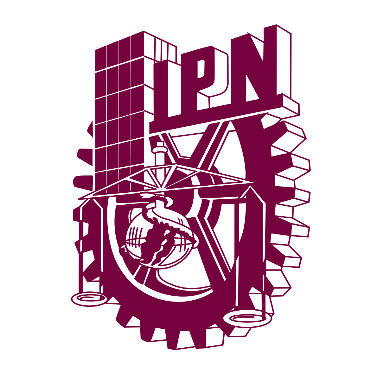
****

**Instituto Politécnico Nacional**

**Escuela Superior de Computo**

*Alumno:*

* Monroy Ramírez Oscar G.

***Grupo:*** *2CM23*

***Unidad de Aprendizaje:*** *Algoritmos y Estructuras de Datos*

***Evidencia:*** *Proyecto De Arboles MINIMAX*

***Docente:*** *De Luna Caballero Roberto*

***Fecha****: 13 de enero de 2021*

Contenido

[Introducción 4](#_Toc59050266)

[Marco Teórico 5](#_Toc59050267)

[Recursividad 5](#_Toc59050268)

[Ejemplo sencillo 5](#_Toc59050269)

[Arboles Binarios (Arboles B) 7](#_Toc59050270)

[Tipos de árboles binarios 7](#_Toc59050271)

[Implementación en C 8](#_Toc59050272)

[Estructura con manejo de memoria dinámica, siendo el puntero que apunta al árbol de tipo Arbol: 8](#_Toc59050273)

[Estructura con arreglo indexado: 8](#_Toc59050274)

[Recorridos sobre árboles binarios 8](#_Toc59050275)

[Recorridos en profundidad 8](#_Toc59050276)

[Recorrido en postorden 9](#_Toc59050277)

[Recorrido en inorden 9](#_Toc59050278)

[Recorridos en amplitud (o por niveles) 10](#_Toc59050279)

[Creación de árboles a partir de los recorridos 11](#_Toc59050280)

[Métodos para almacenar Árboles Binarios 11](#_Toc59050281)

[Codificación de árboles n-arios como árboles binarios 12](#_Toc59050282)

[Arboles Binarios de búsqueda (Arboles ABB) 13](#_Toc59050283)

[ANÁLISIS DE LA EFICIENCIA DE LAS OPERACIONES. 16](#_Toc59050284)

[Ejemplo de ARBOL ABB programado por OGMR 18](#_Toc59050285)

[Operación del programa (Inserción Manual): 36](#_Toc59050286)

[Conclusion. 38](#_Toc59050292)

[Referencias Bibliográficas: 38](#_Toc59050293)

# Introducción

Una computadora es una maquina que maneja información. El estudio de la ciencia de la computación incluye saber la forma en que se organiza la información en una computadora, como puede manejarse y la manera en que puede utilizarse esa información. Por tanto, es de suma importancia para un estudiante de la computación, entender los conceptos de organización y manejo de la información para continuar con el estudio de nuestra diciplina.

Si la ciencia de la computación es fundamental el estudio de la información, la primera pregunta que surge es: ¿qué es la información? Por desgracia, aunque ese concepto es la piedra angular de la diciplina, la interrogante anterior no puede contestar con precisión. En que sentido, el concepto de la información en esta diciplina es similar a los conceptos de punto, recta y plano para la geometría: son un grupo de términos indefinidos con los que se pueden hacer proposiciones, pero no pueden definirse con conceptos más elementales.

En la geometría es posible hablar de la longitud de una recta, a pesar de que el concepto de recta es en si mismo indefinido. La longitud de una recta es una medida cuantitativa. De manera similar, en la ciencia de la computación podemos hablar de cantidades de información. La unidad básica de información es el ***bit*** cuyo valor confirma una de dos posibilidades excluyentes. Por ejemplo, un conmutador puede estar en una de dos posiciones pero no ambas al mismo tiempo pero el hecho es que al estar en una posición representa un bit.

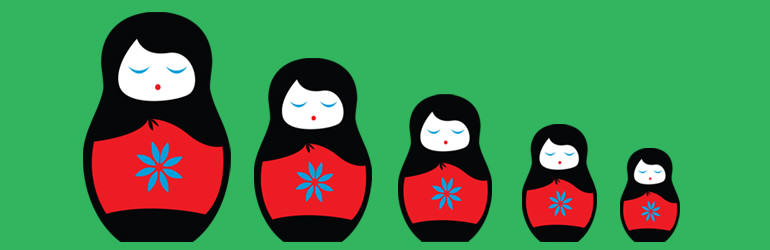
Pues asi es la parte estructurada de la computacion, Que nos permite encontrar diversas formas métodos, con los cual es podemos avanzar en diversas partes yasea para usarlas como herramientas que funcionen como grandes precursores en todas las operaciones básicas que realizamos a la hora de realizar diferentes operaciones en nuestros propios programas, siendo así la estructura de datos una manera de construir un mapa para no perderse a la hora de El manejo de esa intensa y extensa información, métodos que veremos a lo largo del desarrollo de este trabajo.

La búsqueda en árboles es un método de búsqueda simple, dinámico y eficiente considerado como uno de los fundamentales en Ciencia de la Computación. De toda la terminología sobre árboles,tan sólo recordar que la propiedad que define un árbol binario es que cada nodo tiene a lo más un hijo a la izquierda y uno a la derecha.Para construir los algoritmos consideraremos que cada nodo contiene un registro con un valor clave a través del cual efectuaremos las búsquedas.En las implementaciones que presentaremos sólo se considerará en cada nodo del árbol un valor del tipo *de elemento* aunque en un caso general ese tipo estará compuesto por dos:una clave indicando el campo por el cual se realiza la ordenación y una información asociada a dicha clave o visto de otra forma,una información que puede ser compuesta en la cual existe definido un orde

# 

# Marco Teórico

## Recursividad



Es una técnica utilizada en programación que nos permite que un bloque de instrucciones se ejecute un cierto número de veces (el que nosotros determinemos). A veces es algo complicado de entender, pero no os preocupéis. Cuando veamos los ejemplos estará clarísimo. En Java, como en otros muchos lenguajes, los métodos pueden llamarse a sí mismos. Gracias a esto, podemos utilizar a nuestro favor la recursividad en lugar de la iteración para resolver determinados tipos de problemas. **[1]:**

### Ejemplo sencillo

Vamos a ver un pequeño ejemplo que no hace absolutamente nada. Es un método cuyo único objetivo es llamarse a sí mismo: **[1]:**

void cuentaRegresiva () {

cuentaRegresiva();

}

 Si ejecutáis esto, os va a dar un error en la pila (mítico StackOverflow Error, Biblia de los programadores).

Como podemos ver, se ha llamado al método **cuentaRegresiva**porque vamos a mostrar por pantalla la cuenta atrás de un número que nosotros pasemos como parámetro a la función. Por ejemplo, para hacer la cuenta atrás de 10 sin recursividad, haríamos: **[1]:**

for( int i = 10; i >= 0; i--) {

System.out.println(i);

}

 Ahora, para hacerlo de manera recursiva, tendríamos que pasar como parámetro un número. Además, tras imprimir ese número, llamaremos a la misma función con el número actual restando uno: **[1]:**

void cuentaRegresiva(int numero) {

System.out.println(numero);

cuentaRegresiva(numero - 1);

}

Es lo que os he comentado arriba. Llamamos a la función con un 10. Imprimimos el 10 y llamamos a la función con un 9. Imprimimos el 9 y llamamos a la función con un 8. Así hasta el fin de los días. Digo hasta el fin de los días porque os va a saltar error si ejecutáis esto así directamente: **[1]:**

public class Recursividad {

static void cuentaRegresiva(int numero) {

System.out.println(numero);

cuentaRegresiva(numero - 1);

}

public static void main(String[] args) {

cuentaRegresiva(10);

}

}

 Problema: llamada infinita. Para ello, lo que tenemos que hacer es que cuando el número sea 0, deje de llamar a la función. Para eso, metemos una [estructura condicional](http://geekytheory.com/tutorial-3-java-estructuras-condicionales-y-excepciones/) de toda la vida. Gracias al condicional, dejará de ejecutarse a partir de 0: **[1]:**

void cuentaRegresiva(int numero) {

System.out.println(numero);

if(numero > 0) {

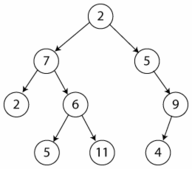
cuentaRegresiva(numero - 1);

}

}

# Arboles Binarios (Arboles B)

En [ciencias de la computación](https://es.wikipedia.org/wiki/Ciencias_de_la_computaci%C3%B3n), un árbol binario es una [estructura de datos](https://es.wikipedia.org/wiki/Estructura_de_datos) en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre "binario"). Si algún hijo tiene como referencia a null, es decir que no almacena ningún dato, entonces este es llamado un nodo externo. En el caso contrario el hijo es llamado un nodo interno. Usos comunes de los árboles binarios son los [árboles binarios de búsqueda](https://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda), los [montículos binarios](https://es.wikipedia.org/wiki/Mont%C3%ADculo_binario) y [Codificación de Huffman](https://es.wikipedia.org/wiki/Codificaci%C3%B3n_de_Huffman).**[2]**

[](https://commons.wikimedia.org/wiki/File:Binary_tree_(oriented_digraph).png)

Un árbol binario sencillo de tamaño 9, 3 niveles (nivel 0 hasta nivel 3) y altura 3 (altura = máximo nivel), con un nodo raíz cuyo valor es 2.

En [teoría de grafos](https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos), se usa la siguiente definición: «Un árbol binario es un [grafo conexo](https://es.wikipedia.org/wiki/Grafo_conexo), [acíclico](https://es.wikipedia.org/wiki/Grafo_ac%C3%ADclico) y [no dirigido](https://es.wikipedia.org/wiki/Grafo_no_dirigido) tal que el [grado](https://es.wikipedia.org/wiki/Grado_(teor%C3%ADa_de_grafos)) de cada [vértice](https://es.wikipedia.org/wiki/V%C3%A9rtice_(teor%C3%ADa_de_grafos)) no es mayor a 2». De esta forma solo existe un camino entre un par de nodos.

Un árbol binario con enraizado es como un grafo que tiene uno de sus vértices, llamado *raíz*, de grado no mayor a 2. Con la raíz escogida, cada vértice tendrá un único padre, y nunca más de dos hijos. Si rehusamos el requerimiento de la conectividad, permitiendo múltiples componentes conectados en el grafo, llamaremos a esta última estructura un *bosque.****[2]***

## Tipos de árboles binarios

Un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos subárboles. En un árbol binario cada nodo puede tener cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como hijo izquierdo y el nodo de la derecha como hijo derecho. **[2]**

Existen tipos de árboles binarios que suelen usarse para fines específicos, como:

[Árbol binario de búsqueda](https://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda)

[Árbol de Fibonacci](https://es.wikipedia.org/wiki/%C3%81rbol_de_Fibonacci)

## Implementación en C

Un árbol binario puede declararse de varias maneras. Algunas de ellas son:

### Estructura con manejo de memoria dinámica, siendo el puntero que apunta al árbol de tipo Arbol:

**typedef** **struct** **nodo** {

int clave;

**struct** **nodo** \*izdo, \*dcho;

}Nodo;

### Estructura con [arreglo](https://es.wikipedia.org/wiki/Arreglo) indexado:

typedef struct tArbol

{

int clave;

tArbol hIzquierdo, hDerecho;

} tArbol;

tArbol árbol[NUMERO\_DE\_NODOS];

En el caso de un árbol binario casi-completo (o un árbol completo), puede utilizarse un sencillo arreglo de enteros con tantas posiciones como nodos deba tener el árbol. La información de la ubicación del nodo en el árbol es implícita a cada posición del arreglo. Así, si un nodo está en la posición *i*, sus hijos se encuentran en las posiciones 2*i*+1 y 2*i*+2, mientras que su padre (si tiene), se encuentra en la posición [truncamiento](https://es.wikipedia.org/wiki/Truncamiento)((*i*-1)/2) (suponiendo que la raíz está en la posición cero). Este método se beneficia de un almacenamiento más compacto y una mejor localidad de referencia, particularmente durante un recorrido en preorden. La estructura para este caso sería por tanto:

int árbol[NUMERO\_DE\_NODOS];

# Recorridos sobre árboles binarios

## Recorridos en profundidad

El método de este recorrido es tratar de encontrar de la cabecera a la raíz en nodo de unidad binaria. Ahora pasamos a ver la implementación de los distintos recorridos:

#### Recorrido en preorden

En este tipo de recorrido se realiza cierta acción (quizás simplemente imprimir por pantalla el valor de la clave de ese nodo) sobre el nodo actual y posteriormente se trata el subárbol izquierdo y cuando se haya concluido, el subárbol derecho. Otra forma para entender el recorrido con este método seria seguir el orden: nodo raíz, nodo izquierda, nodo derecha.

En el árbol de la figura el recorrido en preorden sería: 2, 7, 2, 6, 5, 11, 5, 9 y 4.

void preorden(tArbol \*a)

{

if (a != NULL) {

tratar(a); //Realiza una operación en nodo

preorden(a->hIzquierdo);

preorden(a->hDerecho);

}

}

#### Implementación en pseudocódigo de forma iterativa:

push(s,NULL); //insertamos en una pila (stack) el valor NULL, para asegurarnos de que esté vacía

push(s,raíz); //insertamos el nodo raíz

MIENTRAS (s <> NULL) HACER

p = pop(s); //sacamos un elemento de la pila

tratar(p); //realizamos operaciones sobre el nodo p

SI (D(p) <> NULL) //preguntamos si p tiene árbol derecho

ENTONCES push(s,D(p));

FIN-SI

SI (I(p) <> NULL) //preguntamos si p tiene árbol izquierdo

ENTONCES push(s,I(p));

FIN-SI

FIN-MIENTRAS

### Recorrido en postorden

En este caso se trata primero el subárbol izquierdo, después el derecho y por último el nodo actual. Otra forma para entender el recorrido con este método seria seguir el orden: nodo izquierda, nodo derecha, nodo raíz. En el árbol de la figura el recorrido en postorden sería: 2, 5, 11, 6, 7, 4, 9, 5 y 2.

void postorden(tArbol \*a)

{

if (a != NULL) {

postorden(a->hIzquiedo);

postorden(a->hDerecho);

tratar(a); //Realiza una operación en nodo

}

}

### Recorrido en inorden

En este caso se trata primero el subárbol izquierdo, después el nodo actual y por último el subárbol derecho. En un [ABB](https://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda) este recorrido daría los valores de clave ordenados de menor a mayor. Otra forma para entender el recorrido con este método seria seguir el orden: nodo izquierda, nodo raíz, nodo derecha. En el árbol de la figura el recorrido en inorden sería: 2, 7, 5, 6, 11, 2, 5, 4, 9.

Esquema de implementación:

void inorden(tArbol \*a)

{

if (a != NULL) {

inorden(a->hIzquierdo);

tratar(a); //Realiza una operación en nodo

inorden(a->hDerecho);

}

}

## Recorridos en amplitud (o por niveles)

En este caso el recorrido se realiza en orden por los distintos niveles del árbol. Así, se comenzaría tratando el nivel 1, que solo contiene el nodo raíz, seguidamente el nivel 2, el 3 y así sucesivamente. En el árbol de la figura el recorrido en amplitud sería: 2, 7, 5, 2, 6, 9, 5, 11 y 4.

Al contrario que en los métodos de recorrido en profundidad, el recorrido por niveles no es de naturaleza recursiva. Por ello, se debe utilizar una cola para recordar los subárboles izquierdos y derecho de cada nodo.

El esquema algoritmo para implementar un recorrido por niveles es exactamente el mismo que el utilizado en la versión iterativa del recorrido en preorden pero cambiando la estructura de datos que almacena los nodos por una cola.

Implementación en C:

void arbol\_recorrido\_anch (tipo\_Arbol\* A) {

tipo\_Cola cola\_nodos; *// esta cola esta implementada previamente, almacena punteros (posiciones de nodos de árbol)*

tipo\_Pos nodo\_actual; *// este es un puntero llevara el recorrido*

**if** (vacio(A)) *// si el árbol esta vacio, salimos*

**return**;

cola\_inicializa(&cola\_nodos); *// obvio, y necesario*

cola\_enqueue(A, &cola\_nodos); *// se encola la raíz*

**while** (!vacia(&cola\_nodos)) { *// mientras la cola no se vacie se realizara el recorrido*

nodo\_actual = cola\_dequeue(&cola\_nodos) *// de la cola saldran los nodos ordenados por nivel*

printf("%c,", nodo\_actual->info); *// se "procesa" el nodo donde va el recorrido, en este caso se imprime*

**if** (nodo\_actual->izq != null) *// si existe, ponemos el hijo izquierdo en la cola*

cola\_enqueue(nodo\_actual->izq, &cola\_nodos);

**if** (nodo\_actual->der != null) *// si existe, ponemos el hijo derecho en la cola*

cola\_enqueue(nodo\_actual->der, &cola\_nodos);

} *// al vaciarse la cola se han visitado todos los nodos del árbol*

}

## Creación de árboles a partir de los recorridos

Para poder dibujar un árbol binario sobre la base de los recorridos, se necesitan por lo menos dos de los recorridos de profundidad (en caso de que no se repitan los nodos, ya que si se repiten los nodos es recomendable tener los tres recorridos), ya sean inorden y preorden o inorden y postorden, la única diferencia entre usar el recorrido en preorden o postorden es que en preorden se usa el primer nodo para encontrar la raíz y en postorden se usa el último nodo.

El método consiste en ir dividiendo los recorridos del árbol en pequeños subárboles, se va encontrando la raíz con el preorden o postorden y se divide en dos subárboles basándonos en el recorrido en inorden. En el caso de que los nodos se repitan es conveniente tener los 3 recorridos para identificar más fácilmente cuál de los nodos es la raíz actual.

Para encontrar la raíz es necesario tener el recorrido preorden o postorden, ya que la raíz es el primer nodo o el último nodo respectivamente. En este caso la raíz es el {\displaystyle 2}.

Una vez encontrada la raíz, es necesario saber su posición en el recorrido inorden, del paso anterior se tiene el nodo {\displaystyle 2}, pero existen 2 nodos con ese valor, el primero y el de en medio. Si el primer dos es la raíz, entonces no existe ninguna rama del lado izquierdo, en ese caso la siguiente raíz de acuerdo con el recorrido en postorden es {\displaystyle 5}y de acuerdo con preorden es {\displaystyle 7}, lo cual es una incongruencia, de esa forma sabemos que el otro {\displaystyle 2} es la raíz.

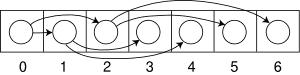
El recorrido inorden, es un recorrido de los árboles binarios en los que se empieza desde el nodo que se encuentra más a la izquierda de todos, sigue con la raíz y termina con los nodos del lado derecho, entonces, como en el recorrido inorden ya encontramos la raíz, la parte izquierda representa el subárbol izquierdo y la parte derecha representa el subárbol derecho.

En los recorridos tenemos 5 nodos a la izquierda del {\displaystyle 2} y a la derecha se encuentran 3 valores, entonces podemos crear los recorridos para el subárbol izquierdo y el subárbol derecho

# Métodos para almacenar Árboles Binarios

Los árboles binarios pueden ser construidos a partir de [lenguajes de programación](https://es.wikipedia.org/wiki/Lenguajes_de_programaci%C3%B3n) de varias formas. En un lenguaje con [registros](https://es.wikipedia.org/wiki/Registro_(hardware)) y [referencias](https://es.wikipedia.org/wiki/Referencias), los árboles binarios son construidos típicamente con una estructura de nodos y punteros en la cual se almacenan datos, cada uno de estos nodos tiene una referencia o puntero a un nodo izquierdo y a un nodo derecho denominados hijos. En ocasiones, también contiene un puntero a un único nodo. Si un nodo tiene menos de dos hijos, algunos de los punteros de los hijos pueden ser definidos como nulos para indicar que no dispone de dicho nodo. En la figura adjunta se puede observar la estructura de dicha implementación.

Los árboles binarios también pueden ser almacenados como una [estructura de datos implícita](https://es.wikipedia.org/w/index.php?title=Estructura_de_datos_impl%C3%ADcita&action=edit&redlink=1) en [vectores](https://es.wikipedia.org/wiki/Vector_(programaci%C3%B3n)), y si el árbol es un árbol binario completo, este método no desaprovecha el espacio en memoria. Tomaremos como notación la siguiente: si un nodo tiene un índice i, sus hijos se encuentran en índices 2i + 1 y 2i + 2, mientras que sus padres (si los tiene) se encuentra en el índice {\displaystyle \left\lfloor {\frac {i-1}{2}}\right\rfloor } (partiendo de que la raíz tenga índice cero). Este método tiene como ventajas el tener almacenados los datos de forma más compacta y por tener una forma más rápida y eficiente de localizar los datos en particular durante un preoden transversal. Sin embargo, desperdicia mucho espacio en memoria.

[](https://commons.wikimedia.org/wiki/File:Lista_nodos.JPG)

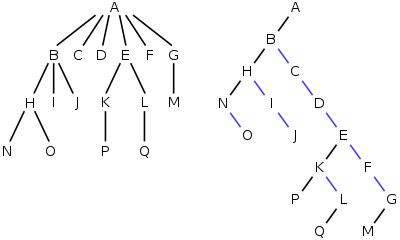
## Codificación de árboles n-arios como árboles binarios

Hay un mapeo uno a uno entre los árboles generales y árboles binarios, el cual en particular es usado en Lisp para representar árboles generales como árboles binarios. Cada nodo N ordenado en el árbol corresponde a un nodo N' en el árbol binario; el hijo de la izquierda de N’ es el nodo correspondiente al primer hijo de N, y el hijo derecho de N' es el nodo correspondiente al siguiente hermano de N, es decir, el próximo nodo en orden entre los hijos de los padres de N.

Esta representación como árbol binario de un árbol general, se conoce a veces como un árbol binario primer hijo hermano, o un árbol doblemente encadenado.

Una manera de pensar acerca de esto es que los hijos de cada nodo estén en una lista enlazada, encadenados junto con el campo derecho, y el nodo solo tiene un puntero al comienzo o la cabeza de esta lista, a través de su campo izquierdo.

Por ejemplo, en el árbol de la izquierda, la A tiene 6 hijos (B, C, D, E, F, G). Puede ser convertido en el árbol binario de la derecha:

[](https://commons.wikimedia.org/wiki/File:N-ary_to_binary.svg)

El árbol binario puede ser pensado como el árbol original inclinado hacia los lados, con los bordes negros izquierdos representando el primer hijo y los azules representado los siguientes hermanos.

Las hojas del árbol de la izquierda serían escritas en [Lisp](https://es.wikipedia.org/wiki/Lisp) como:

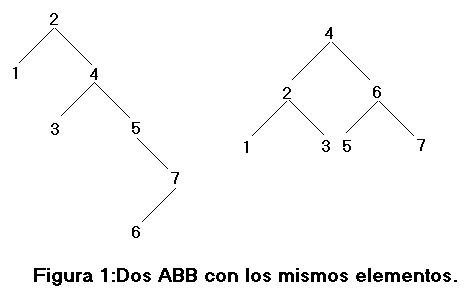
(((N O) I J) C D ((P) (Q)) F (M))

Que se ejecutará en la memoria como el árbol binario de la derecha, sin ningún tipo de letras en aquellos nodos que tienen un hijo izquierdo.

# Arboles Binarios de búsqueda (Arboles ABB)

Un árbol binario de búsqueda(ABB) es un árbol binario con la propiedad de que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo *x* son menores que el elemento almacenado en *x* ,y todos los elementos almacenados en el subárbol derecho de *x* son mayores que el elemento almacenado en *x*.

La figura 1 muestra dos ABB construidos en base al mismo conjunto de enteros:[3]



Obsérvese la interesante propiedad de que si se listan los nodos del ABB en inorden nos da la lista de nodos ordenada.Esta propiedad define un método de ordenación similar al Quicksort,con el nodo raíz jugando un papel similar al del elemento de partición del Quicksort aunque con los ABB hay un gasto extra de memoria mayor debido a los punteros.La propiedad de ABB hace que sea muy simple diseñar un procedimiento para realizar la búsqueda. Para determinar si *k* está presente en el árbol la comparamos con la clave situada en la raíz, *r*.Si coinciden la búsqueda finaliza con éxito, si *k*<*r* es evidente que *k*,de estar presente,ha de ser un descendiente del hijo izquierdo de la raíz,y si es mayor será un descendiente del hijo derecho.La función puede ser codificada fácilmente de la siguiente forma:[#]

#define ABB\_VACIO NULL

#define TRUE 1

#define FALSE 0

typedef int tEtiqueta /\*Algun tipo adecuado\*/

typedef struct tipoceldaABB{

struct tipoceldaABB \*hizqda,\*hdrcha;

tEtiqueta etiqueta;

}\*nodoABB;

typedef nodoABB ABB;

**ABB Crear(tEtiqueta et)**

**{**

ABB raiz;

raiz = (ABB)malloc(sizeof(struct tceldaABB));

if (raiz == NULL)

error("Memoria Insuficiente.");

raiz->hizda = NODO\_NULO;

raiz->hdcha = NODO\_NULO;

raiz->etiqueta = et;

return(raiz);

**}**

**int Pertenece(tElemento x,ABB t)**

**{**

if(!t)

return FALSE

else if(t->etiqueta==x)

return TRUE;

else if(t->etiqueta>x)

return pertenece(x,t->hizqda);

else return pertenece(x,t->hdrcha);

**}**

Es conveniente hacer notar la diferencia entre este procedimiento y el de búsqueda binaria.En éste podría pensarse en que se usa un árbol binario para describir la secuencia de comparaciones hecha por una función de búsqueda sobre el vector.En cambio en los ABB se construye una estructura de datos con registros conectados por punteros y se usa esta estructura para la búsqueda.[3]

El procedimiento de construcción de un ABB puede basarse en un procedimiento de inserción que vaya añadiendo elementos al árbol. Tal procedimiento comenzaría mirando si el árbol es vacío y de ser así se crearía un nuevo nodo para el elemento insertado devolviendo como árbol resultado un puntero a ese nodo.Si el árbol no está vacio se busca el elemento a insertar como lo hace el procedimiento *pertenece* sólo que al encontrar un puntero NULL durante la búsqueda,se reemplaza por un puntero a un nodo nuevo que contenga el elemento a insertar.[3]

El código podría ser el siguiente:

**void Inserta(tElemento x,ABB \*t)**

**{**

if(!(\*t)){

\*t=(nodoABB)malloc(sizeof(struct tipoceldaABB));

if(!(\*t)){

error("Memoria Insuficiente.");

}

(\*t)->etiqueta=x;

(\*t)->hizqda=NULL;

(\*t)->hdrcha=NULL;

} else if(x<(\*t)->etiqueta)

inserta(x,&((\*t)->hizqda));

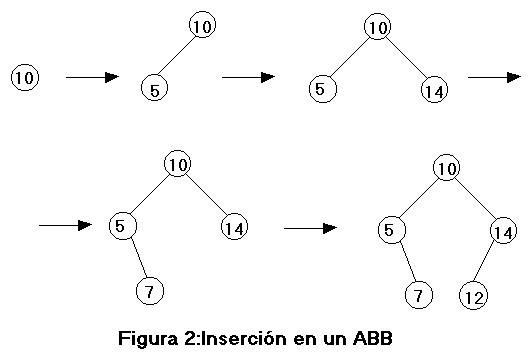
else

inserta(x,&((\*t)->hdrcha));

**}**

Por ejemplo supongamos que queremos construir un ABB a partir del conjunto de enteros *{10,5,14,7,12}* aplicando reiteradamente el proceso de inserción.

El resultado es el que muestra la figura 2.[3]



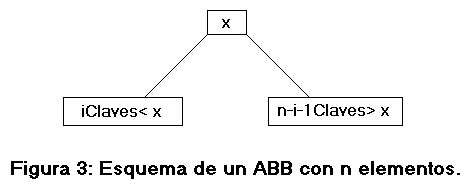
## ANÁLISIS DE LA EFICIENCIA DE LAS OPERACIONES.

Puede probarse que una búsqueda o una inserción en un ABB requiere *O(log2n)* operaciones en el caso medio,en un árbol construido a partir de *n* claves aleatorias, y en el peor caso una búsqueda en un ABB con *n* claves puede implicar revisar las *n* claves,o sea,es *O(n)*.

Es fácil ver que si un árbol binario con *n* nodos está completo (todos los nodos no hojas tienen dos hijos) y así ningún camino tendrá más de *1+log2n* nodos.Por otro lado,las operaciones *pertenece* e *inserta* toman una cantidad de tiempo constante en un nodo.Por tanto,en estos árboles, el camino que forman desde la raíz,la secuencia de nodos que determinan la búsqueda o la inserción, es de longitud *O(log2n)*,y el tiempo total consumido para seguir el camino es también *O(log2n)*.

Sin embargo,al insertar *n* elementos en un orden aleatorio no es seguro que se sitúen en forma de árbol binario completo.Por ejemplo,si sucede que el primer elemento(de *n* situados en orden) insertado es el más pequeño el árbol resultante será una cadena de *n* nodos donde cada nodo,excepto el más bajo en el árbol,tendrá un hijo derecho pero no un hijo izquierdo.En este caso,es fácil demostrar que como lleva *i* pasos insertar el i-ésimo elemento dicho proceso de *n* inserciones necesita  pasos o equivalentemente *O(n)* pasos por operación.

Es necesario pues determinar si el ABB *promedio* con *n* nodos se acerca en estructura al árbol completo o a la cadena,es decir,si el tiempo medio por operación es *O(log2n)*,*O(n)* o una cantidad intermedia.Como es difícil saber la verdadera frecuencia de inserciones sólo se puede analizar la longitud del camino promedio de árboles "aleatorios" adoptando algunas suposiciones como que los árboles se forman sólo a partir de inserciones y que todas las magnitudes de los *n* elementos insertados tienen igual probabilidad.Con esas suposiciones se puede calcular *P(n)*,el número promedio de nodos del camino que va de la raíz hacia algún nodo(no necesariamente una hoja).Se supone que el árbol se formó con la inserción aleatoria de *n* nodos en un árbol que se encontraba inicialmente vacío,es evidente que P(0)=0 y P(1)=1.Supongamos que tenemos una lista de *n>=2* elementos para insertar en un árbol vacío,el primer elemento de la lista,*x*,es igual de probable que sea el primero,el segundo o el n-ésimo en la lista ordenada.Consideremos que *i* elementos de la lista son menores que *x* de modo que *n-i-1* son mayores. Al construir el árbol,*x* aparecerá en la raíz,los *i* elementos más pequeños serán descendientes izquierdos de la raíz y los restantes *n-i-1* serán descendientes derechos.Esquemáticamente quedaría como muestra la figura 3.



Al tener tanto en un lado como en otro todos los elementos igual probabilidad se espera que los subárboles izqdo y derecho de la raíz tengan longitudes de camino medias *P(i)* y *P(n-i-1)* respectivamente. Como es posible acceder a esos elementos desde la raíz del árbol completo es necesario agregar 1 al número de nodos de cada camino de forma que para todo *i* entre 0 y n-1,*P(n)* puede calcularse obteniendo el promedio de la suma:



El primer término es la longitud del camino promedio en el subárbol izquierdo ponderando su tamaño. El segundo término es la cantidad análoga del subárbol derecho y el término *1/n* representa la contribución de la raíz. Al promediar la suma anterior para todo *i* entre 1 y *n* se obtiene la recurrencia:



y con unas transformaciones simples podemos ponerla en la forma:



y el resto es demostrar por inducción sobre *n* que *P(n)<=1+4log2n*.

En consecuencia el tiempo promedio para seguir un camino de la raíz a un nodo aleatorio de un ABB construido mediante inserciones aleatorias es *O(log2n)*.Un análisis más detallado demuestra que la constante 4 es en realidad una constante cercana a 1.4.

De lo anterior podemos concluir que la prueba de pertenencia de una clave aleatoria lleva un tiempo *O(log2n)*.Un análisis similar muestra que si se incluyen en la longitud del camino promedio sólo aquellos nodos que carecen de ambos hijos o solo aquellos que no tienen hijo izqdo. o derecho también la longitud es *O(log2n)*.

Terminaremos este apartado con algunos comentarios sobre los borrados en los ABB. Es evidente que si el elemento a borrar está en una hoja bastaría eliminarla, pero si el elemento está en un nodo interior, eliminándolo, podríamos desconectar el árbol. Para evitar que esto suceda se sigue el siguiente procedimiento: si el nodo a borrar *u* tiene sólo un hijo se sustituye *u* por ese hijo y el ABB quedar&aacue; construido. Si *u* tiene dos hijos, se encuentra el menor elemento de los descendientes del hijo derecho(o el mayor de los descendientes del hijo izquierdo) y se coloca en lugar de *u*, de forma que se continúe manteniendo la propiedad de ABB.

# Arbol AVL

El árbol AVL toma su nombre de las iniciales de los apellidos de sus inventores, [Georgii Adelson-Velskii](https://es.wikipedia.org/wiki/Georgii_Adelson-Velskii) y [Yevgeniy Landis](https://es.wikipedia.org/wiki/Yevgeniy_Landis). Lo dieron a conocer en la publicación de un artículo en [1962](https://es.wikipedia.org/wiki/1962), «An algorithm for the organization of information» («Un algoritmo para la organización de la información»).

Los árboles AVL están siempre equilibrados de tal modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa. Gracias a esta forma de equilibrio (o balanceo), la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de [complejidad](https://es.wikipedia.org/wiki/Complejidad_computacional) [O](https://es.wikipedia.org/wiki/Cota_superior_asint%C3%B3tica)(log n). El factor de equilibrio puede ser almacenado directamente en cada nodo o ser computado a partir de las alturas de los subárboles.

Para conseguir esta propiedad de equilibrio, la inserción y el borrado de los nodos se ha de realizar de una forma especial. Si al realizar una operación de inserción o borrado se rompe la condición de equilibrio, hay que realizar una serie de [rotaciones de los nodos](https://es.wikipedia.org/w/index.php?title=Rotaci%C3%B3n_de_%C3%A1rbol&action=edit&redlink=1).

Los árboles AVL más profundos son los [árboles de Fibonacci](https://es.wikipedia.org/wiki/%C3%81rbol_de_Fibonacci).{\displaystyle |H(T\_{i})-H(T\_{d})|<=1}

## Factor de equilibrio

Cada nodo, además de la información que se pretende almacenar, debe tener los dos punteros a los árboles derecho e izquierdo, igual que los [árboles binarios de búsqueda](https://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda) (ABB), y además el dato que controla el factor de equilibrio.

El factor de equilibrio es la diferencia entre las alturas del árbol derecho y el izquierdo:

FE = altura subárbol derecho - altura subárbol izquierdo

Por definición, para un árbol AVL, este valor debe ser -1, 0 o 1.

Si el factor de equilibrio de un nodo es:

0 -> el nodo está equilibrado y sus subárboles tienen exactamente la misma altura.

1 -> el nodo está equilibrado y su subárbol derecho es un nivel más alto.

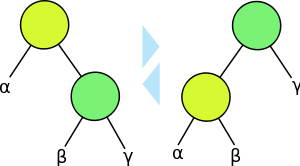
-1 -> el nodo está equilibrado y su subárbol izquierdo es un nivel más alto.

Si el factor de equilibrio {\displaystyle |Fe|>=2} es necesario reequilibrar.

## Operaciones

Las operaciones básicas de un árbol AVL implican generalmente el realizar los mismos algoritmos que serían realizados en un [árbol binario de búsqueda](https://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda) desequilibrado, pero precedido o seguido por una o más de las llamadas "rotaciones AVL".

## Rotaciones

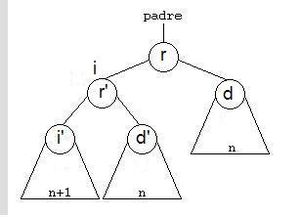
[](https://commons.wikimedia.org/wiki/File:BinaryTreeRotations.svg)

Las rotaciones internas en árboles binarios son operaciones internas comunes utilizadas para mantener el balance perfecto (o casi perfecto) del árbol binario. Un árbol balanceado permite operaciones en tiempo logarítmico

El reequilibrado se produce de abajo hacia arriba sobre los nodos en los que se produce el desequilibrio. Pueden darse dos casos: rotación simple o rotación doble; a su vez ambos casos pueden ser hacia la derecha o hacia la izquierda.

## Rotación simple a la derecha

De un árbol de raíz (r) y de hijos izquierdo (i) y derecho (d), lo que haremos será formar un nuevo árbol cuya raíz sea la raíz del hijo izquierdo, como hijo izquierdo colocamos el hijo izquierdo de i (nuestro i’) y como hijo derecho construimos un nuevo árbol que tendrá como raíz, la raíz del árbol (r), el hijo derecho de i (d’) será el hijo izquierdo y el hijo derecho será el hijo derecho del árbol (d).

[](https://commons.wikimedia.org/wiki/File:Rotacionsimplederecha1.JPG)

op rotDer: AVL{X} -> [AVL{X}] .

eq rotDer(arbolBin(R1, arbolBin(R2, I2, D2), D1)) ==

arbolBin(R2, I2, arbolBin(R1, D2, D)) .

## Rotación simple a la izquierda

De un árbol de raíz (r) y de hijos izquierdo (i) y derecho (d), consiste en formar un nuevo árbol cuya raíz sea la raíz del hijo derecho, como hijo derecho colocamos el hijo derecho de d (nuestro d’) y como hijo izquierdo construimos un nuevo árbol que tendrá como raíz la raíz del árbol (r), el hijo izquierdo de d será el hijo derecho (i’) de r y el hijo izquierdo será el hijo izquierdo del árbol (i).

Precondición : Tiene que tener hijo derecho no vacío.

op rotIzq: AVL{X} -> [AVL{X}] .

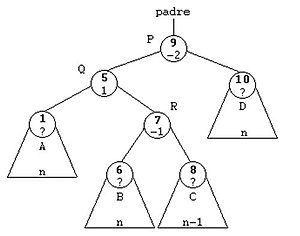
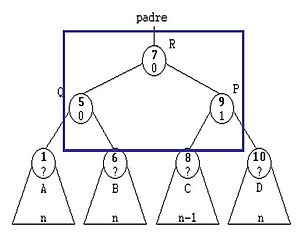
eq rotIzq(arbolBin(R1, I, arbolBin(R2, I2, D2))) ==

arbolBin(R2, arbolBin(R1, I, I2), D2) .

Si la inserción se produce en el hijo derecho del hijo izquierdo del nodo desequilibrado (o viceversa) hay que realizar una doble rotación.

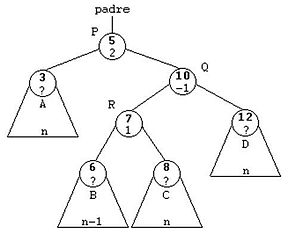
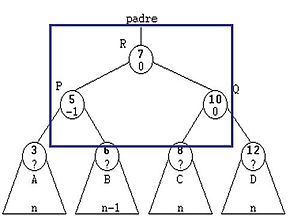
## Rotación doble a la derecha

La Rotación doble a la Derecha son dos rotaciones simples, primero rotación simple izquierda y luego rotación simple derecha.

[](https://commons.wikimedia.org/wiki/File:ROTACIONDCHA1.jpg)[](https://commons.wikimedia.org/wiki/File:ROTACIONDCHA2.jpg)

## Rotación doble a la izquierda

La Rotación doble a la Izquierda son dos rotaciones simples, primero rotación simple derecha y luego rotación simple izquierda.

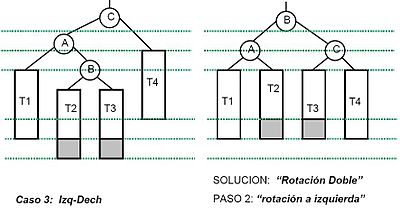
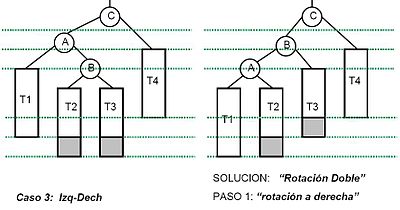
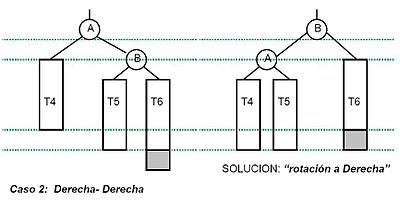
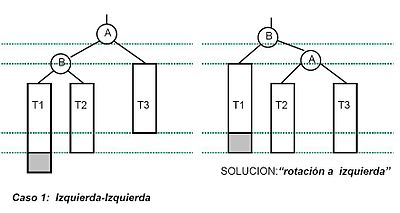
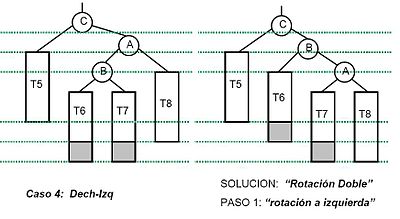
[](https://commons.wikimedia.org/wiki/File:ROTACIONIZDA2.jpg)[](https://commons.wikimedia.org/wiki/File:ROTACIONIZQ2.jpg)

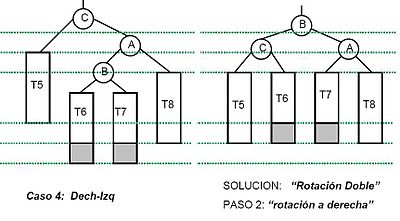
## Inserción

La inserción en un árbol de AVL puede ser realizada insertando el valor dado en el árbol como si fuera un árbol de búsqueda binario desequilibrado y después retrocediendo hacia la raíz, rotando sobre cualquier nodo que pueda haberse desequilibrado durante la inserción.

Proceso de inserción:

* Buscar hasta encontrar la posición de inserción o modificación (proceso idéntico a inserción en árbol binario de búsqueda)
* Insertar el nuevo nodo con factor de equilibrio “equilibrado”
* Desandar el camino de búsqueda, verificando el equilibrio de los nodos, y re-equilibrando si es necesario

[](https://commons.wikimedia.org/wiki/File:Insercion3-2.jpg)[](https://commons.wikimedia.org/wiki/File:Insercion3-1.jpg)[](https://commons.wikimedia.org/wiki/File:Insercion2.jpg)[](https://commons.wikimedia.org/wiki/File:Insercion1.jpg)   [](https://commons.wikimedia.org/wiki/File:Insercion4-1.jpg)

[](https://commons.wikimedia.org/wiki/File:Insercion4-2.jpg)

Debido a que las rotaciones son una

operación que tienen complejidad constante y a que la altura está limitada a O (log(n)), el tiempo de ejecución para la inserción es del orden O (log(n)).

op insertar: X$Elt AVL{X} -> AVLNV{X} .

eq insertar(R, crear) == arbolBin(R, crear, crear) .

ceq insertar(R1, arbolBin(R2, I, D)) ==

**if** (R1==R2) then

arbolBin(R2, I, D)

elseif (R1<R2) then

if ( (altura(insertar(R1,I)) – altura(D)) < 2) then

arbolBin(R2, insertar(R1, I), D)

**else** \*\*\*hay que reequilibrar

if (R1 < raíz(I)) then

rotarDer(arbolBin(R2, insertar(R1, I), D))

**else**

rotarDer(arbolBin(R2, rotarIzq(insertar(R1, I)), D))

fi .

fi .

**else**

**if** ( (altura(insertar(R1,I)) – altura(D)) < 2) then

arbolBin(R2, insertar(R1, I), D)

**else** \*\*\* hay que reequilibrar

if (R1 > raíz(D)) then

rotarIzq(arbolBin(R, I, insertar(R1, D)))

**else**

rotatIzq(arbolBin(R, I, rotarDer(insertar(R1, D))))

fi .

fi .

fi .

## Extracción

El procedimiento de borrado es el mismo que en el caso de árbol binario de búsqueda. La diferencia se encuentra en el proceso de reequilibrado posterior. El problema de la extracción puede resolverse en O (log(n)) pasos. Una extracción trae consigo una disminución de la altura de la rama donde se extrajo y tendrá como efecto un cambio en el factor de equilibrio del nodo padre de la rama en cuestión, pudiendo necesitarse una rotación.

Esta disminución de la altura y la corrección de los factores de equilibrio con sus posibles rotaciones asociadas pueden propagarse hasta la raíz.

# Ejemplo de ARBOL AVL programado por OGMR

# #include <stdio.h>

# #include <stdlib.h>

# #include <graphics.h>

# #include <time.h>

# typedef struct avlnode

# {

# int clave;

# int bal; /\* Factor de balance -1,0,1 \*/

# struct avlnode \*left, \*right;

# } nodo, \*pnodo;

# # define max(A,B) ((A)>(B)?(A):(B)) /\* Definición de macros \*/

# # define min(A,B) ((A)>(B)?(B):(A))

# int flag; /\* Marca para registrar cambios de altura. En rebalance ascendente \*/

# //flag = 1 indica que debe seguir el ascenso rebalanceando.

# int key; /\* Variable global, para disminuir argumentos \*/

# int alto\_avl = 0; /\* Altura árbol avl. Número de nodos desde la raíz a las hojas.\*/

# static pnodo lrot(pnodo t);

# static pnodo rrot(pnodo t);

# static void Error(int tipo);

# int Altura(void);

# pnodo CreaNodo(int key);

# pnodo insertR(pnodo t);

# pnodo InsertarAVL(int clave, pnodo t);

# pnodo deleteR(pnodo t);

# pnodo DescartarAVL(int clave, pnodo t);

# pnodo deltreeR(pnodo t);

# pnodo deltree(pnodo t);

# void inorder(pnodo t, int profundidad, int levelizquierda, int levelderecha, pnodo Raiz);

# int main()

# {

# int num=0, posicion=0, dat=0, sel=0,opc;

# int contenido[31]={955,477,1432,238,715,1194,1670,119,357,596,834,1074,1313,1551,1789,59,178,297,416,536,655,774,893,1014,1133,1253,1372,1491,1610,1729,1848};

# //int contenido[30]={98,33,67,55,14,27,72,46,9,47,11,39,20,4,6,58,91,17,51,89,32,74,93,65,27,32,16,95,37,42};

# printf("\t\t\t\tMENU INICIAL\n");

# system("cls");

# printf("Elija la opcion mas conveniente en su caso:\n\n");

# printf("\t\t\t1.Generacion aleatoria de nodos del 1 al 50\n");

# printf("\t\t\t2.Quiero introducir uno a uno mis datos, para vizualizar como se van creando\n");

# printf("\t\t\t3.Quiero solo probar el programa con un conjunto a eleccion del programador\n");

# scanf("%d",&opc);

# system("cls");

# 

# switch (opc){

# case 1:{

# dat=0;

# printf("Seleccione el numero de nodos que desea introducir al azar:");

# scanf("%d",&sel);

# initwindow(1910,1000);

# pnodo t=NULL;

# srand (time(NULL));

# while (dat<sel){

# posicion=rand()%50;

# t=InsertarAVL(posicion, t);

# inorder(t, 0,0,0,t);

# printf("\*\*\*\*\*\*\n");

# delay(2000);

# cleardevice();

# dat++;

# }

# delay(9999);

# closegraph();

# break;

# }

# case 2:{

# printf("Ahh elegido un camino que visualmente tendra su recompenza\n");

# printf("Seleccione el numero de nodos que desea introducir personalmente:");

# scanf("%d",&sel);

# system("cls");

# initwindow(1910,1000);

# pnodo t=NULL;

# while (dat<sel){

# printf("Ingrese su dato:\n");

# scanf("%d",&posicion);

# t=InsertarAVL(posicion, t);

# inorder(t, 0,0,0,t);

# printf("\*\*\*\*\*\*\n");

# delay(3000);

# cleardevice();

# dat++;

# }

# delay(9999);

# closegraph();

# break;

# }

# case 3:{

# int i=0;

# printf("Okey, gracias por confiar en mi, le mostrare un Arbol equilibrado...");

# printf("La raiz de la cual va a partir este arbol es 55:");

# sel=31;

# system("cls");

# pnodo t=NULL;

# initwindow(1910,1000);

# for (int i=0;i<31;i++){

# t=InsertarAVL(contenido[i], t);

# inorder(t, 0,0,0,t);

# printf("\*\*\*\*\*\*\n");

# delay(666);

# cleardevice();

# }

# delay (1000);

# closegraph();

# break;

# }

# default:{

# printf("No eligio algo correcto, lo siento, me desarmo");

# break;

# }

# }

# }

# /\* Rotación Izquierda \*

# \* A B

# \* / \ / \

# \* a B ==> A c

# \* / \ / \

# \* b c a b

# \* Sólo cambian los factores de balance de los nodos A y B

# \* Los factores de balance de los sub-árboles no cambian. \*/

# static pnodo lrot(pnodo t)

# {

# pnodo temp;

# int x,y;

# temp = t;

# t = t->right;

# temp->right = t->left;

# t->left = temp;

# //Recalcula factores de balance de los dos nodos

# x = temp->bal; // oldbal(A)

# y = t->bal; // oldbal(B)

# temp->bal = x-1-max(y, 0); // nA

# t->bal = min(x-2+min(y, 0), y-1); // nB

# return t;

# }

# /\* Rotación derecha

# \*

# \* A B

# \* / \ / \

# \* B c ==> a A

# \* / \ / \

# \* a b b c

# \*

# \*/

# static pnodo rrot(pnodo t)

# {

# pnodo temp = t;

# int x,y;

# t = t->left;

# temp->left = t->right;

# t->right = temp;

# x = temp->bal; // oldbal(A)

# y = t->bal; // oldbal(B)

# temp->bal = x+1-min(y, 0); // nA

# t->bal = max(x+2+max(y, 0), y+1); //nB

# return t;

# }

# static void Error(int tipo)

# {

# if (tipo) printf("\nError en inserción\n");

# else printf("\nError en descarte\n");

# }

# int Altura(void)

# {

# return alto\_avl;

# }

# pnodo CreaNodo(int key)

# {

# pnodo t;

# t = (pnodo) malloc(sizeof(nodo));

# t->clave=key;

# t->left=0;

# t->right=0;

# return t;

# }

# pnodo insertR(pnodo t)

# {

# if (t == NULL) /\* Llegó a un punto de inserción \*/

# {

# t = CreaNodo(key);

# t->bal = 0; /\* Los dos hijos son nulos \*/

# flag = 1; /\* Marca necesidad de revisar balances \*/

# return t; /\* retorna puntero al insertado \*/

# }

# else if (t->clave < key)

# {

# //desciende por la derecha

# t->right = insertR(t->right);

# //se pasa por la siguiente línea en la revisión ascendente

# t->bal += flag; /\* Incrementa factor de balance \*/

# }

# else if (t->clave > key)

# {

# //desciende por la izquierda

# t->left = insertR(t->left);

# //se corrige en el ascenso

# t->bal -= flag; /\* Decrementa balance \*/

# }

# else /\* (t->k == key) Ya estaba en el árbol \*/

# {

# Error(1);

# flag = 0;

# }

# if (flag == 0) /\* No hay que rebalancear. Sigue el ascenso \*/

# return t;

# /\*El código a continuación es el costo adicional para mantener propiedad AVL \*/

# /\* Mantiene árbol balanceado avl. Sólo una o dos rotaciones por inserción \*/

# if (t->bal < -1)

# {

# /\* Quedó desbalanceado por la izquierda. Espejos Casos c y d.\*/

# if (t->left->bal > 0)

# /\* Si hijo izquierdo está cargado a la derecha \*/

# t->left = lrot(t->left);

# t = rrot(t);

# flag = 0; /\* El subárbol no aumenta su altura \*/

# }

# else if (t->bal > 1)

# {

# /\* Si quedó desbalanceado por la derecha: Casos c y d.\*/

# if (t->right->bal < 0)

# /\* Si hijo derecho está cargado a la izquierda Caso d.\*/

# t->right = rrot(t->right);

# t = lrot(t); /\* Caso c.\*/

# flag = 0; /\* El subárbol no aumenta su altura \*/

# }

# else if (t->bal == 0)/\* La inserción lo balanceo \*/

# flag = 0; /\* El subárbol no aumenta su altura. Caso a\*/

# else /\* Quedó desbalanceado con -1 ó +1 Caso b \*/

# flag = 1; /\* Propaga ascendentemente la necesidad de rebalancear \*/

# return t;

# }

# /\* Mantiene variable global con el alto del árbol. \*/

# pnodo InsertarAVL(int clave, pnodo t)

# {

# key = clave; //pasa argumento a global.

# t = insertR(t);

# if (flag == 1)

# alto\_avl++;

# //si la propagación llega hasta la raíz, aumenta la altura.

# return t;

# }

# pnodo deleteR(pnodo t)

# {

# pnodo p;

# if (t == NULL) /\* No encontró nodo a descartar \*/

# {

# Error(0);

# flag = 0;

# }

# else if (t->clave < key)

# {

# //Comienza el descenso por la derecha

# t->right = deleteR(t->right);

# //aquí se llega en el retorno ascendente.

# t->bal -= flag; /\* Se descartó por la derecha. Disminuye factor \*/

# //Se retorna después de la revisión de los factores

# }

# else if (t->clave > key)

# {

# //Desciende por la izquierda

# t->left = deleteR(t->left);

# //o se llega por esta vía si se descartó por la izquierda.

# t->bal += flag; /\* se descartó por la izq. Aumenta factor de balance \*/

# }

# else /\* (t->clave == key) \*/

# {

# /\* Encontró el nodo a descartar \*/

# if (t->left == NULL) /\*Si hay hijo derecho debe ser hoja, por ser AVL \*/

# {

# p = t;

# t = t->right;

# free(p);

# flag = 1; /\* Debe seguir revisando factores de balance \*/

# return t; /\* ascendentemente \*/

# }

# else if (t->right == NULL) /\*Si hay hijo izquierdo debe ser hoja \*/

# {

# p = t;

# t = t->left;

# free(p);

# flag = 1; /\* Asciende revisando factores de balance \*/

# return t; /\* Corrigiendo \*/

# }

# else /\* Tiene dos hijos \*/

# {

# if (t->bal<0)

# {

# /\* Si cargado a la izquierda, elimina mayor descendiente hijo izq \*/

# p = t->left;

# while (p->right != NULL) p = p->right;

# t->clave = p->clave;

# key = p->clave; //busca hoja a eliminar

# t->left = deleteR(t->left);

# t->bal += flag; /\* incrementa factor de balance \*/

# }

# else /\* Si cargado a la derecha, elimina menor descendiente hijo der \*/

# {

# p = t->right;

# while (p->left != NULL)

# p = p->left;

# t->clave = p->clave;

# key = p->clave;

# t->right = deleteR(t->right);

# t->bal -= flag; /\* decrementa balance \*/

# }

# }

# }

# /\* Mantiene árbol balanceado avl. Sólo una o dos rotaciones por descarte \*/

# if (flag == 0 ) /\* No hay que rebalancear. Sigue el ascenso, sin rebalancear \*/

# return t;

# /\* Hay que revisar factores de balance en el ascenso\*/

# if (t->bal < -1) /\* Si quedó desbalanceado por la izquierda y dejó de ser AVL \*/

# {

# if (t->left->bal > 0) /\*espejos casos c, d y e \*/

# {

# /\* Si el hijo izquierdo está cargado a la derecha \*/

# t->left = lrot(t->left);

# flag = 1; /\*Continuar revisando factores \*/

# }

# else if (t->left->bal == 0)

# flag = 0; /\*No debe seguir el rebalance \*/

# else

# flag = 1;/\* Debe seguir revisando factores de balance \*/

# t = rrot(t);

# }

# else if (t->bal > 1) /\* Si quedó desbalaceado por la derecha \*/

# {

# if (t->right->bal < 0)

# {

# /\* Si hijo derecho está cargado a la izquierda \*/

# t->right = rrot(t->right);

# flag = 1; //debe seguir revisando. Caso d.

# }

# else if (t->right->bal == 0)

# flag = 0; /\* No debe seguir el rebalance. Caso c. \*/

# else //positivo

# flag = 1;/\* Debe seguir revisando factores de balance. Caso e. \*/

# t = lrot(t);

# }

# else if (t->bal == 0) /\* Si estaba en +1 ó -1 y queda en cero \*/

# flag = 1; /\* Debe seguir corrigiendo. Caso b.\*/

# else /\* Si estaba en cero y queda en -1 ó +1 \*/

# flag = 0; /\* No debe seguir rebalanceando. Caso a.\*/

# return t;

# }

# pnodo DescartarAVL(int clave, pnodo t)

# {

# key = clave;

# t = deleteR(t);

# if (flag == 1) alto\_avl--;

# return t;

# }

# pnodo deltreeR(pnodo t)

# {

# if (t != NULL)

# {

# t->left = deltreeR(t->left);

# t->right = deltreeR(t->right);

# free(t); //borra la raíz subárbol

# }

# return NULL;

# }

# pnodo deltree(pnodo t)

# {

# alto\_avl = 0;

# return deltreeR(t);

# }

# void inorder(pnodo t, int profundidad, int levelizquierda, int levelderecha, pnodo Raiz)

# {

# char ptb[1];

# if (t != NULL)

# {

# 

# printf ("v= %d p=%d bal=%d \n", t->clave, profundidad, t->bal);

# if(profundidad==0){

# setlinestyle(3, 3, 3);

# setcolor(2);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 5);

# outtextxy(750,10,"Arboles AVL");

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# circle(955,150,50);

# floodfill(955,150,4);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# setcolor(GREEN);

# sprintf(ptb,"%d",t->clave);

# outtextxy(935,130,ptb);

# setcolor(RED);

# outtextxy(800,100,"Raiz");

# line(870,110,905,140);

# line(905,140,895,145);

# line(905,140,900,125);

# }

# if(levelderecha==1&&levelizquierda==0){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1005,150,1432,250);

# circle(1432,300,50);

# floodfill(1432,300,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1412,280,ptb);

# 

# 

# }

# else if(levelderecha==2&&levelizquierda==0){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1482,300,1670,400);

# circle(1670,450,50);

# floodfill(1670,450,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1650,420,ptb);

# 

# }

# else if(levelderecha==3&&levelizquierda==0){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1789,550,1720,450);

# circle(1789,600,50);

# floodfill(1789,600,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1769,580,ptb);

# 

# }

# else if(levelderecha==4&&levelizquierda==0){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1848,700,1839,600);

# circle(1848,750,50);

# floodfill(1848,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1828,720,ptb);

# 

# }

# else if(levelizquierda==1&&levelderecha==3&&t->clave<Raiz->right->right->clave&&t->clave>Raiz->right->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1610,700,1601,600);

# circle(1610,750,50);

# floodfill(1610,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1590,720,ptb);

# 

# }

# else if(levelizquierda==1&&levelderecha==2&&t->clave<Raiz->right->clave&& t->clave>Raiz->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1313,550,1244,450);

# circle(1313,600,50);

# floodfill(1313,600,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1293,580,ptb);

# 

# }

# 

# else if(levelizquierda==1&&levelderecha==3&&t->clave<Raiz->right->clave&& t->clave>Raiz->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1372,700,1363,600);

# circle(1372,750,50);

# floodfill(1372,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1352,730,ptb);

# 

# }

# 

# else if(levelizquierda==2&&levelderecha==2&&t->clave<Raiz->right->left->clave&&t->clave>Raiz->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1133,700,1124,600);

# circle(1133,750,50);

# floodfill(1133,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1113,730,ptb);

# 

# }

# 

# else if(levelizquierda==1&&levelderecha==1&&t->clave==Raiz->left->right->clave&&t->clave<Raiz->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(715,400,527,300);

# circle(715,450,50);

# floodfill(715,450,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(695,420,ptb);

# 

# }

# else if(levelizquierda==1&&levelderecha==2&&t->clave==Raiz->left->right->right->clave&&t->clave<Raiz->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(834,550,765,450);

# circle(834,600,50);

# floodfill(834,600,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(814,580,ptb);

# 

# }

# else if(levelizquierda==2&&levelderecha==1&&t->clave==Raiz->left->left->right->clave&&t->clave<Raiz->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(357,550,288,450);

# circle(357,600,50);

# floodfill(357,600,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(327,580,ptb);

# 

# }

# else if(levelizquierda==2&&levelderecha==1&&t->clave==Raiz->left->left->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(357,550,288,450);

# circle(357,600,50);

# floodfill(357,600,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(327,580,ptb);

# 

# }

# else if(levelizquierda==3&&levelderecha==1&&t->clave<Raiz->left->left->clave&&t->clave<Raiz->left->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(178,700,169,600);

# circle(178,750,50);

# floodfill(178,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(158,720,ptb);

# 

# }

# else if(levelizquierda==2&&levelderecha==2&&t->clave==Raiz->left->left->right->right->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(416,700,407,600);

# circle(416,750,50);

# floodfill(416,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(396,720,ptb);

# 

# }

# else if(levelizquierda==2&&levelderecha==2&&t->clave==Raiz->left->right->left->right->clave){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(655,700,646,600);

# circle(655,750,50);

# floodfill(655,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(625,720,ptb);

# 

# }

# else if(levelizquierda==1&&levelderecha==3&&t->clave==Raiz->left->right->right->right->clave&&t->clave<Raiz->clave ){

# 

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(893,700,884,600);

# circle(893,750,50);

# floodfill(893,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(873,720,ptb);

# 

# }

# /////////////////////////////////////////////////////////////////////////////

# else if(levelizquierda==1&&levelderecha==0&&t->clave<Raiz->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(905,150,477,250);

# circle(477,300,50);

# floodfill(477,300,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(457,300,ptb);

# 

# }

# else if (levelizquierda==2&&levelderecha==0&&t->clave<Raiz->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(427,300,238,400);

# circle(238,450,50);

# floodfill(238,450,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(218,450,ptb);

# 

# }

# else if (levelizquierda==3&&levelderecha==0&&t->clave<Raiz->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(188,450,119,550);

# circle(119,600,50);

# floodfill(119,600,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(99,580,ptb);

# 

# }

# else if (levelizquierda==4&&levelderecha==0&&t->clave<Raiz->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(69,600,59,700);

# circle(59,750,50);

# floodfill(59,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(39,720,ptb);

# 

# }

# 

# else if (levelizquierda==2&&levelderecha==1&&t->clave==Raiz->left->right->left->clave&&t->clave<Raiz->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(665,450,596,550);

# circle(596,600,50);

# floodfill(596,600,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(576,600,ptb);

# 

# }

# 

# else if (levelizquierda==3&&levelderecha==1&&t->clave==Raiz->left->left->right->left->clave&&t->clave<Raiz->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(307,600,297,700);

# circle(297,750,50);

# floodfill(297,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(277,730,ptb);

# 

# }

# else if (levelizquierda==3&&levelderecha==1&&t->clave==Raiz->left->right->left->left->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(546,600,536,700);

# circle(536,750,50);

# floodfill(536,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(516,730,ptb);

# 

# }

# else if (levelizquierda==2&&levelderecha==2&&t->clave==Raiz->left->right->right->left->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(784,600,774,700);

# circle(774,750,50);

# floodfill(774,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(754,730,ptb);

# 

# }

# else if (levelizquierda==1&&levelderecha==1&&t->clave==Raiz->right->left->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1382,300,1194,400);

# circle(1194,450,50);

# floodfill(1194,450,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1174,420,ptb);

# 

# }

# 

# else if (levelizquierda==2&&levelderecha==1&&t->clave==Raiz->right->left->left->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1144,450,1074,550);

# circle(1074,600,50);

# floodfill(1074,600,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1054,580,ptb);

# 

# }

# else if (levelizquierda==3&&levelderecha==1&&t->clave==Raiz->right->left->left->left->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1024,600,1014,700);

# circle(1014,750,50);

# floodfill(1014,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(994,730,ptb);

# 

# }

# else if (levelizquierda==2&&levelderecha==2&&t->clave==Raiz->right->left->right->left->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1263,600,1253,700);

# circle(1253,750,50);

# floodfill(1253,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1223,730,ptb);

# 

# }

# 

# else if (levelizquierda==1&&levelderecha==2&&t->clave==Raiz->right->right->left->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1620,450,1551,550);

# circle(1551,600,50);

# floodfill(1551,600,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1531,580,ptb);

# 

# }

# else if (levelizquierda==2&&levelderecha==2&&t->clave==Raiz->right->right->left->left->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1501,600,1491,700);

# circle(1491,750,50);

# floodfill(1491,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1471,730,ptb);

# 

# }

# else if (levelizquierda==1&&levelderecha==3&&t->clave==Raiz->right->right->right->left->clave){

# setfillstyle(SOLID\_FILL,4);

# setcolor(4);

# line(1739,600,1729,700);

# circle(1729,750,50);

# floodfill(1729,750,4);

# setcolor(GREEN);

# settextstyle(BOLD\_FONT, HORIZ\_DIR, 3);

# sprintf(ptb,"%d",t->clave);

# outtextxy(1709,730,ptb);

# 

# }

# inorder(t->left, profundidad+1, levelizquierda+1, levelderecha,Raiz);

# inorder(t->right, profundidad+1,levelizquierda, levelderecha+1,Raiz);

# }

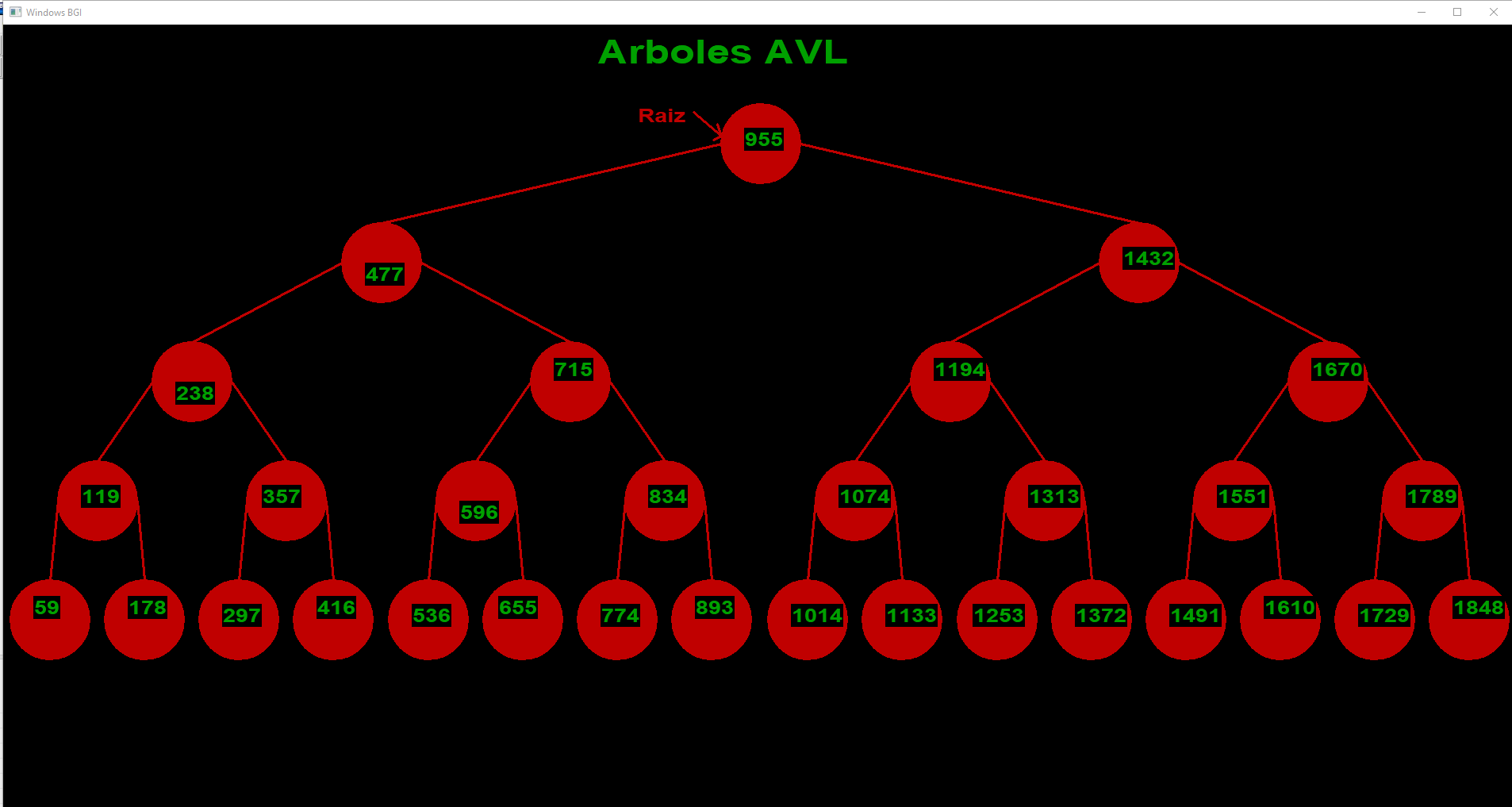
# }

## Imágenes de Funcionamiento

### Menú Inicial

## 

## Programa por defecto



# Conclusion.

Es de esta forma que podemos concluir que todos los que nos podemos encontrar en los diversos programas que realizamos, tienen diversas formas de resolverse sin perder de vista que cada una de estas formas tienen diversas desventajas y ventajas en su haber, parte de esto es el gran trabajo del programador descubrir cuáles son y de qué manera utilizarlas en su beneficio o como herramienta, para hacer de su trabajo más fácil sencillo y eficiente, pues el código por sí mismo es solo código, es la lógica del programador la que logra que todo forme parte de un ciclo en el cual se realizan diversas partes y tareas específicas.

# Referencias Bibliográficas:

**[1]:**

Geeky Theory, & Geeky Theory. (2020). ¿Qué es la recursividad? Retrieved November 24, 2020, from Geeky Theory website: https://geekytheory.com/que-es-la-recursividad

‌**[2]:**

Colaboradores de los proyectos Wikimedia, “Árbol binario,” *Wikipedia.org*, 21-May-2004. [Online]. Avalarle: https://es.wikipedia.org/wiki/%C3%81rbol\_binario. [Access Ed: 17-Dec-2020]

‌**[3]**

“ARBOLES BINARIOS DE BUSQUEDA,” *Decsai.ugr.es*, 2020. [Online]. Available: http://decsai.ugr.es/~jfv/ed1/tedi/cdrom/docs/arb\_BB.htm. [Accessed: 17-Dec-2020]

**[4]**

Colaboradores de los proyectos Wikimedia, “Árbol AVL,” *Wikipedia.org*, 04-Jul-2004. [Online]. Available: https://es.wikipedia.org/wiki/%C3%81rbol\_AVL#:~:text=Un%20%C3%A1rbol%20AVL%20es%20un,auto%2Dbalanceable%20que%20se%20ide%C3%B3. [Accessed: 27-Dec-2020]

‌

‌