



## **Quinta Unidad**

### **Estructura de Datos No Lineales**

#### **Sumario**

Comprende y sintetiza los conceptos asociados a los árboles. Se abstrae, identifica y clasifica las características y acciones básicas (primitivas) relacionadas a los árboles y grafos.

- Árboles
- Árboles Binarios
- Árboles Binarios de Búsqueda (ABB)
- Árboles AVL y B
- Grafos



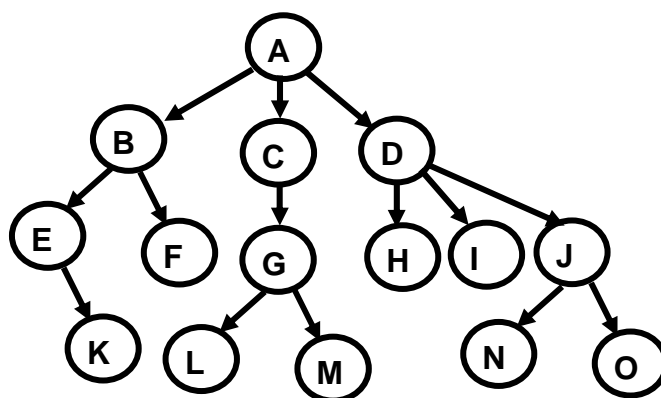
## Lección 8

### Árboles

#### 8.1. Definición

Cuando hablamos de árboles estamos hablando de estructuras que tienen una jerarquía sobre sus elementos y que no tienen una conformación lineal sino mas bien no lineal en el que cada elemento tiene enlaces o apuntadores a uno o varios elementos.

Un árbol puede representarse gráficamente como se observa en la figura siguiente:



**Figura 8.1.** Representación de un árbol

En un árbol al igual que en las listas a los elementos se les denomina nodo y existen una serie de términos y definiciones que van con los tradicionales conceptos que se usan en los arboles genealógicos que expresan las relaciones familiares:

- **Nodo** es el término usado para denotar a los elementos o vértices de un árbol.
- **Raíz** es el nodo a partir del cual se derivan o descienden otros nodos y es el único nodo que no tiene padre ya que no desciende de ningún nodo. Por ejemplo en el árbol de la figura 1, A es el nodo raíz.
- **Nodo hoja** se denomina así al nodo que no tiene hijos o que no tiene como descendiente a ningún sub árbol. Por ejemplo en el árbol de la figura 1, K, F, L, M, N, O, H y I son nodos hojas.
- **Nodos rama o internos** se refiere a los nodos que no son nodos hojas y que no son la raíz del árbol. Por ejemplo en el árbol de la figura 1, los nodos B, E, C, G, D, y J son rama
- **Nodo hijo** se refiere a un nodo que es apuntado por otro nodo del árbol. Por ejemplo en el árbol de la figura 1, E y F son nodos hijos de B y B, C y D son nodos hijos de A.
- **Nodo padre** es el nodo que tiene uno más enlaces o apuntadores hacia otros nodos. Por ejemplo en el árbol de la figura 1, A es padre de B, C y D, y D es padre de H, I, J.



- **Nodos hermanos** son los nodos que son apuntados por un mismo nodo que se conoce como nodo padre. Por ejemplo en el árbol de la figura 1, L y M son hermanos, al igual que H, I y J.
- **Bosque** se denomina a una colección o conjunto de dos o más árboles.
- **Orden** es el número potencial de hijos que puede tener cada nodo de un árbol, así tendremos árboles que serán de orden dos si cada nodo tiene dos enlaces o dos apuntadores a otros nodos y si pudiera apuntar a tres nodos entonces se llamaría árbol de orden tres., etc. Por ejemplo para el árbol de la figura 1, damos una implementación en C++ donde cada nodo puede apuntar a 4 nodos como máximo, entonces decimos que el árbol de la figura es un árbol de orden 4.

```
class ARBOL {
:
    ARBOL *rama1;
    ARBOL *rama2;
    ARBOL *rama3;
    ARBOL *rama4;
};
```

- **Grado** es el mayor número de hijos que tiene un nodo dentro del árbol. Por ejemplo en el árbol de la figura 1, el árbol es de grado 3 ya que el nodo A y el nodo D apuntan como máximo a tres nodos.
- **Nivel** es la abstracción de que un nodo corresponde a un nivel y que esta dado por la longitud de camino o la distancia del nodo específico a la raíz, medida en nodos. Por ejemplo en el árbol de la figura 1 tenemos los siguientes niveles:

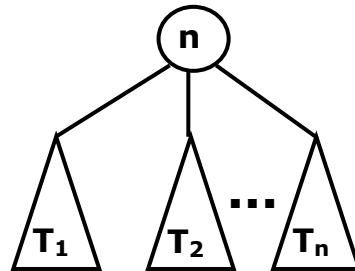
```
Nivel 0 : A
Nivel 1 : B, C, D
Nivel 2 : E, F, G, H, I, J
Nivel 3 : K, L, M, N, O
```

- **Altura** es el nivel del nodo de mayor nivel mas uno. También puede aplicarse para la altura de las ramas, considerando que cada nodo a su vez es la raíz de un árbol. El árbol del ejemplo tiene altura 4, la rama 'B' tiene altura 3, la rama 'G' tiene altura 2, la 'H' tiene altura 1, etc.

Definir un árbol es explicar una estructura recursiva, de tal modo que se suele dar una definición recursiva ya que la definición de un árbol significa referirse a otros árboles:

- Existe un nodo  $n$  o elemento que se denomina raíz del árbol.
- El resto de los nodos se distribuyen en  $m$  subconjuntos disjuntos denominados  $T_1, T_2, \dots, T_K$ , donde  $m \geq 0$ , y cada  $T_i$  es un árbol denominado sub árbol del árbol original con raíces  $n_1, n_2, \dots, n_K$ .

Se puede entonces construir un árbol con  $n$  como raíz y padre a la vez, haciendo que sus hijos sean  $n_1, n_2, \dots, n_K$ .



**Figura 8.2.** Representación de un árbol recursivo

### 8.2. Aplicaciones de árboles

Aplicaciones que usen arboles hay varias, entre ellas podemos mencionar las siguientes:

- Para describir y clasificar los objetos u elementos de datos en una estructura jerárquica para implementar bases de datos
- Su aplicación también se puede observar en el diseño de redes para interconectar multiprocesadores
- Se usa en la Teoría de Compiladores para describir el análisis del reconocimiento de una frase que corresponde a un lenguaje de programación y que es realizado por un traductor.
- Su aplicación también se da en la inteligencia artificial en lo que se denomina árboles de decisión, que vienen a ser modelos de predicción mediante el uso de diagramas que exponen una secuencia lógica de instrucciones y que sirven para graficar y representar y también categorizar una serie de hechos que se suceden de manera continua para dar solución a un problema en particular.

### 8.3. Operaciones básicas con árboles

Como en otras estructuras tenemos el mismo repertorio de operaciones de las que operábamos con las listas:

- Adicionar o insertar elementos.
- Hacer búsquedas o localizar elementos.
- Eliminar elementos.
- Moverse a través del árbol.
- Aplicar recorridos en el árbol completo.

### 8.4. Tipos de arboles

Dentro de la terminología de arboles tenemos un grupo denominado árboles ordenados y otro grupo denominado árboles desordenados. Aquí en la asignatura nos interesan los árboles ordenados por ser estos de mucha importancia como tipos abstractos de datos y a su vez por tener un mayor número de aplicaciones.



### 8.5. Tipos de arboles ordenados

- Árboles binarios de búsqueda (ABB) son arboles de orden 2 que mantienen una secuencia ordenada si se recorren en inorden.
- Árboles AVL
- Árboles B

Se distinguen básicamente los siguientes tipos de arboles ordenados:

- **Árboles binarios de búsqueda(ABB):** es un árbol binario en el cual la condición es que el subárbol izquierdo de cualquier nodo (si no está vacío) registra valores que son menores que los que registra el que contiene dicho nodo, y el subárbol derecho (si no está vacío) registra valores mayores que los que contiene dicho nodo y que observa un orden si se recorre en inorden.
- **Árboles AVL:** un árbol AVL es un árbol binario que esta constantemente equilibrado, lo que quiere decir que para todos los nodos, la altura de la ramas izquierda y derecha podrán diferir como máximo en una unidad.
- **Árbol-B:** un árbol B es un árbol que nunca es binario y es para resolver varios de los inconvenientes que ocurren con los arboles binarios comunes o balanceados.

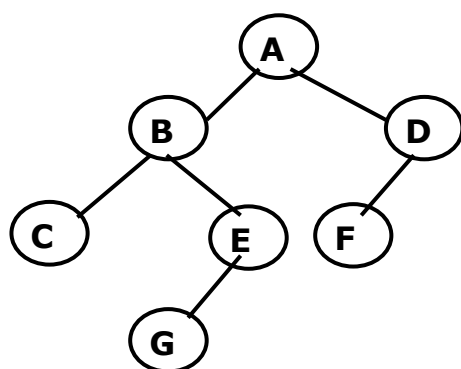
### 8.6. Implementaciones

Los arboles pueden implementarse de dos formas básicamente:

- Usando arreglos
- Usando asignación dinámica

#### 8.6.1. Usando arreglos

Cuando se usa arreglos significa implementar el árbol en una estructura estática en el cual cada nodo es un objeto con cuatro campos. Uno de los campos es para la información, dos campos son para referencia y un campo mas para indicar el estado de cada objeto dentro del arreglo, verdadero para indicar si esa posición esta siendo ocupada y falso si esta libre. Este tipo de implementación puede verse en el siguiente ejemplo:



0  
1  
2  
3  
4  
5  
6  
7  
8  
9

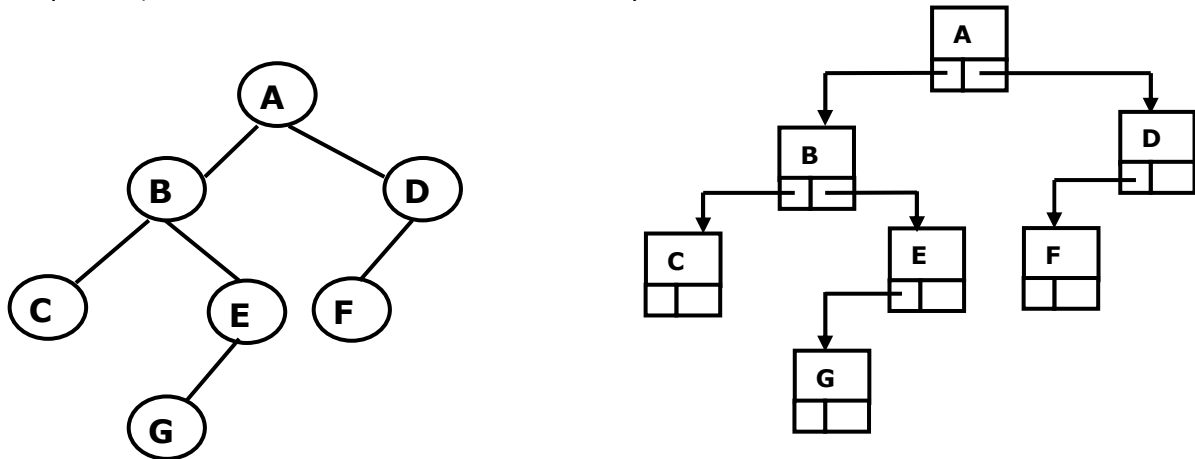
Info	Referencia		Cond
	Izq	Der	
A	1	5	V
B	2	3	V
C	0	0	V
E	4	0	V
G	0	0	V
D	6	0	V
F	0	0	V
			F
			F
			F

**Figura 8.3.** Representación de un árbol usando arreglos



### 8.6.2. Usando asignación dinámica

Cuando se usa asignación dinámica para implementar un árbol, cada nodo del árbol tendrá apuntadores hacia sus nodos hijos. "Se necesita una versión extendida de este concepto, una lista doblemente enlazada, donde cada celda contenga dos apuntadores y un elemento de datos." (Kolman, 1995, 295). En la figura 9.4. se puede ver una grafica a la derecha que explica como en el árbol que esta a la izquierda, sus nodos se enlazan a través de apuntadores.



**Figura 8.4.** Representación de un árbol usando asignación dinámica

La implementación mediante arreglos puede no ser muy adecuada para trabajar con árboles, aun cuando trabajemos con un arreglo flexible al usar un arreglo o una lista de arreglos. Sin embargo el usar una estructura dinámica trae las siguientes ventajas

- Se tiene un programa mas flexible
- Es mas fácil de utilizar
- Hay mayor capacidad de información y eficiencia
- Mayor potencia

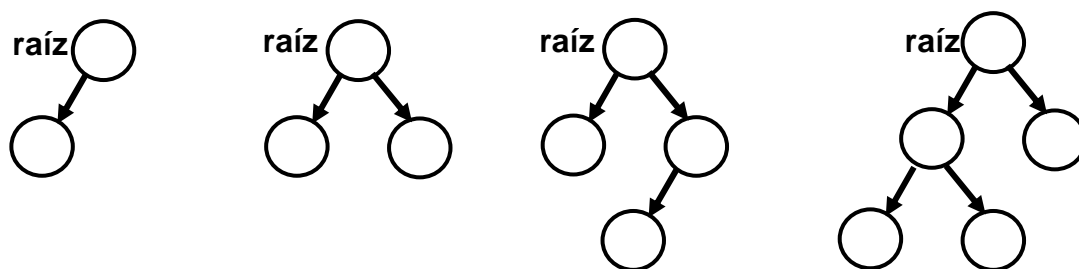
Como desventaja podríamos mencionar las siguientes:

- complejidad que requiere el implementarla
- El uso de la memoria dinámica resta recursos al computador
- El uso incorrecto de la memoria podría bloquear o desconfigurar el software instalado en el computador

### 8.7. Árboles Binarios

"Un árbol binario es un conjunto finito de elementos que o esta vacio o esta dividido en tres subconjuntos desarticulados. El primer subconjunto contiene un solo elemento llamado raíz del árbol. Los otros dos son en si mismos arboles binarios, llamados subarboles izquierdo y derecho del árbol original. Un subárbol izquierdo o derecho puede estar vacio." (Tenenbaum, 1993, p. 241).

### Ejemplo de arboles binarios



**Figura 8.5.** Árboles binarios

### Ejemplo de arboles que no son binarios



**Figura 8.6.** Árboles no binarios

“Un árbol binario es un conjunto de  $N$  registros ( $N \geq 0$ ), el cual puede ser vacío o constar de una raíz y los demás registros particionados en dos conjuntos dispuestos, cada una de las cuales es un árbol binario (definición recursiva), que se conocen como sub-árbol izquierdo y sub-árbol derecho.” (Florez, 2005, 235).

#### 8.7.1. Recorridos en los Árboles Binarios

Para moverse dentro de la estructura de un árbol con la finalidad de buscar, eliminar o modificar un nodo en particular, etc, es obligatorio recorrer la estructura y esto puede hacerse de dos maneras, una se conoce como recorrido en profundidad y la otra se conoce como recorrido en amplitud o también denominado en anchura o por nivel. Como los árboles no tienen una estructura lineal como las listas, es necesario crear algoritmos alternativos que nos permitan visitar cada uno de los nodos del árbol.

#### 8.7.2. Recorridos en Profundidad

Los recorridos en profundidad pueden hacerse de tres formas y es común implementarlas usando técnicas recursivas.

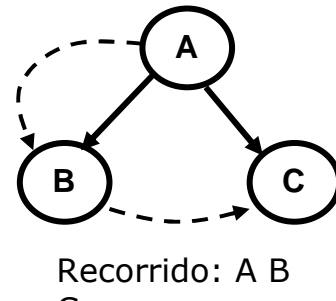


### a) Recorrido en pre-orden

Cuando se hace un recorrido en pre-orden se comienza siempre visitando el nodo raíz del árbol (al decir visitar nos referimos a ubicarnos en un nodo para visualizar su contenido, modificarlo o eliminarlo, etc), el segundo paso consiste en moverse al subárbol izquierdo y como tercer paso moverse al subárbol derecho. Como mencionamos anteriormente es de uso común usar la recursividad para crear el recorrido en preorden.

```

Procedimiento preorden(raiz)
  Si (raiz ≠ nulo) entonces
    visitar(a)
    preorden(izq(raiz))
    preorden(der(raiz))
  Fin_si
Fin-procedimiento
  
```

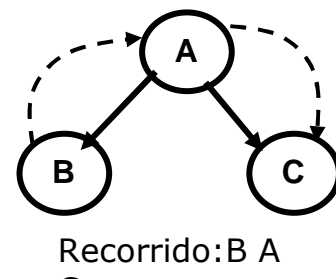


### b) Recorrido en in-orden

Cuando se hace un recorrido en in-orden se comienza siempre por el subárbol izquierdo, luego se visita el nodo raíz del árbol y como último paso nos movemos al subárbol derecho.

```

Procedimiento inorden(raiz)
  Si (raiz ≠ nulo)
    inorden(izq(raiz))
    visitar(raiz)
    inorden(der(raiz))
  Fin_si
Fin-procedimiento
  
```

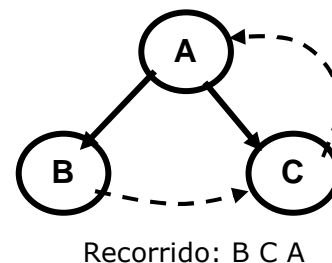


### c) Recorrido en post-orden

Cuando se hace un recorrido en post-orden se comienza siempre por el subárbol izquierdo, luego nos movemos al subárbol derecho y finalmente se visita el nodo raíz del árbol.

```

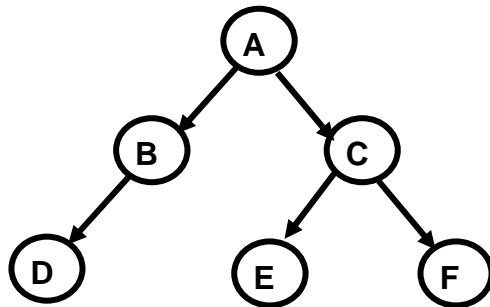
Procedimiento postorden(raiz)
  Si (raiz ≠ nulo) entonces
    postorden(izq(raiz))
    postorden(der(raiz))
    visitar(raiz)
  Fin_si
Fin-procedimiento
  
```







Para el siguiente árbol efectuamos los siguientes recorridos:

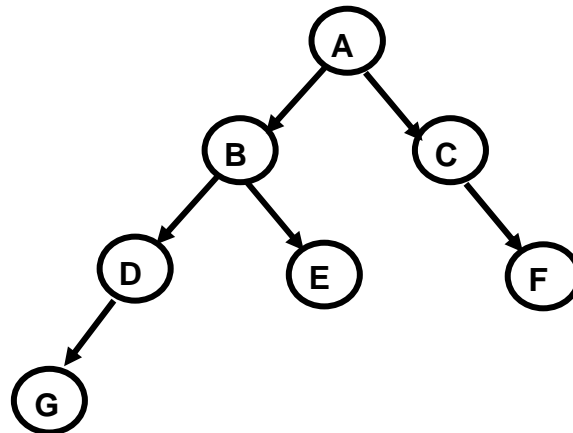


Pre-orden: A B D C E F  
 In-orden : D B A E C F  
 Post-orden : D B E F C A

### 8.7.3. Recorrido en amplitud

Es un recorrido en el cual la visita de cada nodo se hace por niveles. Se comienza por el nivel 1, ósea la raíz del árbol, luego se pasa por los nodos del nivel 2, después por los nodos del nivel 3 y de modo análogo se procede con los demás niveles hasta que ya no queden mas nodos por recorrer.

Por ejemplo para el siguiente árbol:



**Figura 9.7.** Recorrido por amplitud :A, B, C, D, E, F, G

Para hacer posible el recorrido por amplitud se hace utilizando no formas recursivas sino iterativas. Para ello se hace uso de una cola que hará de estructura auxiliar en el proceso de recorrido. La cola sirve para guardar (siempre y cuando tengan al menos un nodo) los sub-arboles izquierdo y derecho del nodo que es retirado de la cola y seguir este mismo proceso de guardar en la cola y retirar de la cola hasta que la cola ya no tenga ningún nodo.

A continuación el algoritmo para el recorrido en amplitud:

1. Procedimiento amplitud(raiz, cola)
2. Si (raiz ≠ nulo) entonces
3.     cola.METER(raíz, dato)
4.     Mientras (no cola.VACIA()) Hacer
5.         dato ← cola. RETORNARCIMA( )

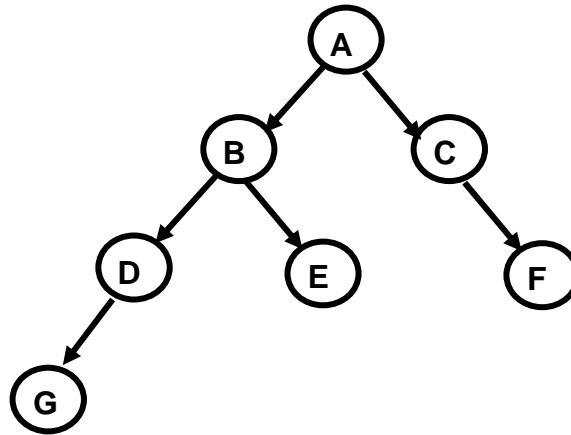


```

6.          cola.SACAR( )
7.          VISITAR(dato)
8.          Si (izq(dato) ≠ nulo) entonces
9.              cola.METER(izq(dato))
10.         Fin_si
11.         Si (der(auxiliar) ≠ nulo) entonces
12.             cola.METER(der(dato))
13.         Fin_si
14.     Fin_mientras
15. Fin_si
16. Fin-procedimiento

```

Seguidamente un árbol y un cuadro debajo mostrando la secuencia de operaciones para efectuar el recorrido por amplitud.



iterar	OPERACIONES								COLA
	3	4	6	7	8	9	11	12	
1	A	true	A	A	true	pone B	true	pone C	B C
2		true	B	B	true	pone D	true	pone E	C D E
3		true	C	C	false		true	pone F	D E F
4		true	D	D	true	pone G	false		EFG
5		true	E	E	false		false		FG
6		true	F	F	false		false		G
7		true	G	G	false		false		

**Tabla 8.1.** Secuencia de operaciones para efectuar el recorrido por amplitud

El rectángulo vertical sombreado en sus bordes dentro del cuadro muestra como se van visitando los nodos en un recorrido por amplitud

#### 8.7.4. Construcción de árboles binarios

Primero hacemos la especificación del TAD árbol del siguiente modo:



## Especificación ÁRBOL

### variable

numero : entero  
 izq, der : ARBOL  
 sw : entero  
 j : entero  
 c[20] : entero

### métodos

ARBOL : no retorna nada  
 menu() : no retorna valor  
 ramas(p) : no retorna valor  
 mostrarArbol(p) : no retorna valor  
 buscarNodo(raíz, dato) : retorna un tipo ARBOL  
 adicionarNodo(raíz, x) : no retorna nada

### significado

ARBOL inicializa el valor de j igual a cero  
 menú muestra las opciones a escoger  
 ramas(p) visualiza el árbol por ramas  
 mostrarArbol(p) invoca a ramas para visualizar el árbol  
 buscarNodo(raíz, dato) busca dato en ARBOL con inicio en raíz  
 adicionarNodo(raíz, x) agrega un nodo x en el ARBOL

## Fin\_especificación

Funcion buscarNodo(raiz, dato)

Si(raiz!= NULL) entonces  
 leer(raíz, numero)  
 Si(dato=numero) entonces  
 retornar raiz  
 Fin\_si  
 p=buscarNodo(izq(raíz),dato)  
 Si(p!=NULL) entonces  
 retornar p  
 Fin\_si  
 p=buscarNodo(der(raíz),dato)  
 Si(p!=NULL) entonces  
 retornar p  
 Fin\_si  
 retornar NULL;  
 Fin\_si  
 retornar NULL;

Fin\_funcion

Procedimiento adicionarNodo(raiz, x)

// definir variables  
 ARBOL : p,n  
 Entero : y  
 Si(raiz=NULL) entonces



```

        crear(raiz)
        izq(raiz)←NULL
        der(raíz)←NULL
        Asignar(raíz, x)
    Sino
        Escribir (" ingrese un numero existente con el cual
enlazar: ")
        leer y
        p←buscarNodo(raíz, y)
        Si (p!=NULL) entonces
            Si(izq(p)=NULL)
                crear(n)
                izq(n)←NULL
                der(n)←NULL
                Asignar( n,x)
                Izq( p)←n
            Sino
                Si (der(p)=NULL) entonces
                    crear(n)
                    izq( n)←NULL
                    der(n)←NULL
                    Asignar(n,x )
                    der(p)←n
                sino
                    escribir " no hay enlace libre para
insertar"
                    Fin_si
                Fin_si
            Sino
                escribir "el elemento elegido no existe"
            Fin_si
        Fin_si
    Fin_adicionarNodo

```

```

Procedimiento ramas(p){
    // definir variables
    ARBOL : p1, p2
    Si(sw=0) entonces
        leer(p, numero)
        c[j]←numero
        j←j+1
        sw←1
    Fin_si
    Si(p!=NULL) entonces
        p1←izq(p)
        p2←der(p)
        si (p1=NULL) y p2=NULL) entonces

```



```

        Desde k ← 0 Hasta k < j con incremento 1 Hacer
            Escribir c[k]
        Fin_desde
    Fin_si
    Si (p1!=NULL) entonces
        leer(p1,numero)
        c[j]←numero
        j←j+1
        ramas(p1)
    Fin_si
    Si (p2!=NULL) entonces
        leer(p2, numero)
        c[j]←numero;
        j←j+1
        ramas(p2)
    Fin_si
    j←j-1
Fin_si
Fin_procedimiento

```

Luego la Implementación del TAD en C++

```

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
class ARBOL{
    int numero;
    ARBOL *izq;
    ARBOL *der;
    int sw;
    int c[20];
    int j;
public:
    ARBOL(){ j=0;}
    void menu(){
        cout<< "\nMENU DE OPCIONES\n";
        cout<< "-----\n" ;
        cout<<"<1> Ingresar    \n";
        cout<<"<2> Mostrar     \n";
        cout<<"<3> Salir       \n";
    }
    void ramas(ARBOL *p){
        ARBOL *p1,*p2;
        int l1,l2;
        if(sw==0) {
            c[j++]=p->numero;
            sw=1;
        }
    }
};

```



```

    }
    if(p!=NULL){
        p1=p->izq;    p2=p->der;
        if((p1==NULL) && (p2==NULL)) {
            for(int k=0; k<j;k++)
                cout<<c[k]<<" ";
            cout<<"\n";
        }
        if(p1!=NULL){
            c[j++]=p1->numero;
            ramas(p1);
        }
        if(p2!=NULL){
            c[j++]=p2->numero;
            ramas(p2);
        }
        j--;
    }
}

void mostrarArbol(ARBOL *&p){
    sw=0;
    if(p!=NULL) { cout<<"\n\n";ramas(p); }
    else cout<<"\n\n El arbol esta en cero "<<endl;
}

ARBOL *buscarNodo(ARBOL *raiz, int dato){
    ARBOL *p;
    if(raiz){
        if(dato==raiz->numero) return raiz;
        if((p=buscarNodo(raiz->izq,dato))!=NULL) return p;
        if((p=buscarNodo(raiz->der,dato))!=NULL) return p;
        return NULL;
    }
    return NULL;
}

void adicionarNodo(ARBOL *&raiz, int x){
    ARBOL *p,*n; int y;
    if(raiz==NULL){
        raiz=new ARBOL;
        raiz->izq=NULL;
        raiz->der=NULL;
        raiz->numero=x;
    } else{
        cout<<"\n ingrese un numero existente con el cual
        enlazar: ";
        cin>>y;
        if((p=buscarNodo(raiz, y))!=NULL){
            if(p->izq==NULL){
                n=new ARBOL;

```



```

        n->izq=NULL;
        n->der=NULL;
        n->numero=x;
        p->izq=n;
    }
    else if(p->der==NULL){
        n=new ARBOL;
        n->izq=NULL;
        n->der=NULL;
        n->numero=x;
        p->der=n;
    }else cout<<"\n no hay enlace libre para insertar";
}
else cout<<"\n el elemento elegido no existe";
}
}
};
int main(){
    ARBOL arb;
    ARBOL *raiz=NULL;
    int valor;
    char opcion;
    do
    {
        arb.menu();
        cout<<"\ningrese opcion : ";
        opcion=cin.get();
        switch(opcion){
            case '1':cout<<"\n Ingrese un numero entero al
arbol: ";
                        cin>>valor;
                        arb.adicionarNodo(raiz,valor);break;
            case '2':arb.mostrarArbol(raiz);break;
        }
        cin.ignore();

        while( opcion !='3');
        system("pause");
        return 0;
    }
}

```

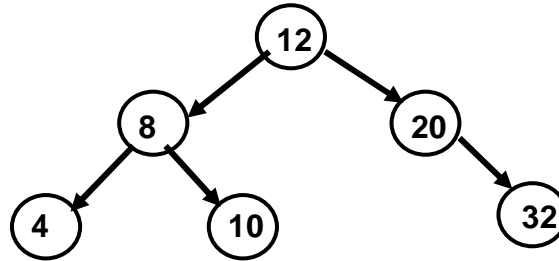
### 8.8. Árboles Binarios de Búsqueda (ABB)

Se define un árbol binario de búsqueda como una estructura en el cual la condición para todos los nodos del árbol es como sigue. Para un nodo si su hijo izquierdo existe, contiene un valor menor que el valor que contiene el nodo padre y si su hijo derecho existe



contendrá un valor en este caso mayor al que contiene su nodo padre.

Es claro que para fijar un orden entre los nodos del árbol, cada nodo debe contener un valor numérico y si se trata de un tipo compuesto, este debe involucrar una clave o atributo numérico para poder hacer posible la ordenación. A continuación se puede observar un árbol binario de búsqueda de números enteros.



**Figura 8.8.** Árboles Binarios de Búsqueda (ABB)

#### 8.8.1. Búsqueda de un elemento

El proceso de búsqueda consiste en iniciar el recorrido en el nodo raíz e ir visitando todos los nodos posibles hasta llegar al nodo buscado usando el siguiente algoritmo. Cada nodo que se visite o que se conoce también como nodo actual incluyendo el nodo raíz se examina y se procede del siguiente modo:

1. Si el valor buscado resulta igual al valor del nodo actual, el algoritmo se detiene y se da por aceptado que el valor buscado existe en el árbol.
2. En caso que el valor buscado sea menor que el valor del nodo actual, la búsqueda proseguirá por el subárbol izquierdo.
3. En caso que el valor buscado sea mayor que el valor del nodo actual, la búsqueda proseguirá por el subárbol derecho.

#### 8.8.2. Inserción de un elemento

Cuando se procede a insertar un nodo en el árbol, el algoritmo realiza los siguientes pasos:

1. El primer paso consiste en gestionar memoria para crear el nuevo nodo que se va a insertar y que contendrá un valor determinado.
2. Como segundo paso se realiza el proceso de búsqueda para encontrar la posición adecuada en el árbol para el nuevo nodo.
3. Como último paso se inserta en el árbol en la posición hallada.





### 8.8.3. Construcción de un Árbol Binario de Búsqueda(ABB)

A continuación describimos el TAD donde cada nodo tendrá como información un numero entero y asociado a este valor sus operaciones. Una de las operaciones es para insertar nodos y otras tres son para recorridos en profundidad.

*Especificación de los TAD's y los algoritmos*

#### Especificación ABB

##### variable

numero : entero  
 izq, der : ABB

##### métodos

menú() : no retorna valor  
 VACIO(raíz) : retorna valor entero  
 INSERTAR(raíz, dato) : no retorna valor  
 INORDEN(raíz) : no retorna valor  
 PREORDEN(raíz) : no retorna valor  
 POSTORDEN(raíz): no retorna nada

##### Significado

*VACIO* retorna 1 si es el árbol no tiene ningún nodo, de lo contrario retorna 0  
*INSERTAR* inserta dato en el árbol  
*INORDEN* visualiza el valor de los nodos del árbol  
*PREORDEN* visualiza el valor de los nodos del árbol  
*POSTORDEN* visualiza el valor de los nodos del árbol

#### Fin\_especificación

Funcion VACIO(raíz)

Si raíz=nulo entonces  
 retornar verdadero

Sino  
 retornar falso

Fin\_si

Fin\_funcion

Procedimiento INSERTAR(raíz, dato )

// Definir variables

ABB : padre, actual

padre ← nulo

actual ← raíz

leer(actual, numero)

Mientras (no VACIO(actual) y dato ≠ numero) hacer

padre ← actual

leer(actual, numero)

Si (dato < numero) entonces

actual ← izq(actual)



```

        Sino
            Si (dato > numero) entonces
                actual ← der(actual)
            Fin_si
        Fin_si
    Fin_mientras
    Si ( Vacio(actual)) entonces
        Si (Vacio(padre)) entonces
            crear(raíz)
            asignar(raíz, dato)
            izq(raíz) ← nulo
            der(raíz) ← nulo
        Sino
            leer(padre, numero)
            Si (dato < numero) entonces
                crear(actual)
                izq(padre) ← actual
                asignar(actual, dato)
                izq(actual) ← nulo
                der(actual) ← nulo
            Sino
                Si (dato > numero) entonces
                    crear(actual)
                    der(padre) <- actual
                    asignar(actual, dato )
                    izq(actual) <- nulo
                    der(actual) <- nulo
                Fin_si
            Fin_si
        Fin_si
    Fin_si
Fin_procedimiento

Procedimiento INORDEN(raíz)
    Si (a ≠ nulo) entonces
        Si ( izq(a) ≠ nulo) entonces
            INORDEN(izq(a))
        Fin_si
        Leer(a, dato)
        escribir (dato)
        Si ( der(a) ≠ nulo ) entonces
            InOrden(der(a))
        Fin_si
    Fin_si
Fin_procedimiento

Procedimiento PREORDEN(raiz)
    Si (a ≠ nulo) entonces

```



```

        Leer(a, dato)
        visualizar (dato)
        Si (izq(a) ≠ nulo ) entonces
            PreOrden(izq(a))
        Fin_si
        Si(der(a) ≠ nulo) entonces
            PreOrden(der(a))
        Fin_si
    Fin_si
Fin_procedimiento

```

```

Procedimiento POSTORDEN(raiz)
    Si(a ≠ nulo) entonces
        Si( izq(a) ≠ nulo) entonces
            PostOrden(izq(a))
        Fin_si
        Si(der(a)) entonces
            PostOrden(der(a))
        Fin_si
        Leer(a, dato)
        visualizar (dato)
    Fin_si
Fin_procedimiento

```

#### *Implementación del TAD*

```

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
class ABB{
    int numero;
    ABB *izq,*der;
public:
    void menu(){
        cout<< "\nMENU DE OPCIONES\n";
        cout<< "\n insertar      <1>";
        cout<< "\n Inorden      <2>";
        cout<< "\n Preorden     <3>";
        cout<< "\n Postorden    <4>";
        cout<< "\n Salir        <5>";
    }
    int VACIO(ABB *raiz){ return raiz==NULL; }
    void INSERTAR(ABB *&raiz, int dato){
        ABB *padre = NULL;
        ABB *actual = raiz;
        while(!VACIO(actual) && dato != actual->numero) {
            padre = actual;

```



```

        if(dato < actual->numero) actual = actual-
        >izq;
        else if(dato > actual->numero) actual =
        actual->der;
    }
    if(!VACIO(actual)) return;
    if(VACIO(padre)) {
        raiz = new ABB;
        raiz->numero= dato;
        raiz->izq = raiz->der = NULL;
    }
    else if(dato < padre->numero) {
        actual = new ABB;
        padre->izq = actual;
        actual->numero = dato;
        actual->izq = actual->der = NULL;
    }
    else if(dato > padre->numero) {
        actual = new ABB;
        padre->der = actual;
        actual->numero = dato;
        actual->izq = actual->der = NULL;
    }
}
void INORDEN(ABB *raiz){
    if(raiz){
        INORDEN(raiz->izq);
        cout<<" "<<raiz->numero;
        INORDEN(raiz->der);
    }
}
void PREORDEN(ABB *raiz){
    if(raiz){
        cout<<" "<<raiz->numero;
        PREORDEN(raiz->izq);
        PREORDEN(raiz->der);
    }
}
void POSTORDEN(ABB *raiz){
    if (raiz){
        POSTORDEN(raiz->izq);
        POSTORDEN(raiz->der);
        cout<<" "<<raiz->numero;
    }
}
};
int main(){
    ABB arb;

```



```

ABB *raiz=NULL;
int num;
char opcion;
do
{
    arb.menu();
    cout<<"\ningrese opcion : ";
    opcion=cin.get();
    switch(opcion){
        case '1':cout<<"\n Ingrese un numero entero: ";cin>>num;
                    arb.INSERTAR(raiz, num); break;
        case '2':cout<<"InOrden: ";
                    arb.INORDEN(raiz); break;
        case '3':cout<<"PreOrden: ";
                    arb.PREORDEN(raiz); break;
        case '4':cout<<"PostOrden: ";
                    arb.POSTORDEN(raiz); break;
    }
    cin.ignore();
}
while( opcion !='5');
system("pause");
return 0;
}

```

## 8.9. Ejercicios resueltos

### Ejemplo 01

Escriba la especificación del TAD, el algoritmo y la implementación en C++ para un árbol binario de números enteros en el cual se desea eliminar a los nodos hojas.

*Solución:*

*Especificación de los TAD's y los algoritmos*

En esta parte la especificación es muy parecida al de la sección 9.6.4 con la diferencia que se adiciona el método eliminarHoja el cual elimina a todos los nodos hoja del árbol:

```

Funcion eliminarHoja(a)
    entero : sw←0
    Si (a≠nulo) entonces
        Si (izq(a)≠nulo) entonces
            Si (no eliminarHoja(izq(a))) entonces
                izq(a)←nulo
        Fin_si

```



```

        sw ← 1
    Fin_si
    Si (der(a) ≠ nulo) entonces
        Si (no eliminarHoja(der(a))) entonces
            der(a) ← nulo
        Fin_si
        sw ← 1
    Fin_si
    Si ( sw ≠ 1) entonces
        liberar(a)
        retornar nulo
    Sino
        retornar a
    Fin_si
Sino
    retornar nulo
Fin_si
Fin_función

```

#### Implementación del TAD

De manera similar a la especificación, en el programa de la sección 9.6.4 adicione la función eliminarHoja que se observa abajo:

```

ARBOL *eliminarHoja(ARBOL *a){
    int sw=0;
    if(a){
        if(a->izq) {
            if(!eliminarHoja(a->izq)) a->izq=NULL;
            sw=1;
        }
        if(a->der){
            if(!eliminarHoja(a->der)) a->der=NULL;
            sw=1;
        }
        if( sw!=1) {
            delete a;
            return NULL;
        }
        else return a;
    }
    else return NULL;
}

```

luego modificar el código del main para que quede como se observa a continuación:



```

int main(){
    ARBOL arb;
    ARBOL *raiz=NULL;
    int valor;
    char opcion;
    do
    {
        arb.menu();
        cout<<"\ningrese opcion : ";
        opcion=cin.get();
        switch(opcion){
            case '1':cout<<"\n Ingrese un numero entero al
arbol: ";
                                cin>>valor;
                                arb.adicionarNodo(raiz,valor);break;
            case '2':arb.mostrarArbol(raiz);break;
            case '3':arb.eliminarHoja(raiz); break;
        }
        cin.ignore();
    }
    while( opcion !='4');
    system("pause");
    return 0;
}

```

## Ejemplo 02

Escriba la especificación del TAD, el algoritmo y la implementación en C++ para un árbol binario de números enteros en el cual se desea encontrar el número de nodos hojas.

*Solución:*

*Especificación de los TAD's y los algoritmos*

En esta parte la especificación es muy parecida al de la sección 9.6.4 con la diferencia que se adiciona el atributo cantidad de tipo entero y se adiciona el método inicializar que pone cantidad igual a cero y cantidadDeHojas que retorna la cantidad de nodos hojas que hay en el arbol:

```

Funcion cantidadDeHojas(a): entero
    entero : sw←0
    Si (a≠nulo) entonces
        Si (izq(a)≠nulo) entonces
            cantidadDeHojas(izq(a))
            sw←1
        Fin_si
    Si (der(a)≠nulo) entonces

```



```

                cantidadDeHojas(der(a))
                sw←1
            Fin_si
            Si (sw ≠1) entonces
                cantidad←cantidad+1
            Sino
                retornar cantidad
            Fin_si
        Fin_si
    Fin_funcion

```

### Implementación del TAD

```

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
class ARBOL{
    int numero;
    ARBOL *izq;
    ARBOL *der;
    int sw;
    int c[20];
    int j;
    int cantidad;
public:
    ARBOL(){ j=0; cantidad=0; }
    void inicializar(){ cantidad=0; }
    void menu(){
        cout<< "\nMENU DE OPCIONES\n";
        cout<< "-----\n" ;
        cout<<"<1> Ingresar          \n";
        cout<<"<2> Mostrar           \n";
        cout<<"<3> Numero de nodos hojas \n";
        cout<<"<4> Salir             \n";
    }
    void ramas(ARBOL *p){
        ARBOL *p1,*p2;
        int l1,l2;
        if(sw==0) {
            c[j++]=p->numero;
            sw=1;
        }
        if(p!=NULL){
            p1=p->izq;    p2=p->der;
            if((p1==NULL) && (p2==NULL)) {
                for(int k=0; k<j;k++)
                    cout<<c[k]<<" ";
            }
        }
    }
};

```





```

        cout<<"\n";
    }
    if(p1!=NULL){
        c[j++]=p1->numero;
        ramas(p1);
    }
    if(p2!=NULL){
        c[j++]=p2->numero;
        ramas(p2);
    }
    j--;
}

}

void mostrarArbol(ARBOL *&p){
    sw=0;
    if(p!=NULL) { cout<<"\n\n";ramas(p); }
    else cout<<"\n El arbol esta en cero \n";
}

ARBOL *buscarNodo(ARBOL *raiz, int dato){
    ARBOL *p;
    if(raiz){
        if(dato==raiz->numero) return raiz;
        if((p=buscarNodo(raiz->izq,dato))!=NULL) return p;
        if((p=buscarNodo(raiz->der,dato))!=NULL) return
        p;
        return NULL;
    }
    return NULL;
}

void adicionarNodo(ARBOL *&raiz, int x){
    ARBOL *p,*n; int y;
    if(raiz==NULL){
        raiz=new ARBOL;
        raiz->izq=NULL;
        raiz->der=NULL;
        raiz->numero=x;
    } else{
        cout<<"\n ingrese un numero existente con el cual
        enlazar: ";
        cin>>y;
        if((p=buscarNodo(raiz, y))!=NULL){
            if(p->izq==NULL){
                n=new ARBOL;
                n->izq=NULL;
                n->der=NULL;
                n->numero=x;
                p->izq=n;
            }
        }
    }
}

```



```

        else if(p->der==NULL){
            n=new ARBOL;
            n->izq=NULL;
            n->der=NULL;
            n->numero=x;
            p->der=n;
        }else cout<<"\n no hay enlace libre para
insertar";
    }
    else cout<<"\n el elemento elegido no existe";
}
}
int cantidadDeHojas(ARBOL *a){
    int sw=0;
    if(a){
        if(a->izq){
            cantidadDeHojas(a->izq); sw=1;
        }
        if(a->der){
            cantidadDeHojas(a->der); sw=1;
        }
        if(sw!=1) cantidad++;
        else return cantidad;
    }
    else cantidad;
}
};
int main(){
    ARBOL arb;
    ARBOL *raiz=NULL;
    int valor;
    char opcion;
    do
    {
        arb.menu();
        cout<<"\ningrese opcion : ";
        opcion=cin.get();
        switch(opcion){
            case '1':cout<<"\n Ingrese un numero entero al
arbol: ";
                cin>>valor;
                arb.adicionarNodo(raiz,valor);break;
            case '2':arb.mostrarArbol(raiz);break;
            case '3':arb.inicializar();
                cout<<" # nodos hojas= "
                <<arb.cantidadDeHojas(raiz); break;
        }
        cin.ignore();
    }

```



```

    }
    while( opcion != '4');
    system("pause");
    return 0;
}

```

### Ejemplo 03

Escriba la especificación del TAD, el algoritmo y la implementación en C++ para un árbol binario de números enteros en el cual se desea encontrar a todos los nodos rama.

*Solución:*

*Especificación de los TAD's y los algoritmos.*

En esta parte la especificación es muy parecida al del ejemplo 02 con la diferencia que no es necesario contar con el método cantidadDeHojas, sino con el método cantidadDeRamas que retorna el número de nodos rama del árbol y que se observa a continuación:

```

Funcion cantidadDeRamas(a) : entero
    Si(a≠nulo) entonces
        Si (izq(a)) entonces
            cantidadDeRamas(izq(a))
        Fin_si
        Si (der(a)) entonces
            cantidadDeRamas(der(a))
        Fin_si
        Si(izq(a)≠nulo || der(a)≠nulo) entonces
            cantidad←cantidad+1
        Fin_si
    Fin_si
    retornar cantidad
Fin_funcion

```

*Implementación del TAD*

```

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
class ARBOL{
    int numero;
    ARBOL *izq;
    ARBOL *der;
    int sw;

```



```

int c[20];
int j;
int cantidad;
public:
ARBOL(){ j=0; cantidad=-1; }
void inicializar(){ cantidad=-1; }
void menu(){
    cout<< "\nMENU DE OPCIONES\n";
    cout<< "-----\n" ;
    cout<<"<1> Ingresar      \n";
    cout<<"<2> Mostrar      \n";
    cout<<"<3> Numero de nodos rama \n";
    cout<<"<4> Salir        \n";
}
void ramas(ARBOL *p){
    ARBOL *p1,*p2;
    int l1,l2;
    if(sw==0) {
        c[j++]=p->numero;
        sw=1;
    }
    if(p!=NULL){
        p1=p->izq;    p2=p->der;
        if((p1==NULL) && (p2==NULL)) {
            for(int k=0; k<j;k++)
                cout<<c[k]<<" ";
            cout<<"\n";
        }
        if(p1!=NULL){
            c[j++]=p1->numero;
            ramas(p1);
        }
        if(p2!=NULL){
            c[j++]=p2->numero;
            ramas(p2);
        }
        j--;
    }
}
void mostrarArbol(ARBOL *&p){
    sw=0;
    if(p!=NULL) { cout<<"\n\n";ramas(p); }
    else cout<<"\n El arbol esta en cero \n";
}
ARBOL *buscarNodo(ARBOL *raiz, int dato){
    ARBOL *p;
    if(raiz){
        if(dato==raiz->numero) return raiz;
    }
}

```



```

        if((p=buscarNodo(raiz->izq,dato))!=NULL) return p;
        if((p=buscarNodo(raiz->der,dato))!=NULL) return
p;
        return NULL;
    }
    return NULL;
}
void adicionarNodo(ARBOL *&raiz, int x){
    ARBOL *p,*n; int y;
    if(raiz==NULL){
        raiz=new ARBOL;
        raiz->izq=NULL;
        raiz->der=NULL;
        raiz->numero=x;
    } else{
        cout<<"\n ingrese un numero existente con el cual
enlazar: ";
        cin>>y;
        if((p=buscarNodo(raiz, y))!=NULL){
            if(p->izq==NULL){
                n=new ARBOL;
                n->izq=NULL;
                n->der=NULL;
                n->numero=x;
                p->izq=n;
            }
            else if(p->der==NULL){
                n=new ARBOL;
                n->izq=NULL;
                n->der=NULL;
                n->numero=x;
                p->der=n;
            }else cout<<"\n no hay enlace libre para insertar";
        }
        else cout<<"\n el elemento elegido no existe";
    }
}
int cantidadDeRamas(ARBOL *a){
    if(a){
        if(a->izq)
            cantidadDeRamas(a->izq);
        if(a->der)
            cantidadDeRamas(a->der);
        if(a->izq || a->der) cantidad++;
    }
    return cantidad;
}
};

```



```

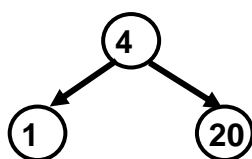
int main(){
    ARBOL arb;
    ARBOL *raiz=NULL;
    int valor;
    char opcion;
    do
    {
        arb.menu();
        cout<<"\ningrese opcion : ";
        opcion=cin.get();
        switch(opcion){
            case '1':cout<<"\n Ingrese un numero entero al arbol:
";cin>>valor;
                arb.adicionarNodo(raiz,valor);break;
            case '2':arb.mostrarArbol(raiz);break;
            case '3':arb.inicializar();
                cout<<"          #          nodos          ramas="
"<<arb.cantidadDeRamas(raiz); break;
        }
        cin.ignore();
    }
    while( opcion !='4');
    system("pause");
    return 0;
}

```

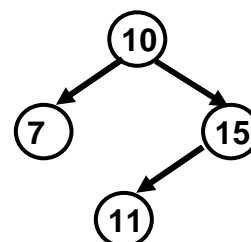
### 8.10. Ejercicios propuestos

1. Escriba la especificación del TAD, el algoritmo y la implementación en C++ para un árbol binario de números enteros en el cual se desea saber si un subárbol se encuentra en otro árbol.
2. Se tiene dos árboles de numeros enteros. Ambos son ABB y se desea integrarlos en uno solo de tal modo que este tercer árbol también sea un ABB como se observa en el siguiente ejemplo:

**Árbol 1**

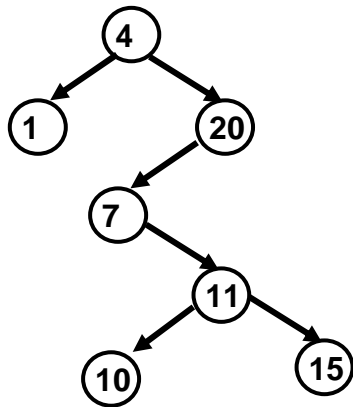


**Árbol 2**





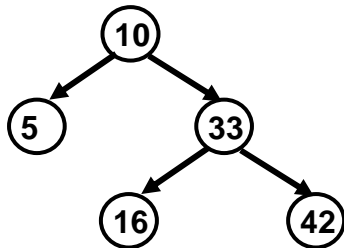
### El árbol 2 se integra al árbol 1



3. Se tiene un ABB de números enteros y una lista dinámica simple también de números enteros y se desea crear un método que permita saber si la lista es una rama del árbol, como se observa en el ejemplo.

**Árbol 1**

**Lista : 10, 33, 42**



4. A continuación se dan tres secuencias de nodos generadas por los recorridos en profundidad, dibuje el árbol que generan estos recorridos:
- Preorden: P-R-A-C-H-T-O-M
  - Inorden: A-R-H-C-P-O-T-M
  - Postorden: A-H-C-R-O-M-T-P
5. Escriba la especificación, el algoritmo y el programa que le permita, dado un nivel en un árbol retorne el número de nodos en ese nivel.
6. Escriba la especificación, el algoritmo y el programa que le permita en un árbol saber cuantas ramas posee dicho árbol.
7. Escriba la especificación, el algoritmo y el programa que dado un árbol retorne un numero natural indicando la longitud de la rama más larga de dicho árbol.
8. Escriba la especificación, el algoritmo y el programa que le permita en un árbol binario, retornar en una lista dinámica simple los nodos que forman la rama más larga de dicho árbol.



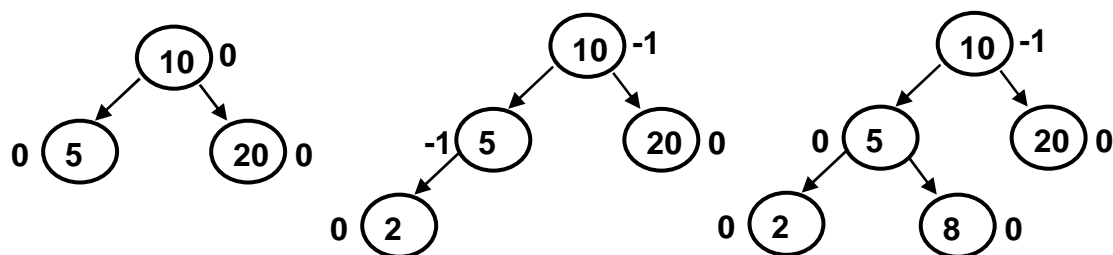
## Lección 9

### Arboles AVL y B

En este capítulo se hace una introducción a los arboles AVL y arboles B los cuales son arboles que en las operaciones de búsqueda resultan en tiempos más cortos que los esperados en un árbol ABB. Los arboles AVL son estructuras que tienen una propiedad que se conoce como equilibrio y que es lo que le da al árbol su velocidad en una consulta. Sin embargo cuando se trata de procesar un volumen de información verdaderamente grande que nos lleve al uso de memoria secundaria, es cuando tenemos que usar un árbol B, siendo el árbol B el sistema de indexación mas requerido sobre todo cuando se clasifican archivos.

#### 9.1. Definición de árbol AVL

El termino árbol AVL, es una palabra que concatena las letras iniciales de los nombres de los creadores de este tipo particular de árbol los cuales son Adelson-Velskii y Landis. "Un árbol binario equilibrado es aquel en el que la altura de los subarboles izquierdo y derecho de cualquier nodo nunca difiere en mas de una unidad." (Joyanes et al., 2005, 319). Por defecto un árbol AVL es ordenado desde la implementación, por lo tanto se necesita de una clave para comparar los elementos e insertar a izquierda o derecha según sea el caso. Cada nodo del árbol registra un factor de balanceo que es la diferencia entre la altura del subárbol derecho y el izquierdo. Este factor puede ser  $-1$ ,  $0$  o  $+1$ . Observe abajo, tres ejemplos de arboles AVL.

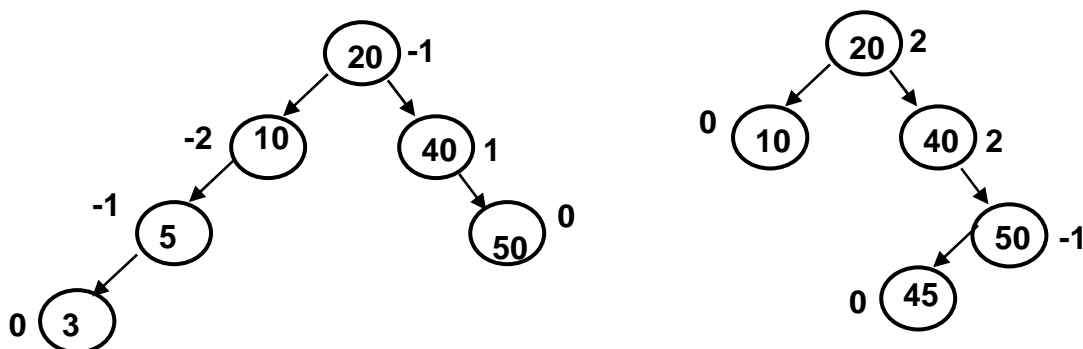


**Figura 9.1.** Árboles AVL





Observe ahora arboles que no son AVL



**Figura 9.2.** Árboles que no son AVL

La estructura de un árbol AVL es similar a la de un árbol binario, excepto por el nodo al cual se le agrega el campo de factor de balance y un enlace mas para tener conexión directa con el padre de un nodo como se muestra abajo en código C++:

```
class ARBOLAVL{
    ENTERO dato;      // información de un numero entero
    int FE;           // factor de balance
    ARBOLAVL *der;    // enlace al nodo izquierdo
    ARBOLAVL *izq;    // enlace al nodo derecho
    ARBOLAVL *arriba; // enlace al padre del nodo
public:
    :
};
```

### 9.1.1. Aplicación de arboles AVL

Los arboles tienen diversas aplicaciones como se menciona anteriormente en el capítulo de arboles ABB. Podemos ver que su utilidad esta en el almacenamiento y búsqueda de información por ser una estructura que nos da tiempos más cortos de lo que nos daría una lista enlazada cuando se trata de efectuar consultas. Un árbol AVL aumenta la eficiencia de la estructura en las aplicaciones anteriormente mencionadas por mantener la información ordenada con un tiempo de acceso a los datos relativamente bajos.

### 9.1.2. Ventajas y desventajas

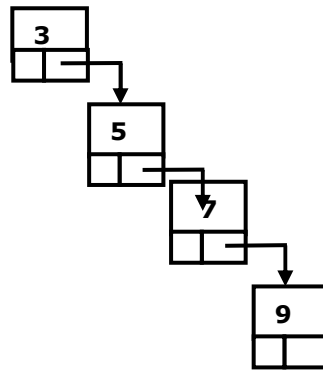
Un árbol AVL es también un árbol ABB, así que las operaciones que tenemos en un árbol ABB también están en un árbol AVL con la diferencia de que hay que mantener un equilibrio en el árbol usando nuevas operaciones que son parte de operaciones como insertar o



eliminar y que generaran ventajas o desventajas en un árbol AVL que se presentan a continuación:

## Ventajas

- Los arboles AVL tienen como propiedad fundamental a su favor el que forman estructuras que posibilitan una mayor velocidad en las operaciones de búsqueda que los ABB simples.
- En los arboles ABB simples del capítulo anterior tienen el problema de que en el peor de los casos pueden dar lugar a un árbol degenerado que es un árbol cuya forma es la de una lista y con un tiempo mucho mayor en las operaciones de búsqueda. Observe el siguiente árbol degenerado que en los arboles AVL no sucede y que es formado a partir de una lista y que no posee en realidad forma de árbol:



**Figura 9.3.** Lista: 3, 5, 7, 9

## Desventaja

- En los arboles ABB simples podemos realizar inserciones y eliminaciones, de manera similar también podemos hacerlas en un árbol AVL, sin embargo estas no resultan tan directas debido a la condición de equilibrio que debe existir en el árbol lo cual generara un tiempo adicional en estas operaciones.

### 9.1.3. Operaciones

Las operaciones en un árbol AVL son las mismas que se usan sobre un árbol ABB con la diferencia que las operaciones como inserción y eliminación de un elemento pueden llevar a dos operaciones adicionales que se conocen como rotaciones:

- Rotaciones simple
- Rotaciones doble

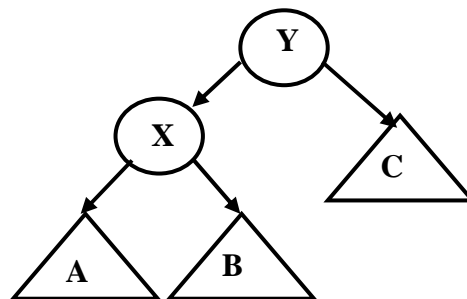


Una rotación simple es necesaria para restablecer el equilibrio cuando después de una eliminación o inserción de un elemento el árbol pierde su equilibrio. En caso una rotación simple no restablezca el equilibrio se usara entonces una rotación doble para volver a balancear el árbol.

#### 9.1.3.1. Inserción

Conceptualmente hay dos etapas en la inserción en un árbol balanceado. En la primera etapa, la inserción en sí puede llevar a la violación de las reglas de balanceo, lo cual si se da, se arregla en una segunda etapa. Si en la primera etapa de la inserción no se produce un desequilibrio, entonces no es necesario realizar ninguna otra operación. Pero si se produjera el desequilibrio, la segunda etapa reacomoda los nodos y modifica los factores de balance para restaurar el equilibrio. A esta segunda etapa se le conoce como rebalanceo o rotación del árbol.

Cuando es necesaria una rotación, sus efectos están limitados a los nodos cercanos o pertenecientes al camino desde el nodo insertado hasta la raíz. Esta operación de rotación que aplicamos sobre un árbol ABB para mantener equilibrado el árbol es la que explicaremos y detallaremos a continuación.

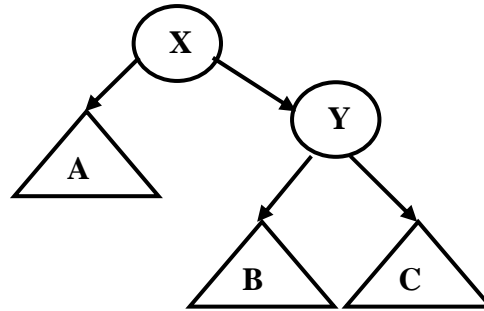


**Figura 9.4.** Árbol ABB

Miremos por un momento el árbol de arriba y veamos que se cumplen las condiciones siguientes:

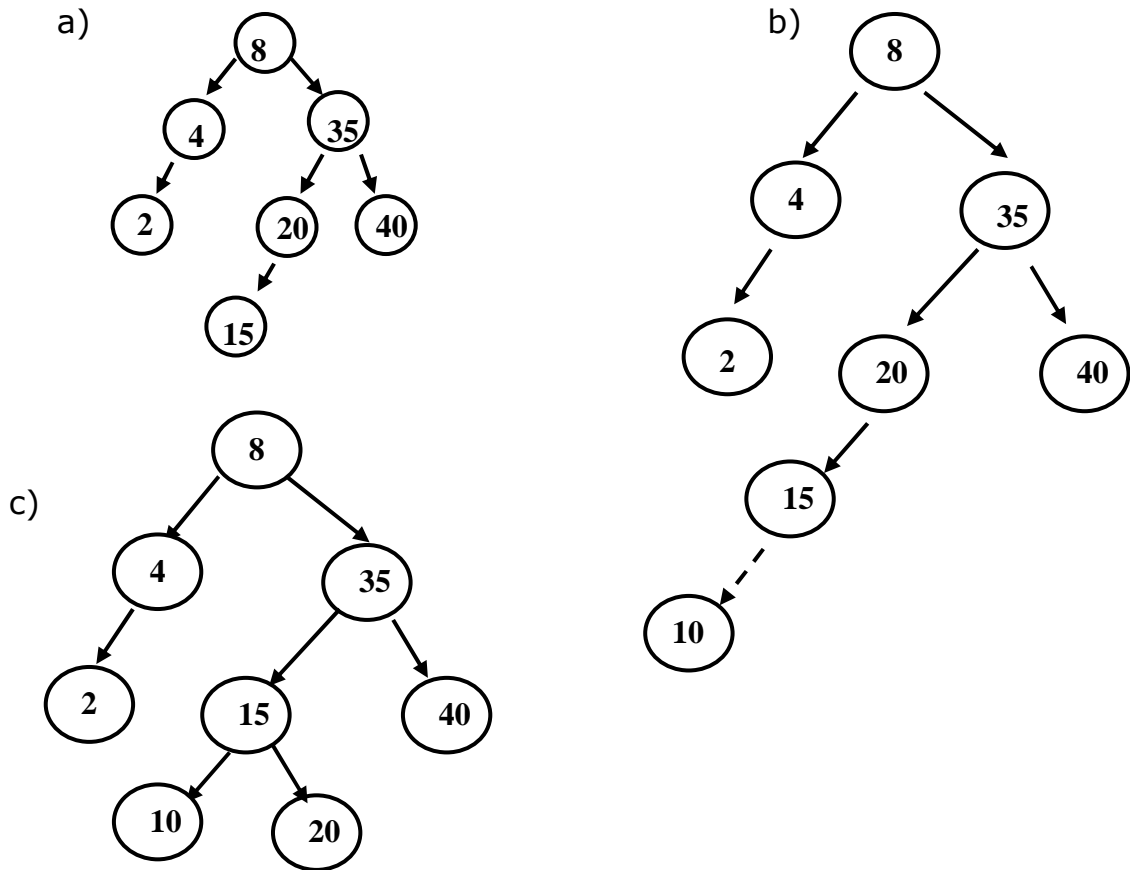
- Que  $x$  sea menor que  $y$ ,
- Además todos los nodos del subárbol A deben ser menores que  $x$  e  $y$
- También todos los nodos del subárbol B deben ser mayores que  $x$  pero menores que  $y$
- Finalmente  $y$  es menor que todos los nodos del subárbol C y a la vez mayores que  $x$ .

Manteniendo estas condiciones el árbol se ha modificado y es equivalente al árbol de abajo. Como puede verificarse fácilmente por las desigualdades descritas en las condiciones anteriores, el nuevo árbol sigue manteniendo el orden entre sus nodos, es decir, sigue siendo un árbol binario. A esta transformación se le denomina rotación simple (o sencilla).



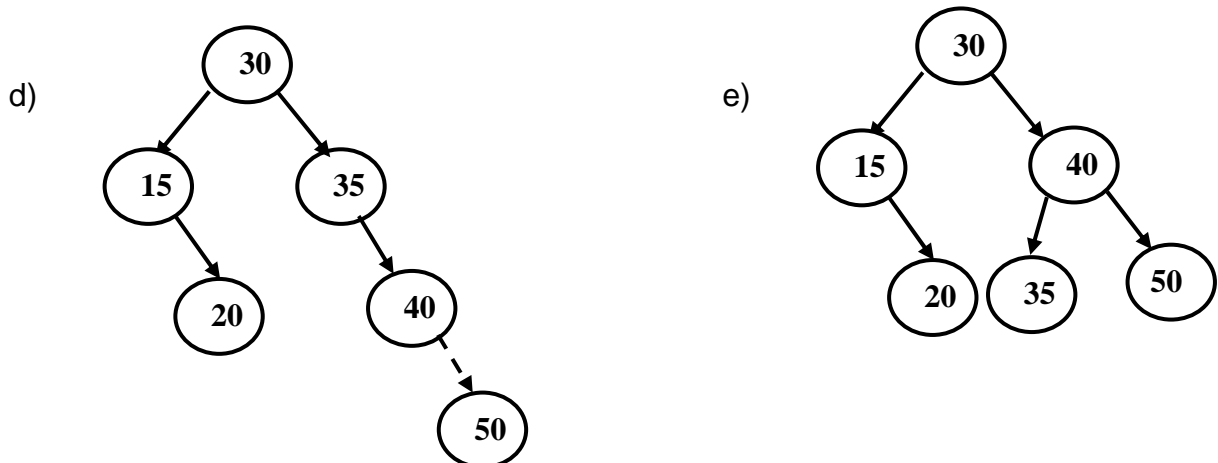
**Figura 9.5.** Árbol ABB luego de una rotación simple

Veamos un ejemplo concreto. Deseamos insertar el número 10 en el árbol de enteros de la figura (a). La inserción se muestra punteada (b). Sin embargo, como puede verse, la inserción ha provocado la pérdida de la propiedad de equilibrio del árbol. ¿Qué se hace para devolver nuevamente el equilibrio? Definitivamente procedemos a realizar una rotación simple. Esta rotación se conoce como *rotación izquierda* ya que la "pérdida de equilibrio se produce hacia ese lado. Puede verse el árbol luego de la rotación(c): la propiedad de equilibrio ha sido devuelta. Como vemos, la rotación mantiene el orden entre los elementos del árbol, de tal manera que el tercer árbol(c) si es AVL.



**Figura 9.6.** Inserción en un árbol AVL

En el árbol (b) el "desequilibrio" en el subárbol con raíz 20 era enteramente hacia la izquierda y por lo tanto, como ya dijimos, la rotación efectuada se denomina **rotación simple a la izquierda**. En el caso de un "desequilibrio" hacia la derecha, la rotación es análoga y se denomina **rotación simple a la derecha**. A continuación se ven dos árboles: el primero(d) tiene un "desequilibrio hacia la derecha" y el segundo(e) es el resultado de aplicar una rotación simple a la derecha.

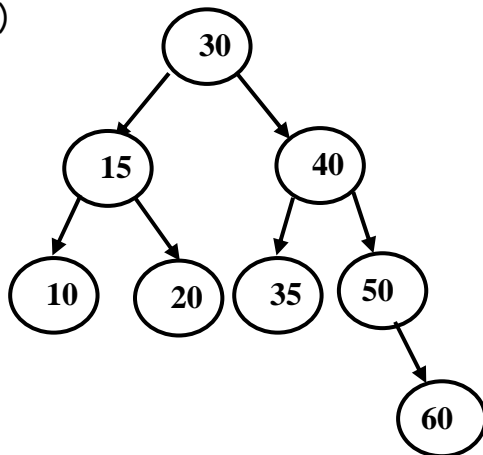


**Figura 9.7.** Rotación en un árbol AVL

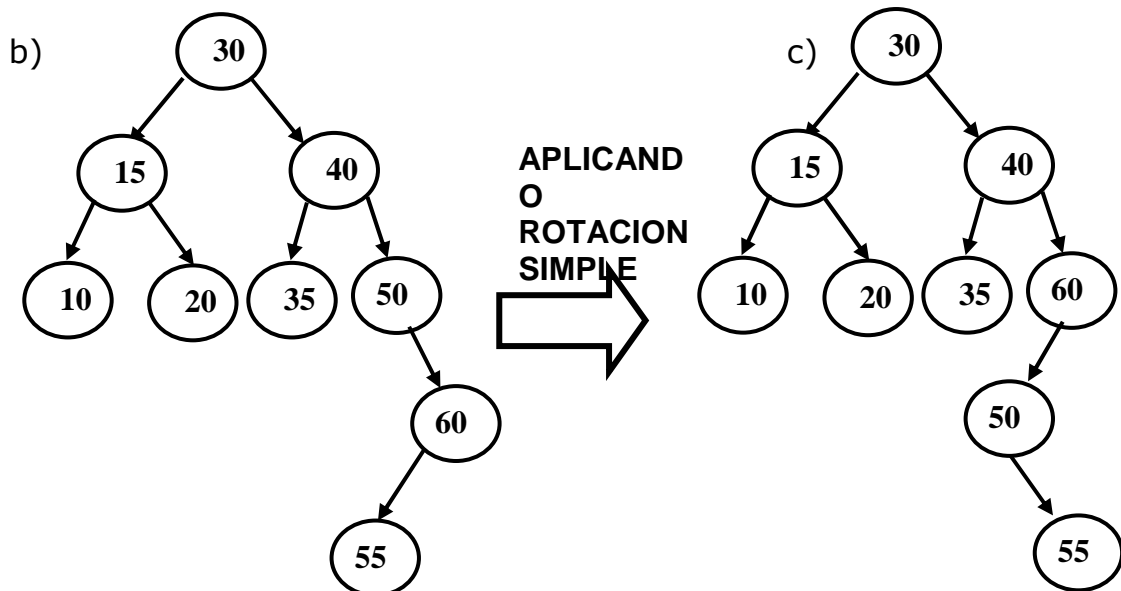


Las rotaciones anteriores devuelven el equilibrio al árbol cuando existe un desbalance a la izquierda o a la derecha en una operación de inserción, sin embargo y como veremos, pueden ocurrir que la rotación no recomponga el árbol y los desequilibrios continúen como es el caso del árbol que se muestran abajo(a) al cual se le insertara un elemento.

a)



Al insertar el elemento 55 el árbol(a) se desequilibra y a pesar de aplicar una rotación simple al árbol(b) continua aun desequilibrado como se observa en el árbol (c).



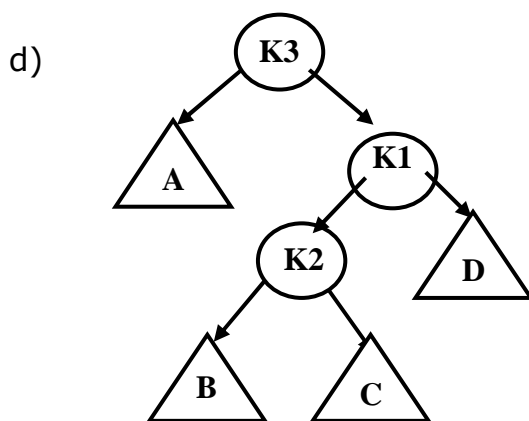
Como puede usted observar la operación de rotación simple no ha corregido el desequilibrio de altura del árbol(b) . Para devolver el balance al árbol se requiere una operación denominada rotación



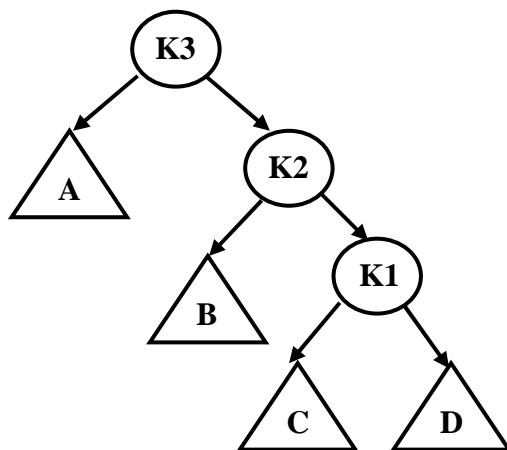
doble, similar a la simple pero que abarca cuatro sub árboles en vez de tres. Se distinguen además dos tipos de rotación doble:

- Rotación doble derecha-izquierda
- Rotación doble izquierda-derecha.

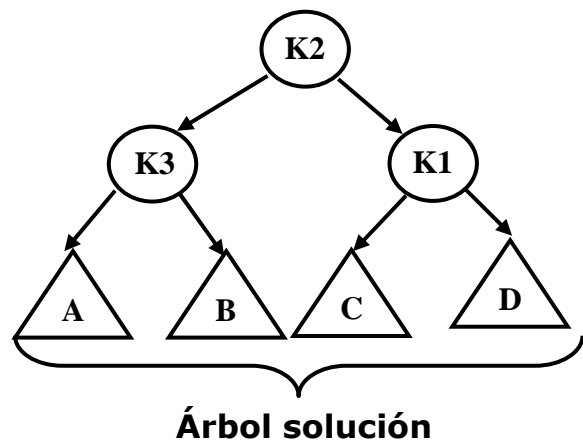
Abajo se puede ver a través de un diagrama, como el árbol (d) que se encuentra desequilibrado, al aplicársele una rotación doble derecha-izquierda vuelve a tomar su equilibrio en el árbol (f), esto implica primero una rotación derecha donde el árbol(e) aun no recobra su equilibrio, pero finalmente vuelve a estar balanceado en el árbol(f) una vez aplicada una rotación izquierda:



e) Primera rotación : derecha  
rotación : izquierda



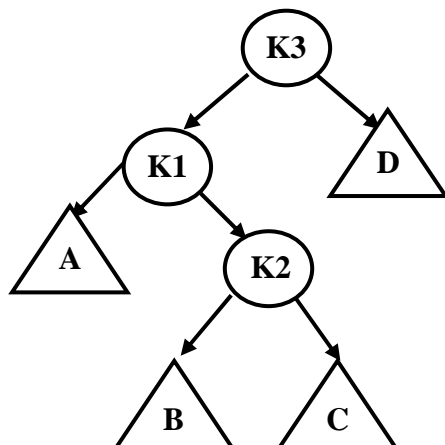
f) Segunda



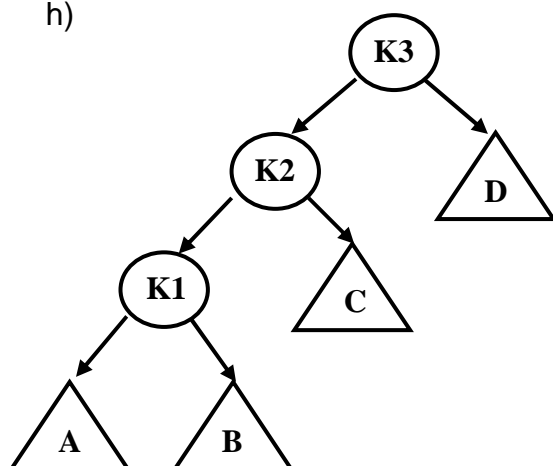
De manera similar el diagrama de abajo detalla una rotación doble izquierda-derecha para devolver el equilibrio a un árbol(g) que requerirá primero una rotación izquierda (h) y luego una rotación derecha(i) para balancear el árbol.



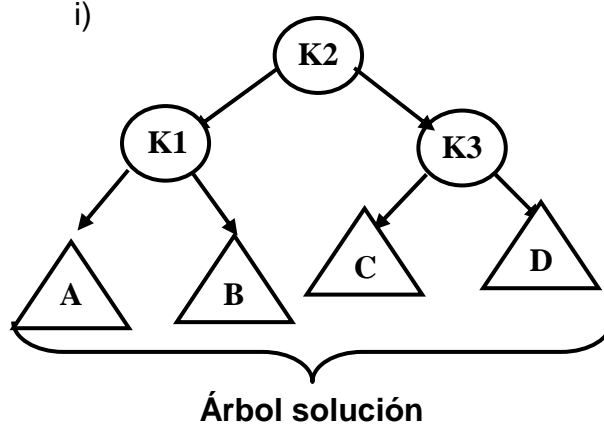
g)



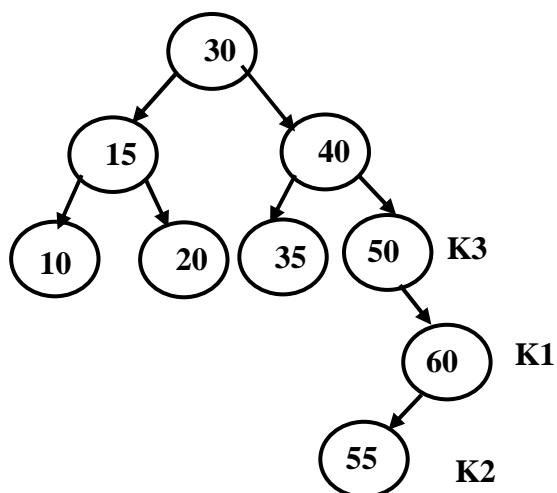
h)



i)



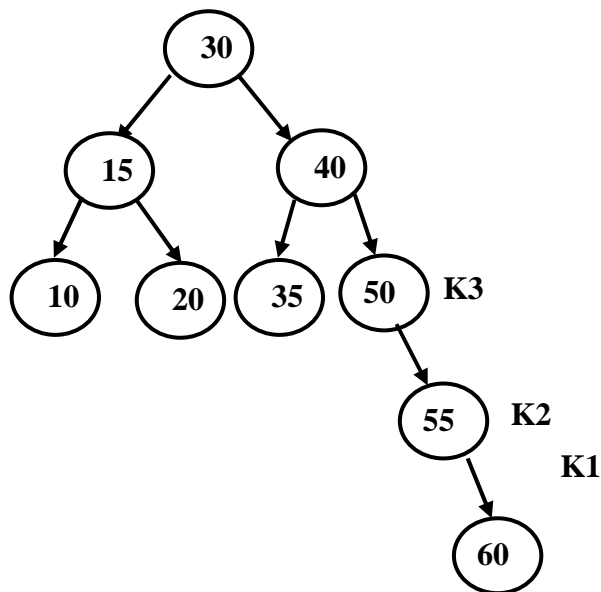
Luego de lo expuesto aplicamos al problema anterior una rotación derecha - izquierda para equilibrar el árbol que no se equilibrio con una rotación simple:



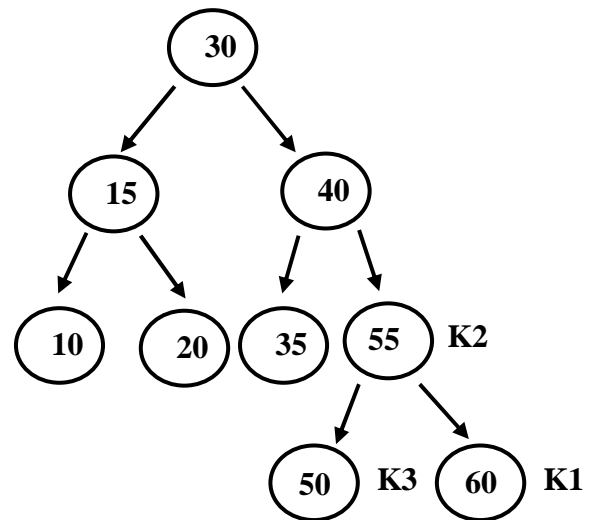




### Rotación derecha izquierda



### Rotación

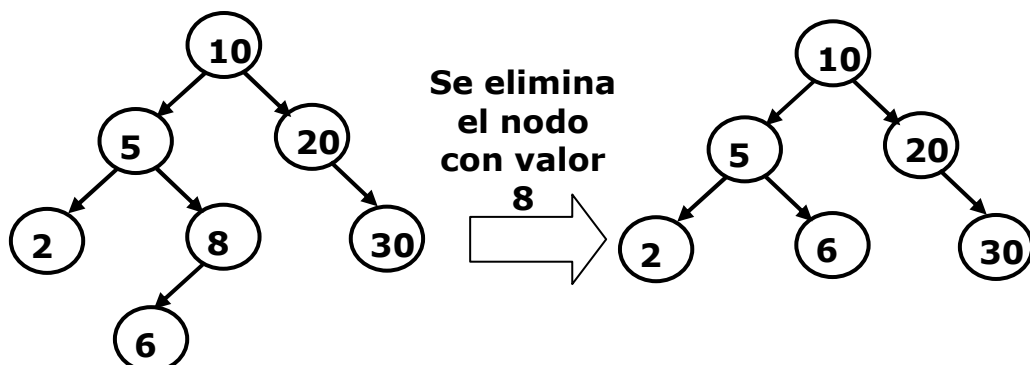


#### 9.1.3.2. Eliminación

El desarrollar un algoritmo que pueda eliminar un nodo de un árbol AVL es básicamente el mismo que se utiliza para la inserción. La eliminación al igual que la inserción va a ocasionar en algunos casos el desequilibrio del árbol, teniendo que usarse para estas situaciones las operaciones de rotación simple o rotación doble para devolver nuevamente el balance al árbol.

Un algoritmo de eliminación al igual que en un árbol ABB debe recorrer el árbol para encontrar el nodo a eliminar, después existen tres casos en cada uno de los cuales se puede dar la siguiente situación:

- En caso el nodo a eliminar fuera un nodo hoja se procede sin mayor problema a su eliminación.
- En caso el nodo que se va a eliminar tuviera un solo hijo, el nodo puede ser eliminado después de ajustar un puntero del nodo padre para saltar el nodo como se observa abajo:





- Si el nodo a eliminar tuviera dos hijos el caso es mas complicado. Se acostumbra a reemplazar el nodo a eliminar por el nodo de menor valor de su subárbol derecho y después proceder con la eliminación.

Se deja al alumno el desarrollo del algoritmo para eliminar un nodo en un árbol AVL.

#### 9.1.4. Construcción de un árbol AVL

Primero hacemos la especificación del TAD árbol del siguiente modo:

*Especificación de los TAD's y los algoritmos*

##### **Especificacion ENTERO**

###### **variable**

num : entero.

###### **metodos**

INGENTERO() : no retorna valor.

WISENTERO() : no retorna valor.

RETNUM() : retorna entero.

CATNUM(x) : no retorna valor.

###### **significado**

INGENTERO lee un numero entero.

WISENTERO visualiza un numero entero.

RETNUM retorna un numero entero.

CATNUM captura un numero entero.

##### **Fin\_especificacion**

##### **Especificación ARBOLAVL**

**usa** ENTERO.

###### **variable**

dato : ENTERO.

FE : entero.

raiz : ARBOLAVL.

sgte : ARBOLAVL.

arriba :ARBOLAVL.

tope : entero.

i : entero.

###### **métodos**

INSERTAR(apuntador,dato) : no retorna valor.

VACIO(apuntador) : retorna valor lógico.

INORDEN(apuntador) : no retorna valor.

PREORDEN(apuntador) : no retorna valor.

POSTORDEN(apuntador) : no retorna valor.

equilibrar(a, nodo,rama,nuevo) : no retorna valor.

RSI : no retorna valor.



RSD	: no retorna valor.
RDD	: no retorna valor.
RDI	: no retorna valor.

### Significado

*INSERTAR* permite insertar un TAD dato del tipo ENTERO

*VACIO* verifica si apuntador tiene una dirección nula o no.

*INORDEN* recorre el árbol en inorden.

*PREORDEN* recorre el árbol en preorden.

*POSTORDEN* recorre el árbol en postorden.

*Equilibrar* equilibra el árbol cuando se produce un desbalance.

RSI rotación simple izquierda.

RSD rotación simple derecha.

RDD rotación doble derecha.

RDI rotación doble izquierda.

### Fin\_especificacion

Procedimiento equilibrar(a, nodo, rama, nuevo)

salir ← FALSE

Mientras (nodo ≠ nulo y no salir) hacer

Si (nuevo ≠ nulo) entonces

Si (rama = IZQUIERDO) entonces

leer(nodo, FE)

asignar(nodo, FE-1)

Sino

leer(nodo, FE)

asignar(nodo, FE+1)

Fin\_si

Sino

Si (rama = IZQUIERDO) entonces

leer(nodo, FE)

asignar(nodo, FE+1)

Sino

leer(nodo, FE)

asignar(nodo, FE-1)

Fin\_si

Fin\_si

leer(nodo, FE)

Si (FE = 0) entonces

salir ← verdad

Sino

Si (FE = -2) entonces

leer(izq(nodo), FE)

Si (FE = 1) entonces

RDD(a, nodo)

Sino

RSD(a, nodo)



```

        Fin_si
        salir ← TRUE
    Sino
        Si (FE = 2) entonces
            leer(der(nodo),FE)
            Si (FE = -1) entonces
                RDI(a, nodo)
            Sino
                RSI(a, nodo)
            Fin_si
            salir ← verdad
        Fin_si
    Fin_si
Si (arriba(nodo)≠nulo) entonces
    Si (der(arriba(nodo))= nodo) entonces
        rama ← DERECHO
    Sino
        rama ← IZQUIERDO
    Fin_si
Fin_si
nodo ←arriba( nodo)
Fin_mientras
Fin_procedimiento

Procedimiento INSERTAR(apuntador, dato)
    padre ←nulo
    actual ←apuntador
    leer(actual, e)
    Mientras(no VACIO(actual) y dato.RETNUM() ≠ e.RETNUM())
hacer
    padre ← actual
    Si (dato.RETNUM() < e.RETNUM()) entonces
        actual ← izq(actual)
    Sino
        Si (dato.RETNUM() > actual->e.RETNUM())
        entonces
            actual ←der( actual)
        Fin_si
    Fin_si
    leer(actual, e)
Fin_mientras
Si ( VACIO(actual)) entonces
    Si ( VACIO(padre)) entonces
        crear(apuntador)
        asignar(apuntador,dato)
        izq(apuntador)←nulo
        der(apuntador)←nulo

```



```

        arriba(apuntador) ← nulo
        FE ← 0
        Asignar(apuntador, FE)
    Sino
        leer(padre, e)
        Si (dato.RETNUM() < e.RETNUM()) entonces
            crear(actual)
            izq(padre) ← actual
            asignar(actual, dato)
            izq(actual) ← nulo
            der(actual) ← nulo
            FE ← 0
            Asignar(actual, FE)
            Equilibrar(a, padre, IZQUIERDO, verdad)
        Sino
            Si (dato.RETNUM() > e.RETNUM()) entonces
                crear(actual)
                der(padre) ← actual
                asignar(actual, dato)
                izq(actual) ← nulo
                der(actual) ← nulo
                FE ← 0
                Asignar(actual, FE)
                Equilibrar(a, padre, DERECHO, verdad)
            Fin_si
        Fin_si
    Fin_si
Fin_procedimiento

```

#### Implementación del TAD

```

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
#define TRUE 1
#define FALSE 0

enum {IZQUIERDO, DERECHO};

class ENTERO{
    int num;
    public:
    void INGENTERO(){ cout<<"\n Ing entero: ";cin>>num; }
    void VISENTERO(){ cout<<" "<<num; }
    int RETNUM(){ return num;}
    void CATNUM(int x){ num=x;}
}

```



```

};

/* Estructuras y tipos */
class ARBOLAVL{
    ENTERO dato;
    int FE;
    ARBOLAVL *der;
    ARBOLAVL *izq;
    ARBOLAVL *arriba;
public:
    void menu(){
        cout<< "\nMENU DE OPCIONES\n";
        cout<<"<1> insertar      \n";
        cout<<"<2> Inorden        \n";
        cout<<"<3> Preorden        \n";
        cout<<"<4> Postorden       \n";
        cout<<"<5> Salir          \n";
    }
    /* Insertar en árbol ordenado: */
    void Insertar(ARBOLAVL *&a, ENTERO dat){
        ARBOLAVL *padre = NULL;
        ARBOLAVL *actual = a;
        /*Buscar el dato en el árbol, manteniendo un puntero al
nodo padre */
        while(!Vacio(actual) && dat.RETNUM() != actual-
>dato.RETNUM()) {
            padre = actual;
            if(dat.RETNUM() < actual->dato.RETNUM()) actual =
actual->izq;
            else if(dat.RETNUM() > actual->dato.RETNUM()) actual =
actual->der;
        }
        /* Si se ha encontrado el elemento, regresar sin insertar
*/
        if(!Vacio(actual)) return;
        /* Si padre es NULL, entonces el árbol estaba vacío, el
nuevo nodo
será el nodo raiz */
        if(Vacio(padre)) {
            a = new ARBOLAVL;
            a->dato = dat;
            a->izq = a->der = NULL;
            a->arriba = NULL;
            a->FE = 0;
        }
        /* Si el dato es menor que el que contiene el nodo padre,
lo
insertamos en la rama izquierda */

```



```

else if(dat.RETNUM() < padre->dato.RETNUM()) {
    actual = new ARBOLAVL;
    padre->izq = actual;
    actual->dato = dat;
    actual->izq = actual->der = NULL;
    actual->arriba = padre;
    actual->FE = 0;
    Equilibrar(a, padre, IZQUIERDO, TRUE);
}
/* Si el dato es mayor que el que contiene el nodo padre,
lo
insertamos en la rama derecha */
else if(dat.RETNUM() > padre->dato.RETNUM()) {
    actual = new ARBOLAVL;
    padre->der = actual;
    actual->dato = dat;
    actual->izq = actual->der = NULL;
    actual->arriba = padre;
    actual->FE = 0;
    Equilibrar(a, padre, DERECHO, TRUE);
}
}
void Equilibrar(ARBOLAVL *&a, ARBOLAVL *&nodo, int rama, int
nuevo)
{
    int salir = FALSE;
    /* Recorrer camino inverso actualizando valores de FE: */
    while(nodo && !salir) {
        if(nuevo)
            if(rama == IZQUIERDO) nodo->FE--; /* Depende
de si
añadimos ... */
            else nodo->FE++;
        else
            if(rama == IZQUIERDO) nodo->FE++; /* ... o
borramos */
            else nodo->FE--;
        if(nodo->FE == 0) salir = TRUE; /* La altura de las rama
que
empieza en nodo no ha variado, salir de Equilibrar */
        else if(nodo->FE == -2) { /* Rotar a derechas y
salir: */
            if(nodo->izq->FE == 1) RDD(a, nodo); /*
Rotación doble */
            else RSD(a, nodo); /* Rotación simple */
            salir = TRUE;
        }
        else if(nodo->FE == 2) { /* Rotar a izquierdas y salir: */

```



```

        if(nodo->der->FE == -1) RDI(a, nodo); /* Rotación
doble */
        else RSI(a, nodo);                /* Rotación simple */
        salir = TRUE;
    }
    if(nodo->arriba)
        if(nodo->arriba->der == nodo) rama = DERECHO;
        else rama = IZQUIERDO;
        nodo = nodo->arriba; /* Calcular FE,
siguiente nodo
del camino. */
    }
}
/* Comprobar si el árbol está vacío: */
int Vacio(ARBOLAVL *r){
    return r==NULL;
}
/* Recorrido de árbol en inorden, aplicamos la función func, que tiene
el prototipo:
void func(int*);
*/
void INORDEN(ARBOLAVL *a){
    if(a){
        INORDEN(a->izq);
        a->dato.VISENTERO();
        INORDEN(a->der);
    }
}
void PREORDEN(ARBOLAVL *a){
    if(a){
        a->dato.VISENTERO();
        PREORDEN(a->izq);
        PREORDEN(a->der);
    }
}
void POSTORDEN(ARBOLAVL *a){
    if(a){
        POSTORDEN(a->izq);
        POSTORDEN(a->der);
        a->dato.VISENTERO();
    }
}
// Funciones de equilibrado:
void RDD(ARBOLAVL *&raiz, ARBOLAVL *&nodo){
    ARBOLAVL *Padre = nodo->arriba;
    ARBOLAVL *P = nodo;
    ARBOLAVL *Q = P->izq;
    ARBOLAVL *R = Q->der;

```





```

ARBOLAVL *B = R->izq;
ARBOLAVL *C = R->der;
if(Padre)
    if(Padre->der == nodo) Padre->der = R;
    else Padre->izq = R;
else raiz = R;
/* Reconstruir árbol: */
Q->der = B;
P->izq = C;
R->izq = Q;
R->der = P;
/* Reasignar padres: */
R->arriba = Padre;
P->arriba = Q->arriba = R;
if(B) B->arriba = Q;
if(C) C->arriba = P;
/* Ajustar valores de FE: */
    switch(R->FE) {
        case -1: Q->FE = 0; P->FE = 1; break;
        case 0: Q->FE = 0; P->FE = 0; break;
        case 1: Q->FE = -1; P->FE = 0; break;
    }
    R->FE = 0;
}
/* Rotación doble a izquierdas */
void RDI(ARBOLAVL *&a, ARBOLAVL *&nodo)
{
    ARBOLAVL *Padre = nodo->arriba;
    ARBOLAVL *P = nodo;
    ARBOLAVL *Q = P->der;
    ARBOLAVL *R = Q->izq;
    ARBOLAVL *B = R->izq;
    ARBOLAVL *C = R->der;
    if(Padre)
        if(Padre->der == nodo) Padre->der = R;
        else Padre->izq = R;
    else a = R;
    /* Reconstruir árbol: */
    P->der = B;
    Q->izq = C;
    R->izq = P;
    R->der = Q;
    /* Reasignar padres: */
    R->arriba = Padre;
    P->arriba = Q->arriba = R;
    if(B) B->arriba = P;
    if(C) C->arriba = Q;
    /* Ajustar valores de FE: */

```



```

switch(R->FE) {
    case -1: P->FE = 0; Q->FE = 1; break;
    case 0: P->FE = 0; Q->FE = 0; break;
    case 1: P->FE = -1; Q->FE = 0; break;
}
R->FE = 0;
}
/* Rotación simple a derechas */
void RSD(ARBOLAVL *&a, ARBOLAVL *&nodo)
{
    ARBOLAVL *Padre = nodo->arriba;
    ARBOLAVL *P = nodo;
    ARBOLAVL *Q = P->izq;
    ARBOLAVL *B = Q->der;
    if(Padre)
        if(Padre->der == P) Padre->der = Q;
        else Padre->izq = Q;
    else a = Q;
    /* Reconstruir árbol: */
    P->izq = B;
    Q->der = P;
    /* Reasignar padres: */
    P->arriba = Q;
    if(B) B->arriba = P;
    Q->arriba = Padre;
    /* Ajustar valores de FE: */
    P->FE = 0;
    Q->FE = 0;
}
/* Rotación simple a izquierdas */
void RSI(ARBOLAVL *&a, ARBOLAVL *&nodo)
{
    ARBOLAVL *Padre = nodo->arriba;
    ARBOLAVL *P = nodo;
    ARBOLAVL *Q = P->der;
    ARBOLAVL *B = Q->izq;
    if(Padre)
        if(Padre->der == P) Padre->der = Q;
        else Padre->izq = Q;
    else a = Q;
    /* Reconstruir árbol: */
    P->der = B;
    Q->izq = P;
    /* Reasignar padres: */
    P->arriba = Q;
    if(B) B->arriba = P;
    Q->arriba = Padre;
    /* Ajustar valores de FE: */

```



```

        P->FE = 0;
        Q->FE = 0;
    }
    /* Función de prueba para recorridos del árbol */
    void Mostrar(int *d)
    {
        printf("%d, ", *d);
    }
};
/* Programa de ejemplo */
int main()
{
    ARBOLAVL *ArbolInt=NULL, arb;
    int altura;
    int nnodos;
    ENTERO ent;
    char opcion;
    do
    {
        arb.menu();
        cout<<"\n ingrese opcion : ";
        opcion=cin.get();
        switch(opcion){
            case '1':ent.INGENTERO();
                    arb.Insertar(ArbolInt, ent); break;
            case '2':cout<<"InOrden: ";
                    arb.INORDEN(ArbolInt); break;
            case '3':cout<<"PreOrden: ";
                    arb.PREORDEN(ArbolInt); break;
            case '4':cout<<"PostOrden: ";
                    arb.POSTORDEN(ArbolInt); break;
        }
        cin.ignore();
    }
    while( opcion !='5');
    system("pause");
    return 0;
}

```

## 9.2. Árbol B

Un árbol B es también un ABB, ósea un árbol de búsqueda con ciertas limitaciones que solucionan problemas en un ABB simple y que garantizan que el árbol B estará en todo momento balanceado y que el desperdicio de espacio producto de eliminaciones si las hubieran no resultara excesivo.



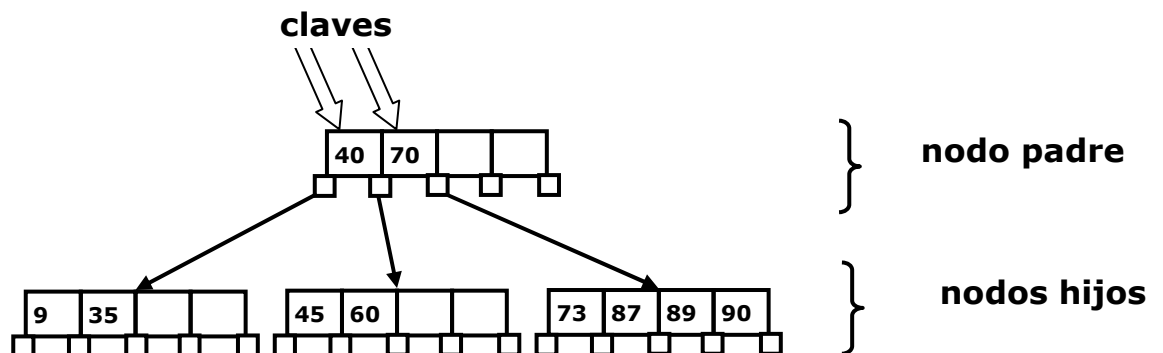
### 9.2.1. Definición de árbol B

Un árbol B es un árbol de  $m$  ramas, con paginas también denominadas nodos, que almacenan un máximo de  $m-1$  claves y que tienen las siguientes características:

- Un factor básico en un árbol B es el orden ( $m$ ), el cual nos dice que es el número máximo de ramas que pueden salir de un nodo.
- Un árbol del cual parten  $m$  ramas, registrara un máximo de  $m-1$  claves por nodo.
- El árbol siempre tiende a estar ordenado.
- En un árbol B, los nodos hojas están en un mismo nivel.
- En un árbol B Todos los nodos rama o intermedios (no la raíz), deben tener entre  $m/2$  y  $m$  ramas no nulas.
- El número mínimo de claves por nodo es  $(m/2)-1$ .
- El número máximo de consultas para dar con una clave está dado por la profundidad( $h$ )

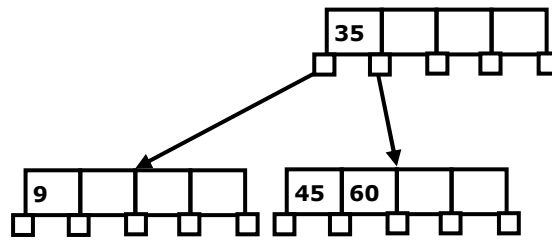
A continuación se observa un ejemplo de un árbol B de orden 5. Puede verse 3 niveles con páginas que tienen de cuatro elementos como máximo, excepto la raíz que puede albergar un solo elemento.

Ejemplo de un árbol-B de ORDEN 5 y de profundidad 2.



**Figura 9.8.** Árbol-B

Ahora observe el árbol de abajo y notará que no es un árbol B, debido a que aun manteniendo el orden, una página que no es la raíz tiene una cantidad de claves menor a  $m/2$  ósea es menor a  $5/2$  claves. La página con error es la que contiene el valor 9.



**Figura 9.8.** No es un árbol-B

### 9.2.2. Aplicación de arboles B

Los arboles B se usan cuando existe un volumen de información a procesar tan grande que se hace necesario almacenarla en disco, y en el cual los arboles ABB simples o arboles AVL no hacen un buen tiempo de acceso a la información registrada en memoria secundaria. "En general se denominan arboles multicamino a aquellos arboles de grado mayor que dos. Estos arboles son muy utilizados en la construcción y mantenimiento de arboles de búsqueda con gran cantidad de nodos, y por tanto, usualmente almacenados en memoria secundaria en los que se necesitan inserciones y supresiones." (Hernández, 2006, 245).

### 9.2.3. Ventajas y desventajas

#### Ventajas

- Un árbol B tiene mayores beneficios que otras estructuras cuando de gran cantidad de información se trata y se da esencialmente cuando los nodos se alojan en almacenamiento secundario, evitando el uso reiterado de acceso al disco. Un árbol ABB simple y un árbol AVL hacen uso deficiente del almacenamiento secundario.
- La ventaja de un árbol B es que al tener un control sobre el número de nodos hijos que pueda tener un nodo interno, la altura del árbol disminuye, las tareas de tener que equilibrar el árbol también disminuyen, aumentando por consecuencia la eficiencia de la estructura.
- El número de consultas en un árbol B se puede predecir.
- El número de consultas no aumentara con el numero de registros a ordenar como si sucede con un árbol ABB simple.

#### Desventajas

- El mantener un árbol B es mas complejo, tal es así que una operación de ingresar, borrar o modificar causa en un nodo unión o desunión obligando a una serie de operaciones para equilibrar el árbol.



### 9.2.4. Operaciones

Las operaciones básicas que se pueden realizar en un árbol B son básicamente tres:

- Insertar una clave.
- Eliminar una clave.

#### 9.2.4.1. Inserción de claves

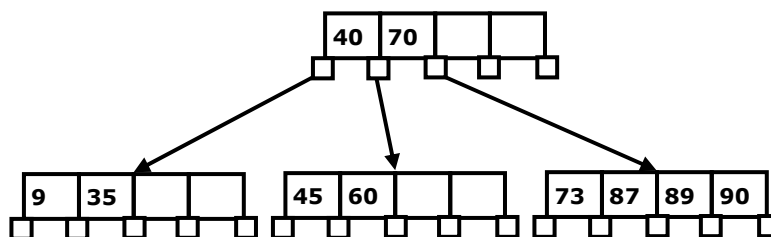
La inserción de una clave implica una operación de búsqueda para poder encontrar el nodo en el que se va a insertar. Si la clave a insertar ya existe, no se procede con ninguna operación de inserción en un árbol B.

Cuando la clave no existe en el árbol y una vez encontrado el nodo adecuado donde se insertara y que no está ocupado totalmente, solo hay que ver en que lugar del nodo se registrara, reacomodando las posiciones de las otras claves en el nodo. Sin embargo el problema se da cuando una clave se quiere insertar en un nodo en el que ya no hay espacio, para tal propósito se divide el nodo en dos nodos y debemos seguir los siguientes pasos:

1. Se toma el valor medio que está entre los claves del nodo y la nueva clave.
2. Las claves que tienen un valor menor al valor medio de la clave tomada se colocan en la nueva página o nodo izquierdo, y las claves que tienen un valor mayor que el valor medio se colocan en el nuevo nodo derecho, actuando el valor medio como la clave separadora.
3. La clave que actuó como separadora se coloca o se promociona al nodo padre, lo que puede llevar a que el nodo padre se divida en dos, y así repetirse nuevamente la operación de división.

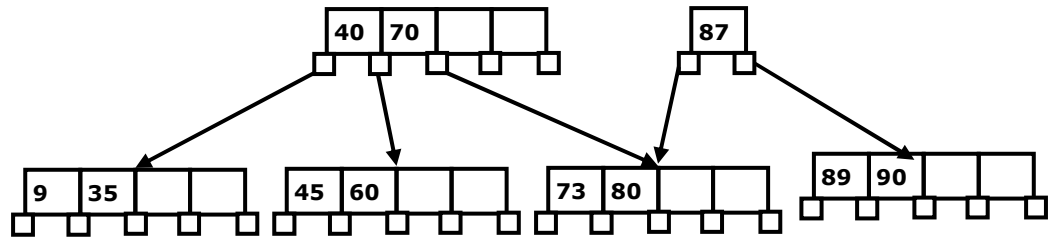
Si las operaciones de división de nodos siguen hasta la raíz, se crea entonces una nueva raíz con una única clave que actuará como separador y con dos hijos.

- Queremos insertar la clave 80 en el árbol de abajo.

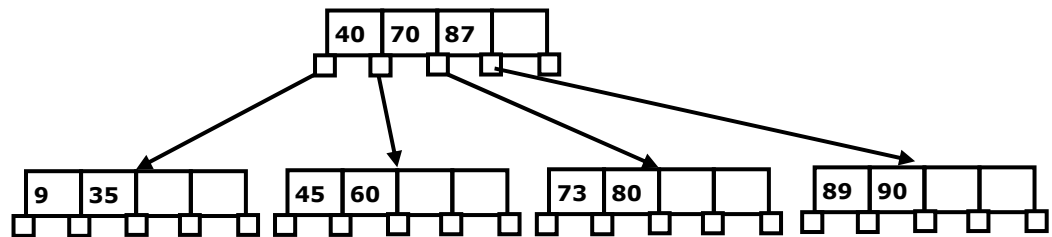




Al no haber espacio se crea un nodo nuevo.



Finalmente se inserta la clave 87 en el nodo padre.



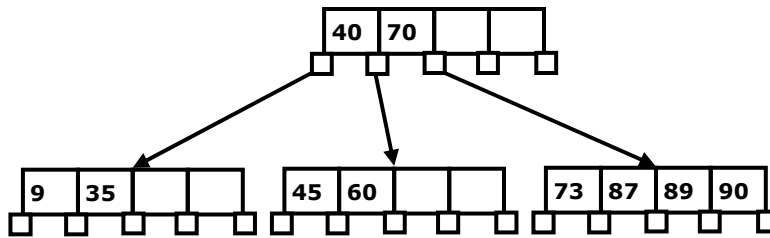
#### 9.2.4.2. Eliminación de claves

El eliminar una clave en un árbol B puede conducirnos a enfrentar los siguientes casos:

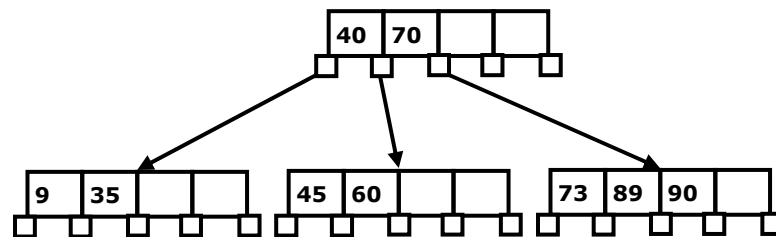
1. Un caso es cuando en las claves de un nodo se encuentra la clave que queremos eliminar y el total de claves es mayor al mínimo de claves. En esta situación, se elimina la clave elegida del nodo y se da por terminado la operación de eliminación.
2. Otra situación es cuando en las claves de un nodo se encuentra la clave que queremos eliminar, pero a diferencia de la situación anterior el nodo tiene solo el mínimo de claves. Ante esta situación no queda más que buscar otra clave que pueda reemplazar a la clave que se borrara. En caso no exista otra clave se buscara la unión de dos nodos. En caso alguno de los nodos hermanos adyacentes (izquierda o derecha) al nodo que se destruirá tiene un numero de claves mayor al mínimo permitido, la clave padre del nodo que se destruirá con clave por borrar lo sustituirá cuando sea borrado. Si no existiera un nodo hermano adyacente con mas claves del mínimo necesario, el nodo donde ocurrirá la eliminación y un nodo hermano adyacente se unirán con la clave padre que desaparece del nodo padre, si este ultimo nodo tuviera mas del mínimo de claves la operación termina, en caso contrario se analiza la eliminación en el nodo padre volviendo a repetir los pasos anteriormente explicados.



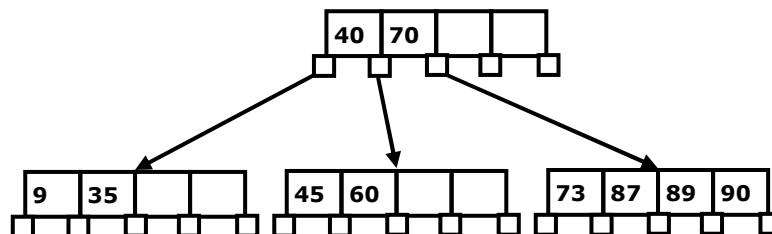
3. Si el nodo que contiene la clave que se va a eliminar constituye la raíz del árbol y además se da el caso que posee solo una clave, la clave se elimina y se genera una nueva raíz para el árbol que estará formada por la unión de los nodos hijos que dependían de la raíz anterior que fue eliminada.
- El caso más sencillo es borrar una clave en un nodo hoja. Por ejemplo se quiere borrar la clave 87.



Luego de eliminar la clave tenemos:

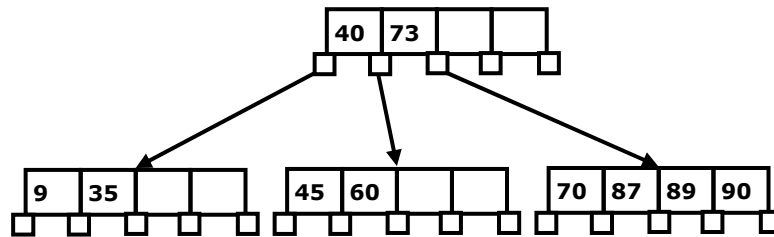


- Cuando la clave a eliminar esta en un nodo intermedio como la clave 70, el algoritmo intercambiara la clave por el valor que sigue. Observe el árbol:

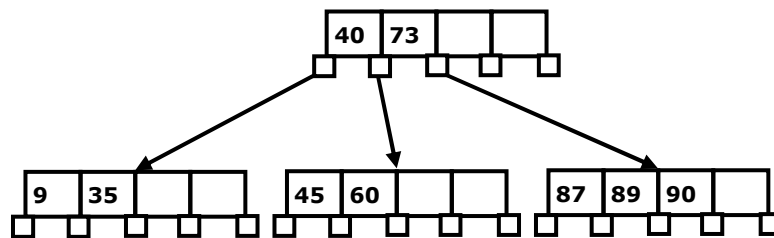


Se ha intercambiado el 70 con el 73 como se observa abajo.





Ahora eliminamos la clave 70 del nodo hoja.



### 9.2.5. Construcción de un árbol B

Primero hacemos la especificación del TAD árbol del siguiente modo:

*Especificación de los TAD's y los algoritmos*

#### Especificación ITEM

##### variable

clave : entero.  
p : PAGINA.

##### métodos

capClave(k) : no retorna valor.  
capPag(q) : no retorna valor.  
retPag() : retorna tipo PAGINA.  
retClave() : retorna entero.

##### significado

capClave registra el valor de una clave.  
capPag registra un apuntador.  
retPag devuelve un apuntador del tipo PAGINA.  
retClave devuelve el valor de una clave.

#### Fin\_especificacion

#### Especificación PAGINA

##### variable

m : entero.  
e : arreglo de tipo ITEM.  
sgte : PAGINA.

**métodos**

menu() : no retorna valor.

generarNodo(mm, q, u) : no retorna valor.

ingresarClave(x, a, CREAMODO,v) : no retorna valor.

bajoFlujo(c, a, s, h) : no retorna valor.

eliminar(p, h, a, r) : no retorna valor.

borrarClave(x, a, h) : no retorna valor.

**Significado**

*menu* muestra las alternativas.

*generarNodo* crea un nuevo nodo en el árbol.

*ingresarClave* ingresa una clave en el árbol.

*bajoFlujo* si hay un numero de claves menor al mínimo en un nodo.

*eliminar* completa la eliminación de una clave del árbol.

*borrarClave* retira una clave x del árbol.

**Fin\_especificacion***Implementación del TAD*

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
#define n 2

class PAGINA;
class ITEM{
    int clave;
    PAGINA *p;
public:
    void capClave(int k){clave=k;}
    void capPag(PAGINA *q){ p=q;}
    PAGINA *retPag(){ return p; }
    int retClave(){return clave;}
};

class PAGINA{
    int m;
    ITEM e[2*n+1];
    PAGINA *sgte;
public:
```



```

void menu(){
    cout<< "\nMENU DE OPCIONES\n";
    cout<<"<1> insertar clave \n";
    cout<<"<2> eliminar clave \n";
    cout<<"<3> Salir \n";
}
void generarNodo(int mm, PAGINA *q, ITEM u){
    m=mm; sgte=q; e[1]=u;
}
void ingresarClave(int x,PAGINA *a,bool
&CREARNODO,ITEM &v){
    int i,l,r;
    PAGINA *b,*t;
    ITEM u;
    if(a==NULL) {
        CREARNODO=true;
        v.capClave(x);
        v.capPag(NULL);
    }
    else {
        l=1;
        r=a->m+1;
        while (l<r) {
            i=(l+r)/2;
            if (a->e[i].retClave() <= x) l=i+1;
            else r=i;
        }
        r--;
        if((r>0) && (a->e[r].retClave()==x))
            CREARNODO=false;
        else{
            if (r==0)
                ingresarClave(x,a->sgte,CREARNODO,u);
            else
                ingresarClave(x,a-
>e[r].retPag(),CREARNODO,u);

            if (CREARNODO) {
                if (a->m<2*n) {
                    CREARNODO=false;
                    a->m=a->m+1;
                    for(i=a->m;i>=r+2;i--){
                        a->e[i] = a->e[i-1];
                    }
                    a->e[r+1]= u;
                }
                else{
                    b=new PAGINA;

```



```

        if (r<=n){
            if (r==n) v=u;
            else {
                v = a->e[n];
                for(i=n;i>=r+2;i--)      a->e[i]=a-
>e[i-1];
                    a->e[r+1]=u;
            };
            for(i=1;i<=n;i++) b->e[i]=a->e[i+n];
        }
        else{
            r=r-n;
            v=a->e[n+1];
            for(i=1;i<=r-1;i++)          b->e[i]=a-
>e[i+n+1];
                b->e[r]=u;
                for(i=r+1;i<=n;i++)      b->e[i]=a-
>e[i+n];
        };
        a->m=n;
        b->m=n;
        b->sgte=v.retPag();
        v.capPag(b);
    }
}
}
}
}
void bajoFlujo(PAGINA *c,PAGINA *a,int s,bool &h){
    PAGINA *b;
    int i,k,mb,mc;
    mc=c->m;
    if (s<mc){
        s = s+1;
        b = c->e[s].retPag();
        mb = b->m;
        k = (mb-n+1)/2;
        a->e[n] = c->e[s];
        a->e[n].capPag(b->sgte);
        if(k>0){
            for(i=1;i<=k-1;i++) a->e[i+n] = b->e[i];
            c->e[s] = b->e[k];
            c->e[s].capPag(b);
            b->sgte = b->e[k].retPag();
            mb = mb - k;
            for(i=1;i<=mb;i++) b->e[i] = b->e[i+k];
            b->m = mb;
            a->m = n-1+k;

```



```

        h=false;
    }
    else{
        for(i=1;i<=n;i++) a->e[i+n]=b->e[i];
        for(i=s;i<=mc-1;i++) c->e[i]=c->e[i+1];
        a->m = 2*n;
        c->m = mc-1;
        h = mc <= n;
    }
}
else{
    if (s==1) b = c->sgte;
    else b = c->e[s-1].retPag();

    mb =b->m+1;
    k =(mb-n)/2;
    if(k>0){
        for(i=n-1;i>=1;i--) a->e[i+k]=a->e[i];
        a->e[k]=c->e[s];
        a->e[k].capPag(a->sgte);
        mb=mb-k;
        for(i=k-1;i>=1;i--) a->e[i]=b->e[i+mb];
        a->sgte=b->e[mb].retPag();
        c->e[s]=b->e[mb];
        c->e[s].capPag(a);
        b->m=mb-1;
        a->m=n-1+k;
        h=false;
    }
    else{
        b->e[mb]=c->e[s];
        b->e[mb].capPag(a->sgte);
        for(i=1;i<=n-1;i++) b->e[i+mb]=a->e[i];
        b->m=2*n;
        c->m=mc-1;
        h=mc<=n;
    };
};
};
void eliminar(PAGINA *p,bool &h, PAGINA *a, int r){
    PAGINA *q;
    q=p->e[p->m].retPag();
    if (q!=NULL){
        eliminar(q,h,a,r);
        if (h) bajoFlujo(p,q,p->m,h);
    }
    else{
        p->e[p->m].capPag(a->e[r].retPag());
    }
}

```



```

        a->e[r]=p->e[p->m];
        p->m=p->m-1;
        h=p->m<n;
    }
};

void borrarClave(int x,PAGINA *a,bool &h){
    int i,l,r;
    PAGINA *q;
    if (a==NULL)
        h=false;
    else{
        l=1;
        r=a->m+1;
        while (l<r){
            i=(l+r)/2;
            if (a->e[i].retClave() < x) l=i+1;
            else r=i;
        }
        if (r==1) q=a->sgte;
        else q=a->e[r-1].retPag();
        if((r<=a->m) && (a->e[r].retClave()==x)) {
            if (q==NULL){
                a->m=a->m-1;
                h=a->m<n;
                for(i=r;i<=a->m;i++) a->e[i]=a->e[i+1];
            }
            else{
                eliminar(q,h,a,r);
                if (h) bajoFlujo(a,q,r-1,h);
            }
        }
        else{
            borrarClave(x,q,h);
            if(h) bajoFlujo(a,q,r-1,h);
        }
    }
}

void verArbol(PAGINA *p,int level){
    int i;
    if(p!=NULL){
        for(i=1;i<=level;i++) cout<<"  ";
        for(i=1;i<=p->m;i++)      cout<<"          "<<p->e[i].retClave();
        cout<<"\n";
        verArbol(p->sgte,level+1);
        for(i=1;i<=p->m;i++)      verArbol(p->e[i].retPag(),level+1);
    }
}

```



```

    }
};

PAGINA *raiz,*q, obj;
int x;
bool CREARNODO,h;
ITEM u;
int main(){
    raiz=NULL;
    char opcion;
    do{
        obj.menu();
        cout<<"\n ingrese opcion : ";
        opcion=cin.get();
        switch(opcion){
            case '1':
                cout<<">";cin>>x;
                obj.ingresarClave(x,raiz,CREARNODO,u);
                if(CREARNODO){
                    q=raiz;
                    raiz=new PAGINA;
                    raiz->generarNodo(1, q, u);
                }
                obj.verArbol(raiz,0);
                break;
            case '2':cout<<">";cin>>x;
                h=true;
                obj.borrarClave(x,raiz,h);
                obj.verArbol(raiz,0);
                break;
        }
        cin.ignore();
    }while(opcion !='3');
    System("pause");
    return 0;
    cout<<"\n fin del programa ";
}

```

Al ejecutarse el programa se verá el siguiente menú de opciones:

```

MENU DE OPCIONES
<1> insertar clave
<2> eliminar clave
<3> Salir
ingrese opcion :

```

Cada vez que se elige la opción 1, se solicita una clave de entrada. Después de ingresar todas las claves se mostraran agrupadas en



nodos en el árbol B. Por ejemplo se han ingresado las claves 10. 20. 30. 40 y 5, la última en ingresar fue la clave 5 y produjo la siguiente salida:

#### MENU DE OPCIONES

<1> insertar clave

<2> eliminar clave

<3> Salir

ingrese opcion : 1

>5

20

5 10

30 40

### 9.3. Ejercicios resueltos

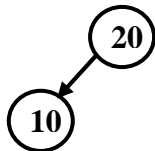
#### Ejemplo 01

Grafique como se va formando un árbol AVL, nodo por nodo según la secuencia siguiente 20, 10,8,40,30,15. Indique además cuando se produzca un desbalance que tipo de rotación está haciendo.

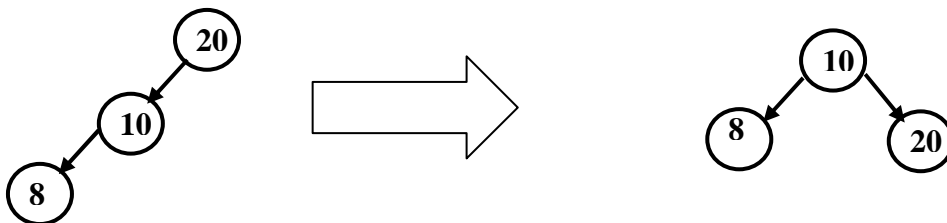
ingresa 20



Ingresa 10



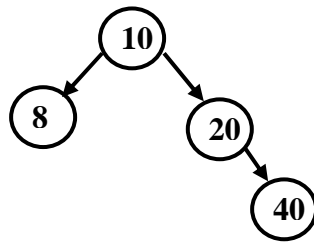
Ingresa 8 : ROTACION SIMPLE



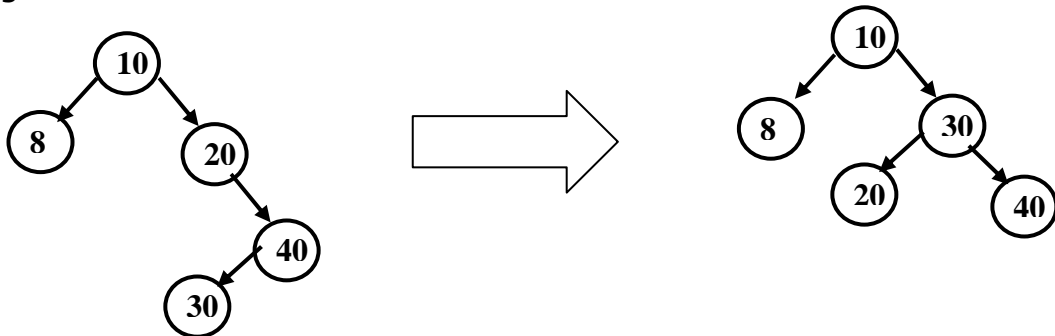




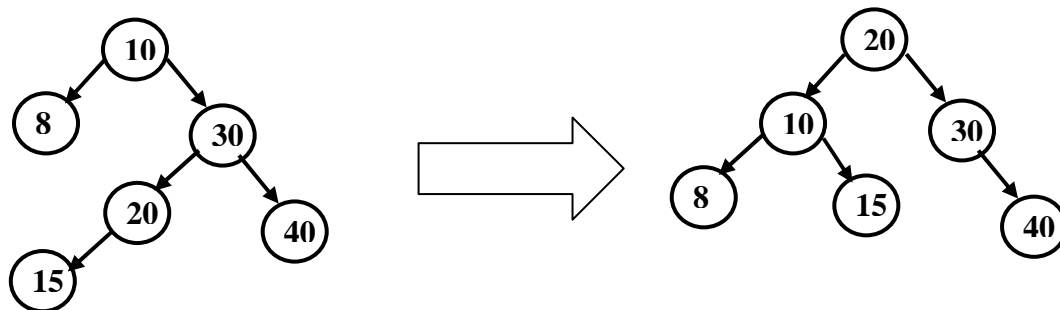
Ingresa 40



Ingresa 30 : ROTACION DOBLE

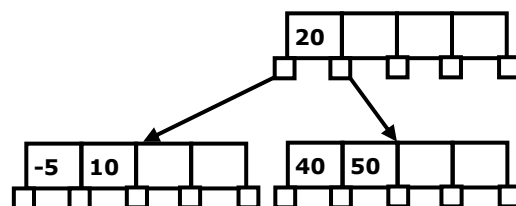


Ingresa 15: ROTACION DOBLE



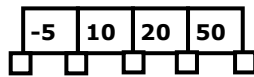
### Ejemplo 02

Dado el árbol B de orden 5 se desea eliminar la clave 40. Grafique el árbol después de eliminar la clave 40.



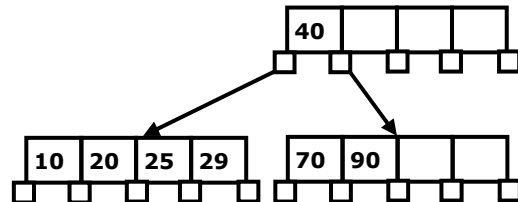


Después de Eliminar la clave 40

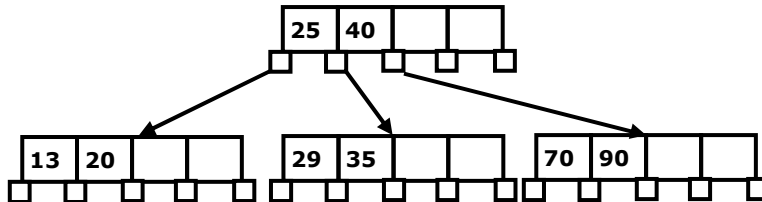


### Ejemplo 03

Dado el árbol B de orden 5 se desea ingresar la clave 35. Grafique el árbol después de ingresar la clave 35.



Después de ingresar la clave 35

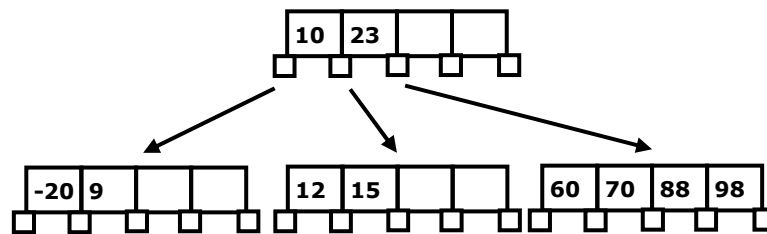


#### 9.4. Ejercicios propuestos

1. Dada la secuencia de claves enteras 20, 10, 30, 5, 25, 12, 3, 35, 22, 11, 6, 2. Represente gráficamente el árbol AVL correspondiente e indique en qué momento(al insertar que nodo) se efectuó una rotación.
2. Dada la secuencia de claves enteras: 100, 29, 71, 82, 48, 39, 101, 22, 46, 17, 3, 20, 25, 10. Represente gráficamente el árbol AVL correspondiente.
3. Dada la secuencia del ejercicio 1, genere una especificación, el algoritmo y el programa que permita saber cuantas rotaciones simples y cuantas rotaciones dobles se han producido.
4. Escriba la especificación y el algoritmo para que con los valores de los nodos hoja de un arbol binario común se forme un árbol AVL.
5. Dada la secuencia de claves enteras 20, 10, 30, 5, 25, 12, 3, 35, 22, 11, 6, 2. Detalle gráficamente como se realiza cada operación de inserción para cada claves expuestas arriba para construir un árbol B de orden 2.
6. Dada la secuencia de claves enteras 20, 10, 30, 5, 25, 12, 3, 35, 22, 11, 6, 2. Con las que se ha formado un árbol B de orden 2. Grafique cada operación en la cual se elimine una clave.



7. Escriba la especificación, el algoritmo y el programa que cargue una fila secuencial de alumnos en un árbol B de orden 5 y luego realice las operaciones de insertar, eliminar, consultar y modificar. Una vez que el programa finaliza su ejecución la información en el árbol B debe regresarse a la fila secuencial.
8. Dada la secuencia de claves enteras 20, 10, 30, 5, 25, 12, 3, 35, 22, 11, 6, 2, para formar un árbol B de orden 2, muestre el recorrido de los nodos en preorden, inorden y postorden.
9. Dado el árbol B de orden 5 se desea eliminar la clave 10. Grafique el árbol después de eliminar la clave 10.





## Lección 10

### Grafos

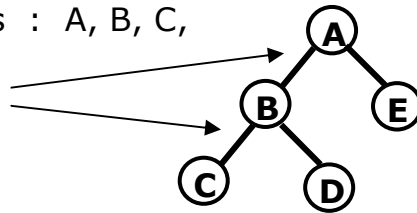
Un grafo es un concepto matemático que nos permite tener la posibilidad de manipular el enmarañado de información que se da al tratar de representar problemas muy complejos como en circuitos, redes o problemas de la vida real en las que se trata de hallar el camino mas corto entre dos ciudades, en fin su ámbito de aplicación es muy general y cubre muchas áreas diversas (Tenenbaum, 1993).

#### 10.1. Definición

Un grafo puede verse como un conjunto de vértices o nodos y aristas o también denominados arcos. Cada arco es una línea que enlaza dos vértices del grafo o también dirigirse o enlazarse a si mismo (Florez, 2005). En este ámbito los vértices son objetos que guardan información y los arcos son la conexión existente entre los vértices. Los grafos pueden representarse de la siguiente manera:

##### Grafo no dirigido

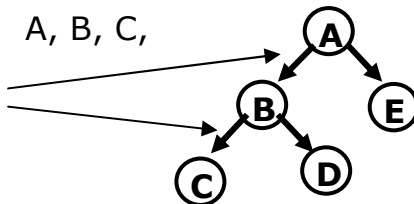
nodos o vértices : A, B, C,  
D, E



Un grafo se denomina grafo dirigido o no dirigido dependiendo si los arcos se denotan con una flecha o no. Por ejemplo el grafo anterior es un grafo no dirigido, mientras que el grafo que esta a continuación es un grafo dirigido.

##### Grafo dirigido

nodos : A, B, C,  
D, E



#### 10.2. Grafo dirigido y no dirigido

En la terminología de grafos encontramos dos tipos de grafos, dirigidos y no dirigidos según se exprese o no el sentido de sus arcos. El capítulo grafos estará orientado al enfoque de los grafos dirigidos, así que cuando usemos el término grafos significará que nos



referimos a un grafo dirigido y el término arco se referirá a un arco también dirigido, a menos que se diga o se deduzca lo contrario.

### 10.2.1. Grafo dirigido

Un grafo dirigido  $G$  es también llamado un dígrafo en el cual cada arco  $e$  de  $G$  tiene ya una dirección. En este grafo cada arista  $e$  está descrita por un par ordenado  $(u,v)$ , donde  $u$  y  $v$  son nodos del grafo  $G$

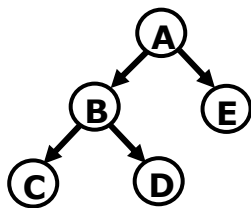
Como  $G$  es un grafo en el que cada arista  $e=(u,v)$ , podemos definir  $G$  también como el par  $(V,A)$  donde  $V$  y  $A$  tienen el siguiente significado:

- $V$ : denota el conjunto finito de nodos o también llamados vértices.
- $A$ : denota el conjunto finito de arcos o también llamadas aristas.

Con respecto al arco  $a=(u,v)$  donde  $a$  es un arco que comienza en  $u$  y finaliza en  $v$ , se define  $G$  como una relación que se expresa de la siguiente forma:

$$G = \{ (u, v) / u, v \in V, \text{ además existe un arco con un origen en } u \text{ y un destino en } v \}$$

Por ejemplo el grafo siguiente puede expresarse como se observa a su derecha:



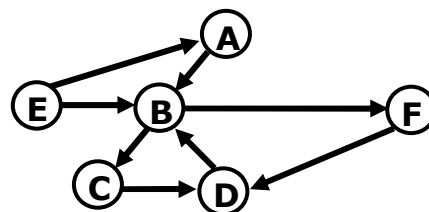
$$V = \{ A, B, C, D, E \}$$

$$A = \{ (A,B), (A,E), (B,C), (B,D) \}$$

### 10.3. Terminología

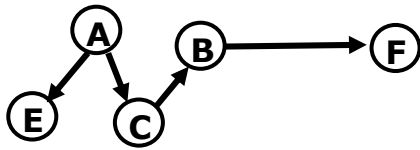
- **Camino**: se define un camino entre dos nodos como una secuencia de nodos o vértices en la que dos nodos contiguos o sucesivos están unidos por un arco del grafo
- **Camino simple**: por camino simple se tiene a un camino en el que desde un vértice a otro, en su secuencia de nodos, ningún nodo se repite, ósea no se pasa dos o mas veces por un mismo nodo.

$\{ E, A, B, F \}$ : camino simple E-F  
 $\{ E, A, B, C, D, B, F \}$  camino no simple E-F

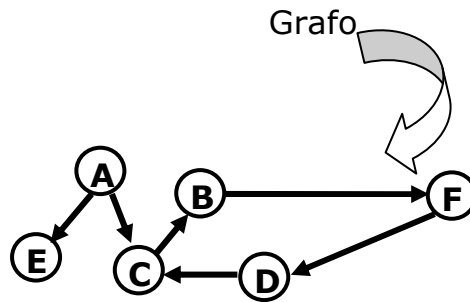


- **Ciclo**: por ciclo se entiende a un camino en el cual el camino simple tiene como inicio y final de camino al mismo vértice. Cuando en un grafo no existen ciclos estamos entonces ante un árbol.

Árbol

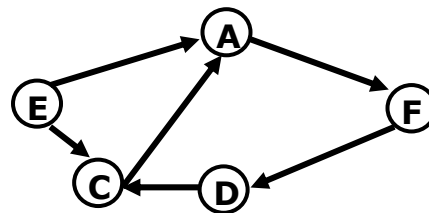


Grafo

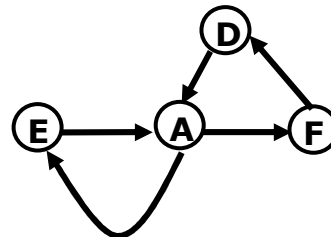
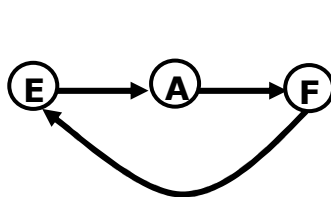


- **Longitud de camino:** es el número de nodos que tiene la secuencia de un camino

{E, A, F} camino de longitud 3  
 {E, C, A, F, D}: camino de longitud 5

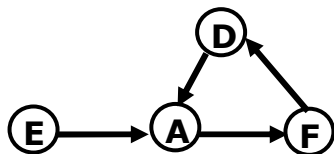


- **Grafo fuertemente conexo:** se le dice al grafo en el cual para todo par de vértices se da un camino entre ellos y a su vez un camino en sentido inverso. A continuación puede observar dos grafos fuertemente conexos.

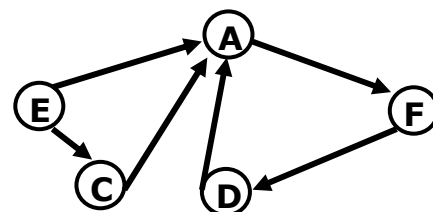


- **Grafo unilateralmente conexo:** se le dice al grafo que presenta el caso contrario al que presenta un grafo fuertemente conexo.

Por ejemplo no se puede llegar de F, A o D a E



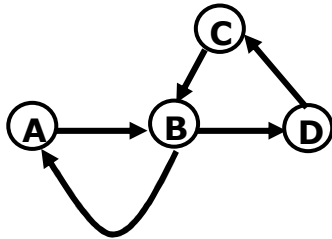
Por ejemplo no se puede llegar de F a E



#### 10.4. Representación de grafos dirigidos


Al igual que otros TAD's, un grafo se puede implementar de varias maneras, así podemos tener una implementación a través del uso de arreglos bidimensionales(matriz de adyacencia) y la otra forma es haciendo uso de listas enlazadas(lista de adyacencia).

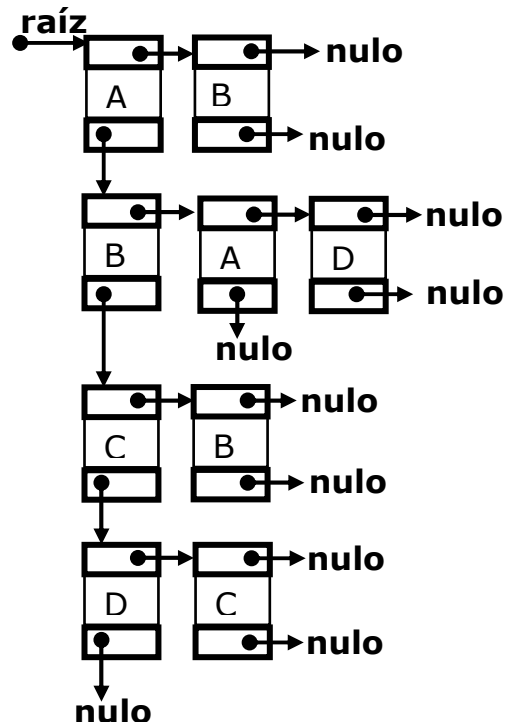
Una matriz de adyacencia se conceptúa como una estructura bidimensional o tabla de dimensiones  $V \times V$ , en el que un par  $a[i][j]$  registrara un valor de uno si realmente existe un arco que va del nodo  $i$  al nodo  $j$ , en caso contrario el valor del par  $a[i][j]$  será cero.



Matriz de adyacencia

	A	B	C	D
A		1		
B	1			1
C		1		
D			1	

Lista de adyacencia 

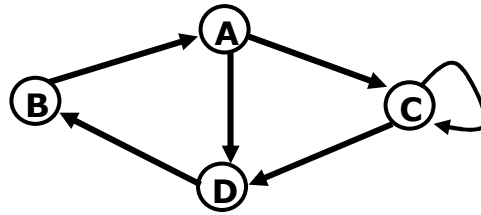


#### 10.4.1. Matriz de adyacencia

Sea  $G$  un grafo dirigido de  $n$  vértices y ordenados desde  $v_1, v_2, v_3$  hasta  $v_n$ , para el cual tenemos la matriz de adyacencia  $A = (a_{ij})$  del grafo  $G$ , matriz de  $n \times n$  y definida de la manera siguiente:

$$a_{ij} = \begin{cases} 1 & \text{si existe un arco que va de } v_i \text{ a } v_j \\ 0 & \text{si no existe una arista de } v_i \text{ a } v_j \end{cases}$$

Una matriz  $A$  que guarda valores de 0 y 1 se conoce como matriz de bits o también matriz booleana y depende de la clasificación de los nodos de  $G$  ya que distintas clasificaciones pueden resultar en distintas matrices de adyacencia. Por ejemplo observemos el siguiente grafo:



$$V = \{ A, B, C, D \}$$

$$A = \{ (A,D), (A,C), (C,D), (D,B), (B,A), (C,C) \}$$

Luego asumiendo que el orden de los vértices es A, B, C y D, la matriz de adyacencia toma la configuración siguiente:

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

En la matriz un valor de 1 indica un camino directo y un 0 representa un camino que no existe.

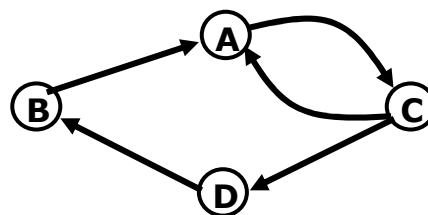
ORIGEN	DESTINO
A	D
A	C
C	D
D	B
B	A
C	C

La importancia de la matriz de adyacencia radica principalmente en que es un mecanismo que nos ayuda a definir los caminos de longitud 1 que se dan en el grafo.

En general:

- A : nos muestra los caminos de longitud 1
- $A_2$  :  $A \cdot A$  nos muestra los caminos de longitud 2
- $A_3$  :  $A_2 \cdot A$  nos muestra los caminos de longitud 3
- :
- $A_k$  :  $A_{k-1} \cdot A$  nos muestra los caminos de longitud k

De lo dicho anteriormente, probémoslo con el siguiente grafo:



$$V = \{ A, B, C, D \}$$

$$A = \{ (A,C), (C,A), (C,D), (D,B), (B,A) \}$$

- Matriz de caminos de longitud 1





$$A =$$

	A	B	C	D
A	0	0	1	0
B	1	0	0	0
C	1	0	0	1
D	0	1	0	0

En la matriz observamos que un 1 representa un camino directo

- Matriz de caminos de longitud 2

$$A_2 =$$

	A	B	C	D
A	1	0	0	1
B	0	0	1	0
C	0	1	1	0
D	1	0	0	0

En la matriz observamos que un 1 representa un camino de longitud 2.

- Matriz de caminos de longitud 3.

$$A_3 =$$

	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	2	0	0	1
D	0	0	1	0

En la matriz observamos que un 1 representa un camino de longitud 3. Sin embargo de C a A hay dos caminos de longitud 3.

- Matriz de caminos de longitud 4.

$$A_4 =$$

	A	B	C	D
A	2	0	0	1
B	0	1	1	0
C	0	1	2	0
D	1	0	0	1

En la matriz observamos que un 1 representa un camino de longitud 4, sin embargo de A a A hay dos caminos de longitud 4, así como también hay dos caminos de longitud 4 de C a C.

Como A es nuestra matriz de adyacencia, definamos ahora la matriz B como la suma total de matrices como sigue:

$$B = A + A_2 + A_3 + \dots + A_m$$

Como resultado dada la entrada  $i, j$  de la matriz B, nos dará el número total de caminos de longitud n o menor, del vértice o nodo  $i$  al  $j$ .



### 10.4.2. Matriz de caminos

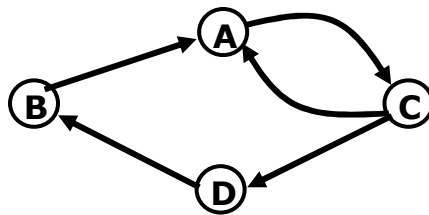
Dado  $G$  un grafo dirigido simple con  $m$  vértices  $v_1, v_2, \dots, v_m$ . Su matriz de caminos es la matriz cuadrada  $m \times m$  donde  $C = v(i, j)$  y que se define como se muestra:

$$C_{ij} = \begin{cases} 1 & \text{si hay un camino de } v_i \text{ a } v_j \\ 0 & \text{si no existe camino de } v_i \text{ a } v_j \end{cases}$$

Desde la matriz  $B$  comenzamos a construir la matriz de caminos  $C$  de la manera siguiente:

$$C_{ij} = \begin{cases} 1 & \text{si tenemos un número diferente de} \\ \text{cero} & \text{en la entrada } i, j \text{ de la matriz } B. \end{cases}$$

Considere el grafo  $G$  anterior con 4 vértices



En el cual sumando las matrices  $A$ ,  $A_2$ ,  $A_3$  y  $A_4$  obtenemos como resultado la matriz  $B$ :

$$B = \begin{matrix} & \begin{matrix} A \end{matrix} \\ \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{matrix} & + & \begin{matrix} A_2 \end{matrix} & + & \begin{matrix} A_3 \end{matrix} & + & \begin{matrix} A_4 \end{matrix} \\ \begin{matrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{matrix} & + & \begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{matrix} & + & \begin{matrix} 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 0 \\ 1 & 0 & 0 & 1 \end{matrix} \end{matrix}$$

$$B = \begin{bmatrix} 3 & 1 & 2 & 2 \\ 2 & 1 & 2 & 1 \\ 3 & 2 & 3 & 2 \\ 2 & 1 & 1 & 1 \end{bmatrix}$$

reemplazando las entradas no nulas por 1, obtenemos la matriz de caminos  $C$  del grafo  $G$ :

$$C = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$



Otra manera de encontrar la matriz de caminos sin tener que hacer los cálculos anteriores es usando el algoritmo de Warshall.

#### 10.4.2.1. Algoritmo de Warshall

Dado un grafo  $G$  dirigido con  $m$  vértices,  $v_1, v_2, \dots, v_m$ . Queremos encontrar la matriz de caminos  $C$  para el grafo  $G$ . Para tal propósito primero se definen las matrices cuadradas  $m \times m$  como sigue:

$$A = C_0, C_1, C_2, \dots, C_m$$

Sea entonces  $C_k(i, j)$  la entrada  $i, j$  de la matriz  $C_k$ , definimos:

$$C_k(i, j) = \begin{cases} 1 & \text{si tenemos un camino simple de } v_i \text{ a } v_j \text{ que no} \\ & \text{usa otros vértices aparte de posiblemente } v_1, \\ & v_2, \dots, v_k \\ 0 & \text{En otro caso} \end{cases}$$

en otras palabras,

$$C_0(i, j) = 1 \quad \text{si hay un arco de } v_i \text{ a } v_j$$

$$C_1(i, j) = 1 \quad \text{si hay un arco de } v_i \text{ a } v_j \text{ que no usa otros vértices excepto } v_1 \text{ o } v_0$$

$$C_2(i, j) = 1 \quad \text{si hay un arco de } v_i \text{ a } v_j \text{ que no usa otros vértices excepto } v_2 \text{ o } v_1 \text{ o } v_0$$

$$C_n = A$$

Warshall se percato que  $C_k(i, j) = 1$  solo se puede dar si ocurre uno de los siguientes casos:

- a) Se tiene un camino simple de  $v_i$  a  $v_j$  que no requiere usar otros vértices excepto  $v_1, v_2, \dots, v_{k-1}$ , por tanto:

$$C_{k-1}(i, j) = 1$$

- b) Se tiene un camino simple de  $v_i$  a  $v_k$  y otro camino simple  $v_k$  a  $v_j$  que no requiere usar otros vértices excepto  $v_1, v_2, \dots, v_{k-1}$ , por tanto:

$$C_{k-1}(i, k) = 1 \quad \text{y} \quad C_{k-1}(k, j) = 1$$

De tal modo los valores de la matriz  $C_k$  se pueden obtener por la siguiente afirmación:

$$C_k(i, j) = C_{k-1}(i, j) \vee (C_{k-1}(i, k) \wedge C_{k-1}(k, j))$$

#### 10.4.2.2. Construcción de Matriz de Camino

*Especificación del Tad y los algoritmos*

##### Especificación GRAFO

##### variable

$C$  : arreglo matricial lógico  
 $m$  : entero

##### métodos



GRAFO() :no retorna valor.  
 ingresarMatrizDeAdyacencia() :no retorna valor.  
 matrizDeCaminos() :no retorna valor.  
 mostrarMatrizCamino() :no retorna valor.  
 menú() :no retorna valor.  
**significado**  
*GRAFO* se asigna memoria al arreglo dinámico.  
*ingresarMatrizDeAdyacencia* se ingresan valores 1 y 0.  
*matrizDeCaminos* se genera la matriz de camino(C).  
*mostrarMatrizCamino* visualiza la matriz de camino(C).  
*menú* se visualizan las alternativas del menú.

### Fin\_especificacion

```

Procedimiento matrizDeCaminos()
  entero : k, i, j
  Desde k ← 0 Hasta k < m con incremento 1 Hacer
    Desde i ← 0 Hasta i < m con incremento 1 Hacer
      Desde j ← 0 Hasta j < m con incremento 1 Hacer
         $C[i][j] \leftarrow C[i][j] \vee (C[i][k] \wedge C[k][j])$ 
      Fin_desde
    Fin_desde
  Fin_desde
Fin_procedimiento
  
```

### Implementación del TAD en C++

```

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
class GRAFO{
  bool **C;
  int m;
public:
  void menu(){
    cout<< "\nMENU DE OPCIONES\n";
    cout<< "-----\n" ;
    cout<<"<1> Ingresar Matriz de Adyacencia\n";
    cout<<"<2> Mostrar Matriz de Camino\n";
    cout<<"<3> Salir \n";
  }
  GRAFO(int n){
    m=n;
    C = new bool*[n]; //vector filas
    for (int i=0;i<n;i++)
      C[i] = new bool[n];
  }
  void ingreseMatrizDeAdyacencia(){
    for(int i=0; i<m; i++){
  
```



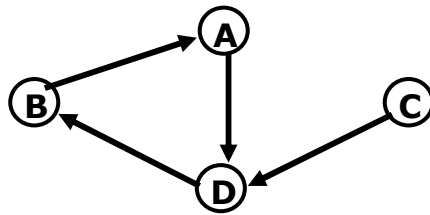
```

        cout<<"\n fila "<<i<<"\n";
    for(int j=0; j<m; j++){
        cout<<"    j="<j<<": "; cin>>C[i][j];
    }
}
matrizDeCaminos();
}
void matrizDeCaminos(){
    int k,i,j;
    for(k=0;k<m;k++)
        for(i=0;i<m;i++)
            for(j=0;j<m;j++)
                C[i][j]=C[i][j] | (C[i][k] & C[k][j]) ;
    cout<<"\n SE HA GENERADO MATRIZ DE CAMINO";
}
void mostrarMatrizCamino(){
    int i,j;
    cout << "\n\nMATRIZ DE CAMINOS \n\n";
    for(i=0;i<m;i++)
    {
        cout << "\n";
        for(j=0;j<m;j++)
            cout << "\t"<<C[i][j];
    }
}
};
int main()
{
    char opcion;
    int x;
    GRAFO g(4);
    do
    {
        g.menu();
        cout<<"\ningrese opcion : ";
        opcion=cin.get();
        switch(opcion){
            case '1':
                g.ingreseMatrizDeAdyacencia();
                break;
            case '2':
                g.mostrarMatrizCamino();
                break;
        }
        cin.ignore();
    }
    while( opcion !='3');
    system("pause");
    return 0;
}

```



Probemos el programa con el grafo que se muestra a continuación:



$$V = \{ A, B, C, D \}$$

$$A = \{ (A, D), (D, B), (B, A), (C, D) \}$$

A =

	A	B	C	D
A	0	0	0	1
B	1	0	0	0
C	0	0	0	1
D	0	1	0	0

Como observamos esta es la matriz de adyacencia que ingresaremos al programa al ejecutar la opción 1 del menú.

#### MENU DE OPCIONES

-----

<1> Ingresar Matriz de Adyacencia

<2> Mostrar Matriz de Camino

<3> Salir

ingrese opcion : 1

fila 0

j=0: 0

j=1: 0

j=2: 0

j=3: 1

fila 1

j=0: 1

j=1: 0

j=2: 0

j=3: 0

fila 2

j=0: 0

j=1: 0

j=2: 0

j=3: 1

fila 3

j=0: 0

j=1: 1

j=2: 0

j=3: 0



### SE HA GENERADO MATRIZ DE CAMINO

Una vez ingresada la matriz de adyacencia se genera la matriz de caminos que se puede visualizar con la opción 2 como se observa a continuación:

#### MENU DE OPCIONES

-----

- <1> Ingresar Matriz de Adyacencia
- <2> Mostrar Matriz de Camino
- <3> Salir

ingrese opcion : 2

#### MATRIZ DE CAMINOS

1	1	0	1
1	1	0	1
1	1	0	1
1	1	0	1

### 10.5. Aplicación

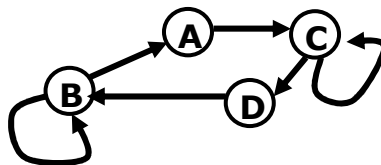
Un grafo es un concepto matemático que tiene múltiples aplicaciones para representar diversos tipos de relaciones que pueden corresponder a diferentes disciplinas:

- Su aplicación es tan diversa que se da también en la resolución de problemas clásicos como el problema de los siete puentes de Königsberg, el problema del cartero chino, el problema del viajante de comercio, etc.
- Como modelo matemático su aplicación se presenta en el control de semáforos en una intersección de calles, para el plano de una planta de un edificio, cadenas de Markov, etc
- Representación de la World- Wide Web, la red mas famosa del mundo.

### 10.6. Ejercicios resueltos

#### Ejemplo 01

Halle la matriz de camino si usar Warshall.



**Solución:**

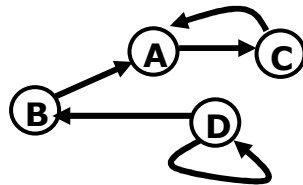
Se considera en todas las matrices un valor de 1 si existe un camino entre dos vértices, exista un camino o más caminos de una determinada longitud.

$$B = \begin{matrix} & \begin{matrix} A \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & & \\ 1 & 1 & 0 \\ \cap \end{pmatrix} \end{matrix} & + & \begin{matrix} A_2 \\ \begin{pmatrix} 0 & 0 & 1 \\ 1 & & \\ 1 & 0 & 1 \\ \cap \end{pmatrix} \end{matrix} & + & \begin{matrix} A_3 \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & & \\ 1 & 1 & 1 \\ 1 \end{pmatrix} \end{matrix} & + & \begin{matrix} A_4 \\ \begin{pmatrix} 1 & 1 & 1 \\ 1 & & \\ 1 & 1 & 1 \\ 1 \end{pmatrix} \end{matrix} \end{matrix}$$

$$C = \begin{pmatrix} 1 & 1 & 1 \\ 1 & & \\ 1 & 1 & 1 \\ 1 \end{pmatrix}$$

**Ejemplo 02**

Halle la matriz de camino si usar Warshall.

**Solución:**

Se considera en todas las matrices un valor de 1 si existe un camino entre dos vértices, exista un camino o más caminos de una determinada longitud.

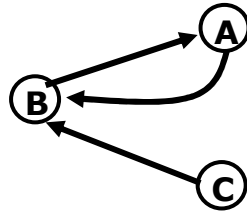
$$B = \begin{matrix} & \begin{matrix} A \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & & \\ 1 & 0 & 0 \\ \cap \end{pmatrix} \end{matrix} & + & \begin{matrix} A_2 \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & & \\ 0 & 0 & 1 \\ \cap \end{pmatrix} \end{matrix} & + & \begin{matrix} A_3 \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & & \\ 1 & 0 & 0 \\ \cap \end{pmatrix} \end{matrix} & + & \begin{matrix} A_4 \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & & \\ 0 & 0 & 1 \\ \cap \end{pmatrix} \end{matrix} \end{matrix}$$

$$C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & & \\ 0 & 0 & 1 \\ \cap \end{pmatrix}$$

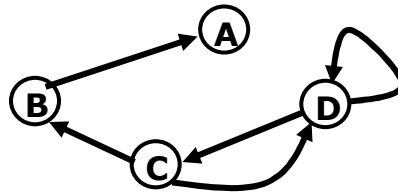


**10.7. Ejercicios propuestos**

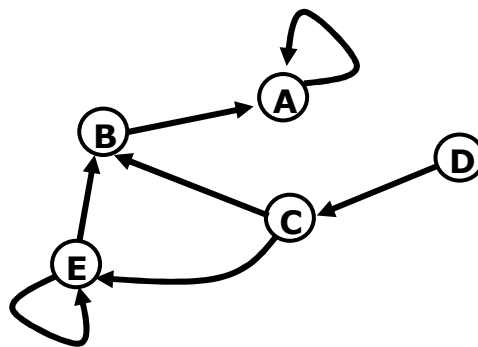
1. Halle la matriz de camino para el siguiente grafo sin usar Warshall.



2. Halle la matriz de camino para el siguiente grafo sin usar Warshall.



3. Halle la matriz de camino para el siguiente grafo sin usar Warshall.

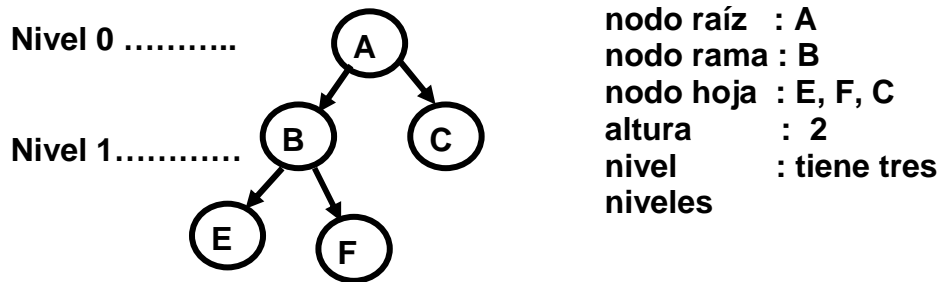


4. Halle la matriz de camino usando Warshall, para cada caso de los ejemplos 01, 02 y 03.



## Resumen

Un árbol es un TAD en el cual sus elementos (nodos) están jerarquizados en una estructura no lineal a través de enlaces(apuntadores) y que puede representarse gráficamente. Para tratarlos se hace necesario una serie de términos como nodo raíz, nodo rama, nodo hoja, grado, orden, altura y nivel, para poder comprender y explicar este tipo de estructura. A continuación observe un árbol donde se usan algunos de estos conceptos:



Los arboles tienen su aplicación en multiprocesadores, en el desarrollo de compiladores, en inteligencia artificial, en operaciones de clasificación u ordenamiento, etc, y al igual que otros TAD tenemos operaciones para adicionar o insertar elementos, eliminar elementos, recorridos en amplitud y profundidad(preorden, inorden y postorden).

Se distinguen dos grupos de arboles, uno es denominado arboles ordenados(ABB, AVL y B) y otro grupo denominado arboles desordenados, ambos grupos se implementan tradicionalmente a través del uso de arreglos o usando asignación dinámica de memoria.

Los arboles ABB se denominan arboles binarios de búsqueda, no son difíciles de implementar pero inadecuados en operaciones de búsqueda donde se requiere un tiempo reducido. Los arboles AVL en las operaciones de consulta no tienen ese problema, son mas veloces debido a una propiedad de equilibrio y que consiste en que cada nodo del árbol registra un factor de balanceo que es la diferencia entre la altura del subárbol derecho y el izquierdo. Este factor puede ser -1 , 0 o +1. Sobre un árbol AVL se pueden aplicar las mismas operaciones que sobre un ABB sin embargo las operaciones de insertar o eliminar pueden llevarnos a operaciones que se conocen como rotaciones las cuales pueden ser rotaciones simples o dobles para devolver el equilibrio al árbol AVL.

Por otro lado cuando se requiere procesar un volumen de información verdaderamente grande que nos lleve al uso de memoria secundaria, es cuando tenemos que usar un árbol B. Un árbol B es un tipo particular de árbol balanceado, en el que un nodo puede acumular más información de lo que almacenaría un nodo en un árbol ABB simple. Un árbol B es también un ABB en el que el desperdicio de



espacio producto de eliminaciones si las hubieran no resultara excesivo.



## Lectura

### Tiempo de acceso

El tiempo que uno se tarda en acceder y recuperar una palabra a partir de la memoria de alta velocidad es a lo sumo unos cuantos microsegundos. El tiempo que se requiere para localizar un archivo en un disco se mide en milisegundos, y en el caso de discos blandos (disquetes) puede exceder de un segundo. Por tanto, el tiempo que se necesita para un solo acceso es miles de veces mayor en la recuperación externa que en la interna. Por otra parte, cuando un registro esta situado en un disco, se acostumbra no leer solo una palabra sino leer una gran pagina o bloque de información al mismo tiempo. Los tamaños normales de los bloques oscilan entre 256 y 1024 caracteres o palabras.

Nuestra meta en la búsqueda externa ha de ser minimizar el numero de accesos al disco, puesto que cada uno tarda mucho en comparación con la operación interna. Con cada acceso obtenemos un bloque que puede tener espacio para varios registros. Usándolos podemos tomar una decisión multimodal acerca del bloque al que se accederá a continuación. De ahí que los arboles multimodales sean sumamente adecuados en la búsqueda externa

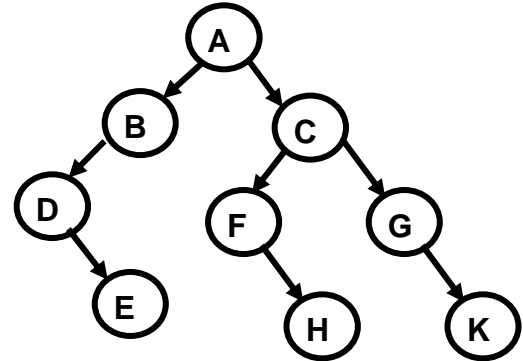
Robert L. Kruse (1988) *Estructura de Datos y Diseño de Programas*. México D.F. Prentice-Hall Hispanoamericana S.A. p. 384.



### Autoevaluación

1. Según el árbol siguiente diga usted cual alternativa es correcta:

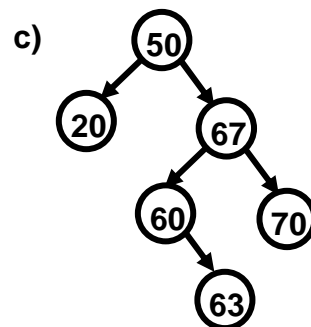
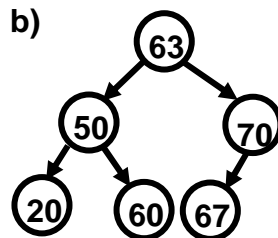
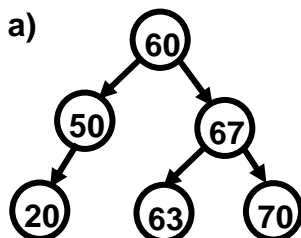
- a)
  - rama: B, D, C, F, H
- b) rama: E, H, K, D
- c) rama: B, F, C, A, G, D  
hoja : E, H, K
- d) rama: C, G, F, B, D



2. Diga usted cual de los recorridos son correctos para el árbol de la pregunta 1:

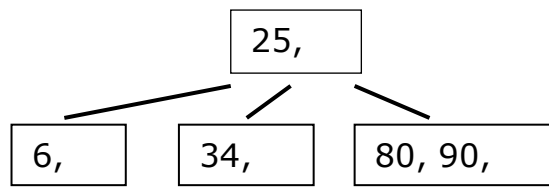
- a) PREORDEN: ABDEC FHGK  
INORDEN: DEBFHCGKA  
POSTORDEN: EDBHFKGCA
- b) PREORDEN: ABDCEHFGK  
INORDEN: DEBAFHCGK  
POSTORDEN: EDBHFCGKA
- c) PREORDEN: ABDEC FHGK  
INORDEN: DEBAFHCGK  
POSTORDEN: EDBHFKGCA
- d) ninguna

3. Dada la secuencia de numeros que se ingresan en este orden: 50, 20, 70, 60, 67, 63 para crear un árbol AVL , señale cual de las alternativas es la correcta:



d) ninguna

4. Dado el árbol B de orden 5 se desea eliminar la clave 10. Grafique el árbol después de eliminar la clave 6.



- a)
- 
- ```
graph TD; A["25,"] --- B["10"]; A --- C["34,"]; A --- D["80, 90,"];
```
- b)
- 
- ```
graph TD; A["70, 90"] --- B["10,25,34,"]; A --- C["80"]; A --- D["100"];
```
- c)
- 
- ```
graph TD; A["70"] --- B["10,25, 34,"]; A --- C["80,90,"];
```
- d) ninguna

Claves: 1:d; 2:c; 3:a; 4:c;



### Enlaces

<http://structio.sourceforge.net/guias/arbgraf/arbgraf.html>  
[http://www.matediscreta.8k.com/grafos\\_y\\_arboles.htm](http://www.matediscreta.8k.com/grafos_y_arboles.htm)  
<http://www.ccg.unam.mx/~contrera/bioinfoPerl/node56.html>  
<http://www.scribd.com/doc/24062352/Arboles-y-Grafos>

### Bibliografía

Tenenbaum A. M., Langsam Y., Augenstein, M.A., (1993) *Estructura de Datos en C*, 2ª. Ed., México D.F., Prentice Hall Hispanoamericana S.A. Cap. 5: Árboles pp. 249-327.

Joyanes Aguilar, L., Sanchez Garcia, L., Fernandez Azuela, M. Zahonero Martinez, I., (2005) *Estructura de Datos en C*. Madrid, McGraw-Hill - Interoamericana. Cap. 13: Árboles, árboles binarios y árboles ordenados pp. 236-312.

Hernández, R., Lazaro, J.C., Dormido, R., Ros, S. (2006) *Estructura de Datos y Algoritmos*. 2ª. Ed. Madrid. Prentice-Hall. Cap. 5: Árboles pp. 210-245.

Florez Rueda, R. (2005) *Algoritmos, Estructuras de Datos y Programación Orientada a Objetos*. Bogotá. ECOE Ediciones Ltda. Cap. 13: Grafos pp. 269-292.

Kolman B., Busby R. C., Ross S. (1995) *Estructuras de Matemáticas Discretas para la Computación*. 2ª. Ed. México D.F., Prentice-Hall Hispanoamericana S.A. Cap. 8: Árboles pp. 286-321.



## Glosario

|                                  |                                                                                                                                                                                                                                                                        |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Tipos abstractos de datos (TAD): | Llamado también tipos compuesto, es una herramienta que permite describir una estructura de datos como un conjunto de variables junto con las operaciones que sobre el se pueden aplicar.                                                                              |
| Objeto:                          | Un objeto es una entidad que engloba en sí mismo atributos (datos) y métodos (procedimientos o funciones) necesarios para el tratamiento de esos datos.                                                                                                                |
| Clase:                           | Una clase se puede considerar como un patrón para construir objetos. Los atributos y los métodos comunes a un conjunto de objetos forman un conjunto que se conoce como clase.                                                                                         |
| Archivo:                         | Permite almacenar datos en forma permanente, normalmente en los dispositivos de almacenamiento secundario (discos, cintas, disquetes, usb, etc.).                                                                                                                      |
| Arreglo:                         | Es un conjunto finito de elementos, todos del mismo tipo, que son representados mediante una variable del mismo nombre, y se registran en posiciones consecutivas de memoria.                                                                                          |
| Lista enlazada:                  | Es una serie de registros que no necesariamente son adyacentes en memoria. Cada registro contiene el elemento y un apuntador a un registro que contiene su sucesor. A este se le llama el apuntador siguiente. El apuntador siguiente de la última celda apunta a nil. |
| Puntero:                         | Un puntero es una variable que almacena la dirección de memoria de un elemento determinado. Para trabajar con punteros C++ dispone de los                                                                                                                              |





operadores \* y &.

Lista lineal doblemente enlazada:

Una lista doblemente ligada es una colección de nodos, en la cual cada nodo tiene dos punteros, uno de ellos apuntando a su predecesor y otro a su sucesor.

Lista circular:

Las listas circulares tienen la característica de que el último elemento de la misma apunta al primero.

Pila:

La pila se utiliza siempre que se quiera recuperar una serie de elementos en orden inverso a como fueron introducidos. La eliminación de un elemento de una pila se realiza por la parte superior, lo mismo que la inserción, la pila también se conoce con el nombre de LIFO.

Cola:

En una cola a diferencia de una pila la operación de ingreso o encolar se hace por un extremo, según se observa en la grafica se hace por el lado izquierdo o final de la cola y la operación retirar o desencolar se hace por el otro extremo denominado frente o principio de la cola. El primer elemento que entra en la cola será el primero en salir, debido a esta característica la cola también recibe el nombre de estructuras FIFO.

Árboles:

Estructuras que tienen una jerarquía sobre sus elementos y que no tienen una conformación lineal sino mas bien no lineal en el que cada elemento tiene enlaces o apuntadores a uno o varios elementos.

Nodo:

Es el término usado para denotar a los elementos o vértices de un árbol.

Raíz:

Es el nodo a partir del cual se derivan o descienden otros nodos y es el único nodo que no tiene padre ya que no desciende de ningún nodo. Por ejemplo en el árbol de la figura 1, A es el nodo



|                        |                                                                                                                                                                                               |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                        | raíz.                                                                                                                                                                                         |
| Nodo hoja:             | Se denomina así al nodo que no tiene hijos o que no tiene como descendiente a ningún sub árbol. Por ejemplo en el árbol de la figura 1, K, F, L, M, N, O, H y I son nodos hojas.              |
| Nodos rama o internos: | Se refiere a los nodos que no son nodos hojas y que no son la raíz del árbol. Por ejemplo en el árbol de la figura 1, los nodos B, E, C, G, D, y J son rama                                   |
| Nodo hijo:             | Se refiere a un nodo que es apuntado por otro nodo del árbol. Por ejemplo en el árbol de la figura 1, E y F son nodos hijos de B y B, C y D son nodos hijos de A.                             |
| Nodo padre:            | Es el nodo que tiene uno más enlaces o apuntadores hacia otros nodos. Por ejemplo en el árbol de la figura 1, A es padre de B, C y D, y D es padre de H, I, J.                                |
| Nodos hermanos:        | Son los nodos que son apuntados por un mismo nodo que se conoce como nodo padre. Por ejemplo en el árbol de la figura 1, L y M son hermanos, al igual que H, I y J.                           |
| Bosque:                | Se denomina a una colección o conjunto de dos o más árboles.                                                                                                                                  |
| Grado:                 | Es el mayor número de hijos que tiene un nodo dentro del árbol. Por ejemplo en el árbol de la figura 1, el árbol es de grado 3 ya que el nodo A y el nodo D apuntan como máximo a tres nodos. |
| Nivel:                 | Es la abstracción de que un nodo corresponde a un nivel y que esta dado por la longitud de camino o la distancia del nodo específico a la raíz, medida en nodos.                              |
| Altura:                | Es el nivel del nodo de mayor nivel mas uno. También puede aplicarse para la                                                                                                                  |



altura de las ramas, considerando que cada nodo a su vez es la raíz de un árbol. El árbol del ejemplo tiene altura 4, la rama 'B' tiene altura 3, la rama 'G' tiene altura 2, la 'H' tiene altura 1, etc.

Árbol binario:

Es un conjunto finito de elementos que o está vacío o está dividido en tres subconjuntos desarticulados. El primer subconjunto contiene un solo elemento llamado raíz del árbol. Los otros dos son en sí mismos árboles binarios, llamados subárboles izquierdo y derecho del árbol original.

Árbol AVL:

El término árbol AVL, es una palabra que concatena las letras iniciales de los nombres de los creadores de este tipo particular de árbol los cuales son Adelson-Velskii y Landis.

Árbol B:

Un árbol B es también un ABB, ósea un árbol de búsqueda con ciertas limitaciones que solucionan problemas en un ABB simple y que garantizan que el árbol B estará en todo momento balanceado y que el desperdicio de espacio producto de eliminaciones si las hubieran no resultara excesivo.

Grafo:

Es un conjunto de vértices o nodos y aristas o también denominados arcos. Cada arco es una línea que enlaza dos vértices del grafo o también dirigirse o enlazarse a sí mismo.



### Bibliografía

Tenenbaum A. M., Langsam Y., Augenstein, M.A., (1993) *Estructura de Datos en C*, 2ª Ed. México D.F., Prentice Hall Hispanoamericana S.A.

Joyanes Aguilar, L., Sanchez Garcia, L., Fernandez Azuela, M. Zahonero Martinez, I., (2005) *Estructura de Datos en C*, Madrid, Mc Graw-Hill - Interoamericana.

Hernandez, R., Lazaro, J.C., Dormido, R., Ros, S. (2006) *Estructura de Datos y Algoritmos*, 2ª Ed. México D.F., Prentice Hall Hispanoamericana S.A.

Schildt H., (1994) *Turbo C/C++ Manual de Referencia, Una información completa ideal para todo usuario de Turbo C/C++*, Madrid, Mc Graw-Hill/Interamericana.

Lopez R. L., (2006) *Metodología de la Programación Orientada a Objetos*, México D.F., Alfaomega, Grupo Editor S.A.

Allen Weiss, M., (1995) *Estructura de Datos y Algoritmos*, México D.F., Addison-Wesley Iberoamericana.

Tenenbaum A. M., Langsam Y., Augenstein, M.A., (1986) *Estructura de Datos en Pascal*, México D.F., Prentice Hall Hispanoamericana S.A.

Brassard, G., Bratley, P., (1998) *Fundamentos de Algoritmia*, 2ª. Ed. México D.F., Prentice Hall.

Cairó O., Guardati M.C. S., (2002) *Estructura de Datos*, 2ª. Ed. México D.F., Mc Graw-Hill / Interamericana Editores S.A.

Larry R. Nyhoff (2006) *Estructuras de datos y resolución de problemas con C++*, Madrid, Prentice Hall.

Kruse, Robert L. (1988) *Estructura de Datos y Diseño de Programas*. México D.F. Prentice-Hall Hispanoamericana S.A.

Florez Rueda, R. (2005) *Algoritmos, Estructuras de Datos y Programación Orientada a Objetos*. Bogotá. ECOE Ediciones Ltda.

Kolman B., Busby R. C., Ross S. (1995) *Estructuras de Matemáticas Discretas para la Computación*. 2ª. Ed. México D.F., Prentice-Hall Hispanoamericana S.A.