;
   int error;

   AB_MoverCte(&ab,MOV,&error);
   if ( !error )
   {
      PosOrden(ab,IZQ);
      PosOrden(ab,DER);
      AB_ElemCte(ab,&eab);
      procesar(eab);
   }
}
```

Árbol Binario Ordenado (ABO)

Este tipo de árbol se caracteriza por manejar desde las primitivas la inserción ordenada de los elementos. Para ello, la primitiva utilizará una función de comparación que será indicada por parámetro al momento de la creación del árbol. Además, se incorpora una primitiva que permite buscar un elemento por su clave.
Para utilizar la función de comparación se hace uso de una característica de C que son los punteros a funciones. De esta forma es posible guardar en un puntero el llamado a una función, en nuestro caso se usara una función que recibe los dos elementos a comparar y devuelve un entero indicando que elemento si los elementos son distintos o iguales. Así, es posible guardar en este árbol cualquier tipo de elementos indicando la función de comparación correspondiente (Ver el apunte de C para más información sobre punteros a funciones)

Implementación

Solo se muestran las primitivas que cambian respecto al Árbol Binario

ABO.H

```
#ifndef __TDA_ABO_H__
   #define __TDA_ABO_H__

   #include <stdlib.h>

   #define IZQ 1
   #define DER 2
   #define PAD 3
   #define RAIZ 4
```

```
   typedef struct TNodoABO
   {
      void* elem;
      struct TNodoABO *izq, *der;
   } TNodoABO;

   typedef struct TAB
   {
      TNodoABO *raiz,*cte;
      int tamdato;
      int (*fcomp)(void *,void *); /* Puntero a la función de comparación */
   } TABO;

   void ABO_Crear(TABO *a,int tdato, int (*comp)(void *,void *));

   void ABO_Insertar(TABO *a,void *elem,int *error);

#endif
```

ABO.C

```
#include "tda_abo.h"

void ABO_Crear(TABO *a,int tdato, int (*comp)(void *,void *))
{
   a->tamdato = tdato;
   a->raiz = NULL;
   a->cte = NULL;
   a->fcomp = comp;
}


int buscar_lugar(TABO *a, TNodoABO **p, void *elem)
{
   int c;

   if (*p != NULL)
   {
      c = (*(a->fcomp))((*p)->elem,elem);

      if (c == 1)
         buscar_lugar(a,&((*p)->izq),elem);
      else if (c == -1)
         buscar_lugar(a,&((*p)->der),elem);
      else
         return 1;
   }
   else
   {
      *p = (TNodoABO*) malloc(sizeof(TNodoABO));
      (*p)->izq = NULL;
      (*p)->der = NULL;
      if (*p)
      {
         (*p)->elem = malloc(a->tamdato);
         if ((*p)->elem)
         {
            memcpy((*p)->elem,elem,a->tamdato);
            return 0;
```

```
        }
        free(*p);
        return 1;
      }
      return 1;
   }
   return 1;
}

void ABO_Insertar(TABO *a, void *elem,int *error)
{
   *error = buscar_lugar(a,&(a->raiz),elem);
}
```