## Ejercicios de programación con árboles

Por CCG/Mayo2014

1. Escriba un programa que acepte un apuntador a un nodo y devuelva un valor verdadero si este nodo es la raíz de un árbol binario válido y falso en caso contrario.

2. Escriba un programa que acepte un apuntador a un árbol binario y un apuntador a un nodo del árbol, y devuelva el nivel del nodo en el árbol.

3. Escriba un programa para ejecutar el experimento siguiente: genere 100 números aleatorios. Conforme se genera cada número, insértelo en un árbol de búsqueda binaria inicialmente vacío. Después de insertar los 100 números, imprima el nivel de la hoja que tiene el nivel más grande y el nivel de la hoja que tiene el nivel más chico. Repita este proceso 50 veces. Imprima una tabla que indique cuantas veces de las 50 ejecuciones produjeron una diferencia entre el nivel de hoja máximo y mínimo de 0, 1, 2, 3, y así sucesivamente.

4. Implemente los recorridos de los árboles binarios.

5. Si un bosque se representa mediante un árbol binario, muestre que el número de vínculos derechos nulos es 1 mayor que el número de no hojas del bosque.

6.- Supongamos que tenemos una función valor tal que dado un valor de tipo char (una letra del alfabeto) devuelve un valor entero asociado a dicho identificador. Supongamos también la existencia de un árbol de expresión T cuyos nodos hoja son letras del alfabeto y cuyos nodos interiores son los caracteres *,+,-,/.Diseñar una función que tome como parámetros un nodo y un árbol binario y devuelva el resultado entero de la evaluación de la expresión.

```
int Evalua(NodoB n,ArbolB T)
{
 char ident;

 EtiquetaArbolB(&c,n,T);
 switch(c){
  case '+':
   return Evalua(HijoIzqdaB(n,T),T)+Evalua(HijoDrchaB(n,T),T);
   break;
  case '-':
   return Evalua(HijoIzqdaB(n,T),T)-Evalua(HijoDrchaB(n,T),T);
   break;
  case '*':
   return Evalua(HijoIzqdaB(n,T),T)*Evalua(HijoDrchaB(n,T),T);
   break;
  case '/':
   return Evalua(HijoIzqdaB(n,T),T)/Evalua(HijoDrchaB(n,T),T);
   break;
  default:
   return valor(c);
 }
}
```

7.-El recorrido en preorden de un determinado árbol binario es: GEAIBMCLDFKJH y en inorden IABEGLDCFMKHJ .Resolver:

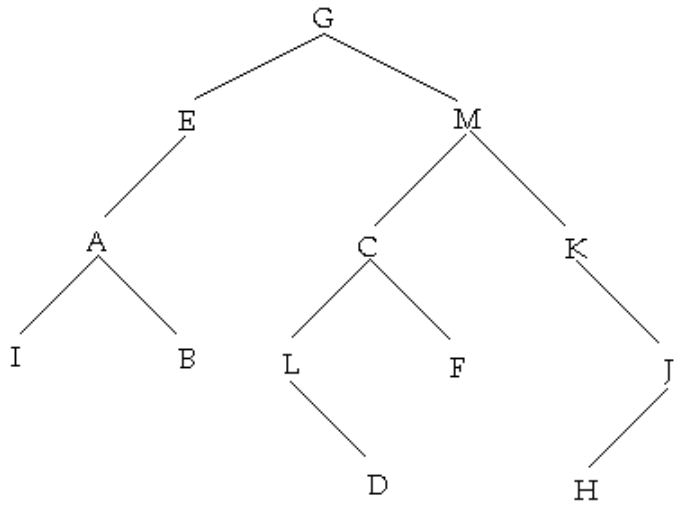
A) Dibujar el árbol binario.

B) Dar el recorrido en postorden.

C) Diseñar una función para dar el recorrido en postorden dado el recorrido en preorden e inorden y escribir un programa para comprobar el resultado del apartado anterior.

**SOLUCIÓNES**

A) El árbol correspondiente es el de la siguiente figura:




B) El recorrido en postorden es IBAEDLFCHJKMG.

C) El código solución al tercer apartado es el siguiente:

```
#include < stdio.h>
#include < stdlib.h>
#include < string.h>

char *preorden="GEAIBMCLDFKJH";
char *inorden="IABEGLDCFMKHJ";
char *postorden;

/*--------------------------------------*/

void post(char *pre,char *in,char *pos,int n)
{
  int longIzqda;

  if(n!=0){
    pos[n-1]=pre[0];
    longIzqda=strchr(in,pre[0])-in;
    post (pre+1,in,pos,longIzqda);
    post (pre+1+longIzqda,in+1+longIzqda,pos+longIzqda,n-1-longIzqda);
  }
}
```

```
/*---------------------------------------*/

int main(int argc,char *argv[])
{
  int aux;

  aux=strlen(preorden);
  postorden=(char *)malloc(aux*sizeof(char));
  if (postorden){
    printf("El preorden es: %s\n",preorden);
    printf("El inorden es: %s\n",inorden);
    post(preorden,inorden,postorden,aux);
    postorden[aux]='\0';
    printf("El postorden calculado es: %s\n",postorden);
    free(postorden);
  }
  else{
    fprintf(stderr,"Error: Sin memoria\n");
    return 1;
  }

  return 0;
}
```

8.-Implementar una función no recursiva para recorrer un árbol binario en inorden.

**SOLUCIÓN**

```
#include < pilas.h>
#include < arbolesB.h>

/*---------------------------------------*/

void inordenNR(ArbolB T,void (* EscribirElemento)(void *),int tamano)
{
 NodoB nodoActual,aux;
 void *et;
 Pila p;
 int fin;
 int faltaHD;                  /*Indica si falta el hijo derecho*/

 p=CrearPila(sizeof(NodoB));
 et=malloc(tamano);
 if(!et){
  ....                    /*Error:Sin memoria*/
 }

 aux=NODOB_NULO;
 Push(&aux,p);
 nodoActual=RaizB(T);
 fin=0;
 while(!fin){
  while(nodoActual!=NODOB_NULO){
   Push(&nodoActual,p);
   nodoActual=HijoIzqdaB(nodoActual,T);
```

```
   }
   Tope(&nodoActual,p);
   Pop(p);
   if (nodoActual!=NODOB_NULO){
    EtiquetaArbolB(et,nodoActual,T);
    (*EscribirElemento)(et);
    nodoActual=HijoDrchaB(nodoActual,T);
   }
   else fin=1;
 }

 free(et);
 DestruirPila(p);
}
```

9.- Implementar una función no recursiva para recorrer un árbol binario en postorden.

## SOLUCIÓN

10.- Escribir una función que realice la reflexión de un árbol binario.

## SOLUCIÓN

El código es el siguiente:

```
void Refleja(ArbolB T)
{
  ArbolB Ti,Td;

  if(T!=ARBOLB_VACIO){
    Ti=PodarHijoIzqdaB(RaizB(T),T);
    Td=PodarHijoDrchaB(RaizB(T),T);
    Refleja(Ti);
    InsertarHijoDrchaB(RaizB(T),Ti,T);
    Refleja(Td);
    InsertarHijoIzqdaB(RaizB(T),Td,T);
  }
}
```

11.- Escribir una función recursiva que encuentre el número de nodos de un árbol binario.

## SOLUCIÓN

El algoritmo es muy sencillo considerando que el número de nodos de un árbol binario es el número de nodos del hijo a la izquierda más el de su hijo a la derecha más 1.El código es el siguiente:

```
int numero(NodoB n,ArbolB T)
```

```
{
  if (n==NODOB_NULO)
    return 0;
  else
    return 1+numero(HijoIzqdaB(n,T),T)+numero(HijoDrchaB(n,T),T);
}
```

Para calcular el número de nodos de un árbol T se haría mediante la llamada numero(RaizB(T),T).

12.- Escribir una función recursiva que encuentre la altura de un árbol binario.

## SOLUCIÓN

La altura de un árbol T es uno más el máximo de alturas de los subárboles izquierdo y derecho (La altura de un árbol nulo está indefinida).El código es el siguiente:

```
#define MAXIMO(a,b) ((a) < (b)?(b):(a))

int altura(NodoN n,ArbolB T)
{
  if(n==NODOB_NULO)
    return -1;
  else
    return 1+MAXIMO(altura(HijoIzqdaB(n,T),T),altura(HijoDrchaB(n,T),T));
}
```

# Teoría sobre árboles.

## Arboles

Los árboles son estructuras de datos útiles en muchas aplicaciones. Hay varias formas de árboles y cada una de ellas es práctica en situaciones especiales, en este capítulo vamos a definir algunas de esas formas y sus aplicaciones.

6.1. Concepto general de árbol

Desde el punto de vista de estructuras de datos, un árbol es un concepto simple en su definición, sin embargo es muy ingenioso. Un árbol es un grafo con características muy especiales:

Definición 9 Un árbol es un grafo A que tiene un único nodo llamado raíz que:

Tiene 0 relaciones, en cuyo caso se llama nodo hoja tiene un número finito de relaciones, en cuyo caso, cada una de esas relaciones es un subárbol

Para empezar a estudiar los árboles, nos concentraremos en primer lugar en el caso en que el nodo raíz tenga 0, 1 ´o 2 subárboles.

6.2. ´ Arboles binarios

Definición 10 Un árbol binario es una estructura de datos de tipo árbol en donde cada uno de los nodos del árbol puede tener 0, 1, ´o 2 subárboles llamados de acuerdo a su caso como:

Si el nodo raíz tiene 0 relaciones se llama hoja.

Si el nodo raíz tiene 1 relación a la izquierda, el segundo elemento de la relación es el subárbol izquierdo.

Si el nodo raíz tiene 1 relación a la derecha, el segundo elemento de la relación es el subárbol derecho.

La figura 25 muestra algunas configuraciones de grafos que sí son árboles binarios, y la figura 26 muestra algunas configuraciones de grafos que no son árboles binarios.

Vamos a dar una lista de términos que se usan frecuentemente cuando se trabaja con árboles:

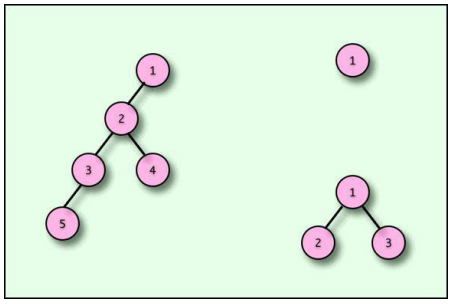



Figura  Grafos que son estructuras tipo árbol binario

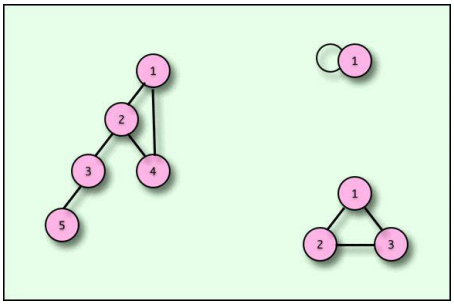



Figura. Grafos que no son árboles binarios

Si A es la raíz de un árbol y B es la raíz de su subárbol izquierdo (o derecho), se dice que A es el padre de B y se dice que B es el hijo izquierdo (o derecho) de A.

Un nodo que no tiene hijos se denomina hoja

El nodo a es antecesor del nodo b (y recíprocamente el nodo b es descendiente del nodo a), si a es el padre de b o el padre de algún ancestro de b.

Un nodo b es un descendiente izquierdo del nodo a, si b es el hijo izquierdo de a o un descendiente del hijo izquierdo de a. Un descendiente derecho se define de la misma forma.

Dos nodos son hermanos si son hijos izquierdo y derecho del mismo padre.

Otros términos relacionados con árboles, tienen que ver con su funcionamiento y topología:

Si cada nodo que NO es una hoja tiene un subárbol izquierdo y un subárbol derecho, entonces se trata de un árbol binario completo.

El nivel de un nodo es el número de aristas que se deben recorrer para llegar desde ese nodo al nodo raíz. De manera que el nivel del nodo raíz es 0, y el nivel de cualquier otro nodo es el nivel del padre más uno.

La profundidad de un nodo es el máximo nivel de cualquier hoja en el árbol.

Si un árbol binario tiene m nodos en el nivel l, el máximo número de nodos en el nivel l + 1 es 2m. Dado que un árbol binario solo tiene un nodo en el nivel 0, puede contener un máximo de 2l nodos en el nivel l. Un árbol binario completo de profundidad d es el árbol que contiene exactamente 2l nodos en cada nivel l entre 0 y d. La cantidad total de nodos tn en un árbol binario completo de profundidad d, es igual a la suma de nodos en cada nivel entre 0 y d, por tanto:

$$t_n = 2^0 + 2^1 + 2^2 + \cdots + 2^d = \sum_{j=0}^{d} 2^j$$

Usando inducción matemática se puede demostrar que $\sum_{j=0}^{d} 2^j = 2^{d+1} - 1.$

Dado que todas las hojas en este árbol están en el nivel d, el árbol contiene $2^d$ hojas y, por tanto, $2^d - 1$ nodos que no son hojas.

Si conocemos el número total de nodos tn en un árbol binario completo, podemos calcular su profundidad d, a partir de la expresión $tn = 2^{d+1}-1$. Así sabemos que la profundidad d es igual a 1

menos que el número de veces que 2 debe ser multiplicado por sí mismo para llegar a tn + 1. Es decir, que en un árbol binario completo,

$$d = \log_2(t_n + 1)$$

Definición 11 Un árbol binario es un árbol binario casi completo si:

1. Cualquier nodo nd a un nivel menor que d − 1 tiene 2 hijos

2. Para cualquier nodo nd en el árbol con un descendiente derecho en el nivel d debe tener un hijo izquierdo y cada descendiente izquierdo de nd:

- Es una hoja en el nivel d ó
- Tiene dos hijos

Los nodos en un árbol binario (completo, casi completo o incompleto) se pueden enumerar del siguiente modo. Al nodo raíz le corresponde el número 1, al hijo izquierdo le corresponde el doble del número asignado al padre y al hijo derecho le corresponde el doble más 1 del número asignado al padre.

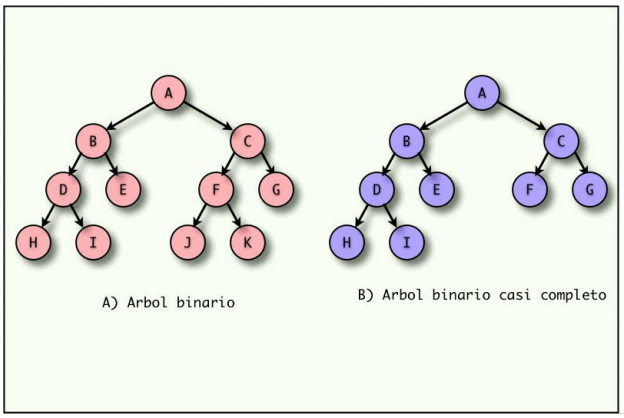



Figura. Comparación de un árbol binario y un árbol binario casi completo. El árbol mostrado en (A) descumple la regla 2 de los árboles binarios casi completos.

6.2.1. Operaciones con árboles binarios

Con los árboles binarios es posible definir algunas operaciones primitivas, estas operaciones son en el sentido de saber la información de un nodo y sirven para desplazarse en el árbol, hacia arriba o hacia abajo.

info(p) que devuelve el contenido del nodo apuntado por p.

left(p) devuelve un apuntador al hijo izquierdo del nodo apuntado por p, o bien, devuelve NULL si el nodo apuntado por p es una hoja.

right(p) devuelve un apuntador al hijo derecho del nodo apuntado por p, o bien, devuelve NULL si el nodo apuntado por p es una hoja.

father(p) devuelve un apuntador al padre del nodo apuntado por p, o bien, devuelve NULL si el nodo apuntado por p es la ra´ız.

brother(p) devuelve un apuntador al hermano del nodo apuntado por p, o bien, devuelve NULL si el nodo apuntado por p no tiene hermano.

Estas otras operaciones son l´ogicas, tienen que ver con la identidad de cada nodo:

isLeft(p) devuelve el valor true si el nodo actual es el hijo izquierdo del nodo apuntado por p, y false en caso contrario.

isRight(p) devuelve el valor true si el nodo actual es el hijo derecho del nodo apuntado por p, y false en caso contrario.

isBrother(p) devuelve el valor true si el nodo actual es el hermano del nodo apuntado por p, y false en caso contrario.

Como ejemplo, un algoritmo para el procedimiento isLeft es como sigue:

```
q=father(p);

if(q==NULL)

return(false)

/* porque p apunta a la raíz */

if (left(q)==p)

return(true);

return(false);
```

En la construcción de un árbol binario son útiles las operaciones makeTree, setLeft y setRight. La operación makeTree(x) crea un nuevo árbol binario que consta de un único nodo con un campo de información x y devuelve un apuntador a ese nodos. La operación setLeft(p,x) acepta un apuntador p a un nodo de árbol binario sin hijo izquierdo. Crea un nuevo hijo izquierdo de node(p) con el campo de información x. La operación setRight(p,x) es similar, excepto que crea un hijo derecho.

6.2.2. Aplicaciones de árboles binarios

Un árbol binario es una estructura de datos útil cuando se trata de hacer modelos de procesos en donde se requiere tomar decisiones en uno de dos sentidos en cada parte del proceso. Por ejemplo, supongamos que tenemos un arreglo en donde queremos encontrar todos los duplicados. Esta situación es bastante útil en el manejo de las bases de datos, para evitar un problema que se llama redundancia.

Una manera de encontrar los elementos duplicados en un arreglo es recorrer todo el arreglo y comparar con cada uno de los elementos del arreglo. Esto implica que si el arreglo tiene n elementos, se deben hacer n comparaciones, claro, no es mucho problema si n es un número pequeño, pero el problema se va complicando más a medida que n aumenta.

Si usamos un árbol binario, el número de comparaciones se reduce bastante, veamos cómo.

El primer número del arreglo se coloca en la raíz del árbol (como en este ejemplo siempre vamos a trabajar con árboles binarios, simplemente diremos árbol, para referirnos a un árbol binario) con

sus subárboles izquierdo y derecho vacíos. Luego, cada elemento del arreglo se compara son la información del nodo raíz y se crean los nuevos hijos con el siguiente criterio:

Si el elemento del arreglo es igual que la información del nodo raíz, entonces notificar duplicidad.

Si el elemento del arreglo es menor que la información del nodo raíz, entonces se crea un hijo izquierdo.

Si el elemento del arreglo es mayor que la información del nodo raíz, entonces se crea un hijo derecho.

Una vez que ya está creado el árbol, se pueden buscar los elementos repetidos.

Si x el elemento buscado, se debe recorrer el árbol del siguiente modo:

Sea k la información del nodo actual p. Si x > k entonces cambiar el nodo actual a right(p), en caso contrario, en caso de que x = k informar una ocurrencia duplicada y en caso de que x >= k cambiar el nodo actual a left(p).

El siguiente algoritmo

```
leer numero buscado >> n
tree=makeTree(n)
while(hay numeros en el arreglo){
leeSiguienteNumero >> k
p=q=tree;
while(k!=info(p)&&q!=NULL){
p=q
if(k<info(p))
q=left(p)
else
q=right(p)
}
if(k==info(p))
despliega<<" el numero es duplicado";
else
if (k<info(p))
setLeft(p,k)
else
setRight(p,k)
}
```

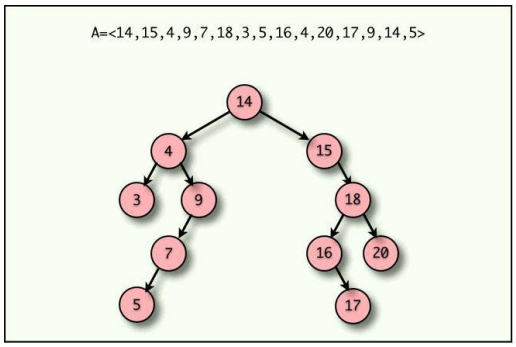



Figura. Árbol binario para encontrar números duplicados

Para saber el contenido de todos los nodos en un árbol es necesario recorrer el árbol. Esto es debido a que solo tenemos conocimiento del contenido de la dirección de un nodo a la vez. Al recorrer el árbol es necesario tener la dirección de cada nodo, no necesariamente todos al mismo tiempo, de hecho normalmente se tiene la dirección de uno o dos nodos a la vez; de manera que cuando se tiene la dirección de un nodo, se dice que se visita ese nodo.

Aunque hay un orden preestablecido (la enumeración de los nodos) no siempre es bueno recorrer el árbol en ese orden, porque el manejo de los apuntadores se vuelve más complejo. En su lugar se han adoptado tres criterios principales para recorrer un árbol binario, sin que se omita cualquier otro criterio diferente.

Los tres criterios principales para recorrer un árbol binario y visitar todos sus nodos son, recorrer el árbol en:

Preorden: Se ejecutan las operaciones:

1. Visitar la raíz

2. recorrer el subárbol izquierdo en Preorden

3. recorrer el subárbol derecho en Preorden

Entreorden: Se ejecutan las operaciones:

1. recorrer el subárbol izquierdo en entreorden

2. Visitar la raíz

3. recorrer el subárbol derecho en entreorden

Postorden: Se ejecutan las operaciones:

1. recorrer el subárbol izquierdo en postorden

2. recorrer el subárbol derecho en postorden

3. Visitar la raíz

Al considerar el árbol binario que se muestra en la figura 28 usando cada uno de los tres criterios para recorrer el árbol se tienen las siguientes secuencias de nodos:

En preorden: h14, 4, 3, 9, 7, 5, 15, 18, 16, 17, 20i

En entreorden: h3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20i

En postorden: h3, 5, 7, 9, 4, 17, 16, 20, 18, 15, 14i

Esto nos lleva a pensar en otra aplicación, el ordenamiento de los elementos de un arreglo.

Para ordenar los elementos de un arreglo en sentido ascendente, se debe construir un árbol similar al árbol binario de búsqueda, pero sin omitir las coincidencias.

El arreglo usado para crear el árbol binario de búsqueda fue

<14,15,4,9,7,18,3,5,16,4,20,17,9,14,5>

El árbol de ordenamiento es el que se muestra en la figura 29

Para ordenar los elementos de este arreglo basta recorrer el árbol en forma de entreorden.

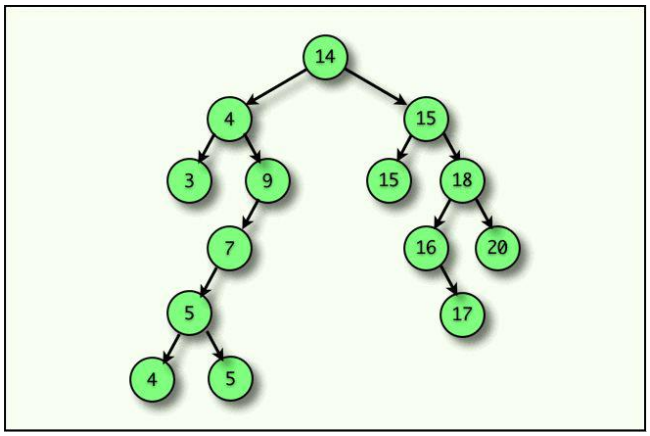



Figura. Árbol binario para ordenar una secuencia de números

¿Cuál sería el algoritmo para ordenarlo de manera descendente?

6.3. Representación en C/C++ de los árboles binarios

Vamos a estudiar estas representaciones por partes, primero los nodos y el árbol; después las operaciones para el manejo del árbol.

6.3.1. Representación de los nodos

- Los nodos de los árboles binarios son estructuras en C/C++ que están compuestas por tres partes:
- Un apuntador al subárbol izquierdo, left
- Un apuntador al subárbol derecho, right
- Una parte de información, que puede ser una estructura en sí misma, info.

Adicionalmente es muy útil poner un apuntador al padre del nodo. father.

Usando una implementación de arreglos tenemos:

```
#define numNodes 500

struct nodeType{

int info;

int left;

int right;

int father;

};

struct nodeType node[numNodes];
```

Y usando una representación con memoria dinámica, los nodos de un árbol se puede representar también con una estructura en C/C++:

```
struct nodeType{
```

```
int info;

struct nodeType *left;

struct nodeType *right;

struct nodeType *father;

};

struct nodeType *nodePtr;
```

La operaciones info(p), left(p), right(p) y father(p) se implementarían mediante referencias a p→info, p->left, p->right y p->father respectivamente.

Las rutinas getnode y freenode simplemente asignan y liberan nodos usando las rutinas malloc y free.

```
nodePtr makeTree(int x){
nodePtr p;
p = getNode();
p->info = x;
p->left = NULL;
p->right = NULL;
return p;
}
```

La rutina setLeft(p,x) establece un nodo con contenido x como el hijo izquierdo de node(p).

```
void setLeft(nodePtr p, int x){
if(p == NULL)
std::cout<<"Insercion nula\n";
else
if(p->left != NULL)
std::cout<<"Insercion no valida\n";
else
p->left=maketree(x);
}
```

La rutina para setRight(p,x) es similar a la rutina anterior.

Cuando se establece la diferencia entre los nodos de hojas y las no-hojas, los nodos que no son hojas se llaman nodos internos y los nodos que sí son hojas se llaman nodos externos.

6.3.2. Recorridos de árbol binario en C/C++

Aquí usaremos recursividad para hacer estas rutidas de los recorridos de árboles binarios. Las rutinas se llaman preTr, inTr y postTr, que imprimen el contenido de los nodos de un árbol binario en orden previo, en orden y en orden posterior, respectivamente.

El recorrido en pre orden se logra con esta rutina:

```
void preTr(nodePtr tree){
if (tree != NULL){
std::cout<<tree->info;
```

```
preTr(tree->left);
preTr(tree->right);
}
}
```

El recorrido en entre-orden se logra con esta rutina:

```
void inTr(nodePtr tree){
if (tree != NULL){
inTr(tree->left);
std::cout<<tree->info;
inTr(tree->right);
}
}
```

Y el recorrido en post-orden se logra con esta rutina:

```
void postTr(nodePtr tree){
if (tree != NULL){
postTr(tree->left);
postTr(tree->right);
std::cout<<tree->info;
}
}
```

RECOMENDACIÓN VISITA EL SIGUIENTE LINK:

http://www.el.bqto.unexpo.edu.ve/ftorres/libro_a.html

# Árboles

Hasta ahora hemos visto los árboles binarios que son aquellos árboles que sus nodos solamente pueden tener un máximo de dos hijos. Cuando ocurre que los nodos tienen cualquier número finito de hijos, son árboles (en general). De manera que:

Definición Un árbol es un conjunto finito no vacío de elementos en el cual un elemento se denomina la raíz y los restantes se dividen en m >= 0 subconjuntos disjuntos, cada uno de los cuales es por sí mismo un árbol. Cada elemento en un árbol se denomina un nodo del árbol.

Un nodo sin subárboles es una hoja. Usamos los términos padre, hijo, hermano, antecesor, descendiente, nivel y profundidad del mismo modo que en los árboles binarios. El grado de un nodo es en número máximo de hijos que algún nodo tiene.

Un árbol ordenado de define como un árbol en el que los subárboles de cada nodo forman un conjunto ordenado. En un árbol ordenado, podemos hablar del primero, segundo o ´ultimo hijo de un nodo en particular. El primer hijo de un nodo en un árbol ordenado se denomina con frecuencia el hijo más viejo de este nodo y el último se denomina el hijo más joven. Véase la figura.

Un bosque es un conjunto ordenado de árboles ordenados.

Figura. El árbol de la izquierda es ordenado y el árbol de la derecha es un árbol no ordenado.

**Representación dinámica en C de los árboles.**

Al igual que en los árboles binarios, los nodos en un árbol tienen una parte de información, un apuntador al padre y uno o más apuntadores a los hijos.

De manera que una solución es crear una estructura que incluya una lista dinámica de apuntadores, como lo muestra la figura.

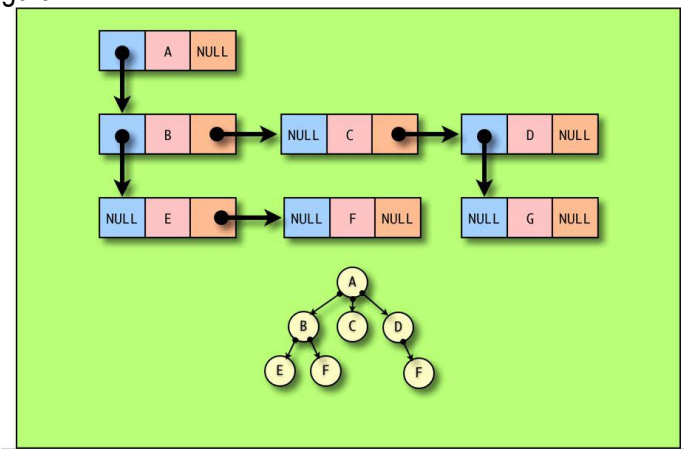



Figura. Representación con listas de los nodos de un árbol

```
struct treeNode{
int info;
struct treeNode *father;
struct treeNode *son;
struct treeNode *next;
};

typedef struct treeNode *nodePtr;
```

Si todos los recorridos se realizan de un nodo a sus hijos se omite el campo father. Incluso si es necesario acceder al padre de un nodo, el campo father se omite colocando un apuntador al padre en el campo next del hijo más joven, en lugar de dejarlo en null. Se podría usar un campo lógico adicional para indicar si el campo next apunta al siguiente hijo “real” o al padre.

Si consideramos que son corresponde al apuntador left de un nodo de ´árbol binario y que next corresponde a su apuntador right, este método representa en realidad un árbol ordenado general mediante un árbol binario.

6.4.2. Recorridos de árbol

Los métodos de recorrido para ´arboles binarios inducen métodos para recorrer los árboles en general. Si un árbol se representa como un conjunto de nodos de variables dinámicas con apuntadores son y next, una rutina en C/C++ para imprimir el contenido de sus nodos se escribiría como:

```
void inTr(nodePtr tree){
if (tree != NULL){
inTr(tree->left);
std::cout<<tree->info;
inTr(tree->right);
}
}
```

Las rutinas para recorrer el árbol en los demás órdenes son similares. Estos recorridos también se definen directamente así:

Orden previo: similar al caso binario.

1. Visitar la raíz
2. Recorrer en orden previo los subárboles de izquierda a derecha

Las demás rutinas son similares.

Un bosque puede ser representado mediante un árbol binario.

Para hacer esta representación, la raíz de cada árbol se coloca en una lista de apuntadores; luego para cada nodo en la lista (la raíz de cada árbol) se procede del siguiente modo:

1. Se crea una lista de subárboles izquierdos con los apuntadores a cada uno de los árboles en el bosque.
2. si un nodo tiene más de un hijo, entonces se crea un subárbol izquierdo y se forma una lista de subárboles izquierdos con todos los hijos de ese nodo.

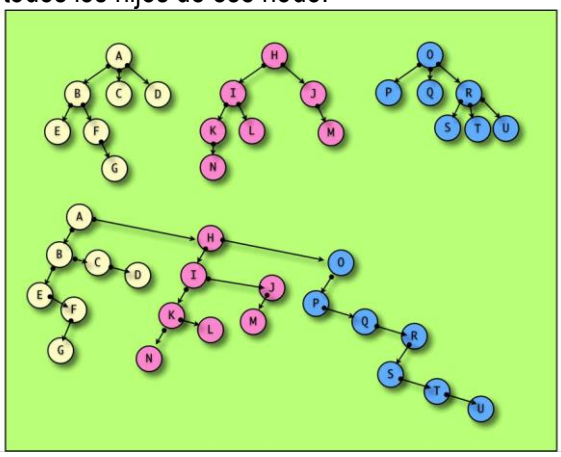



Figura. Arriba: Un bosque de árboles. Abajo: El árbol binario que corresponde a ese bosque.

Para recorrer los nodos de un bosque, es preferible convertir todo el bosque en un árbol binario correspondiente, como se ilustra en la figura 32. Cuando ya se tiene el árbol binario que corresponde a ese bosque, entonces se aplican las rutinas ya conocidas.

Si el bosque es un bosque ordenado, es decir, que todos los árboles del bosque son árboles ordenados; entonces un recorrido en entreorden dará como resultado una secuencia de nodos ordenada en sentido ascendente.