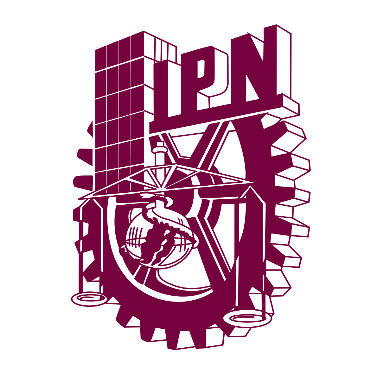
****

**Instituto Politécnico Nacional**

**Escuela Superior de Computo**

*Alumno:*

* Monroy Ramírez Oscar G.

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

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

***Evidencia:*** *Reporte 3er Parcial*

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

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

Contenido

[Introducción 3](#_Toc61456376)

[Marco Teórico 4](#_Toc61456377)

[Arboles 4](#_Toc61456378)

[Conversión de árboles generales en árboles binarios 4](#_Toc61456379)

[Arboles Binarios 7](#_Toc61456380)

[Tipos de árboles binarios 8](#_Toc61456381)

[Implementación en C 8](#_Toc61456382)

[Recorridos sobre árboles binarios 9](#_Toc61456383)

[Recorridos en profundidad 9](#_Toc61456384)

[Recorridos en amplitud (o por niveles) 10](#_Toc61456385)

[Arboles Binarios de búsqueda (Arboles ABB) 11](#_Toc61456386)

[Análisis en la eficiencia de las operaciones 14](#_Toc61456387)

[Árbol Binario (AVL) 17](#_Toc61456388)

[Factor de equilibrio 17](#_Toc61456389)

[Operaciones 17](#_Toc61456390)

[Grafos 23](#_Toc61456391)

[Representaciones matriciales y basadas en listas 23](#_Toc61456392)

[Estructura de lista 23](#_Toc61456393)

[Estructuras matriciales 23](#_Toc61456394)

[Búsquedas de amplitud 24](#_Toc61456395)

[Procedimiento 24](#_Toc61456396)

[Código 25](#_Toc61456397)

[Búsqueda en profundidad 25](#_Toc61456398)

[Evaluación 26](#_Toc61456399)

[Pseudocódigo 27](#_Toc61456400)

[Problema del camino más corto 28](#_Toc61456401)

[Definición 29](#_Toc61456402)

[Conclusión. 30](#_Toc61456403)

[Referencias Bibliográficas: 30](#_Toc61456404)

# 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.

En este reporte analizaremos una estructura de datos que es útil en muchas aplicaciones: el arból. Se definen varias dormas de esta estructura de datos y que se muestra como pueden representarse en C y cómo se pueden aplicar para resolver una amplia gama de problemas. Al igual que son las listas, los arboles se tratan principalemente comoe estructura de datos y no como un tipo de datos. Es decir, el interés inicial es por la impantacion mas que por la definicion matemática.

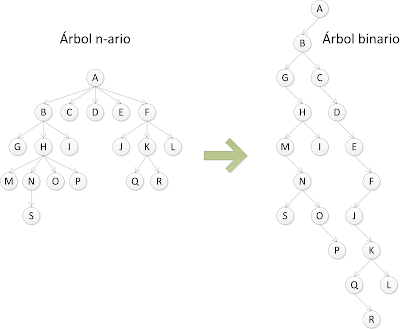
Tambien consideramos una nueva estructura de datos: el grafo. Definimos algunos de los terminos asosiados con los grafos y mostramos cómo implementarlos en C. También presentamos varias aplicaciones de grafos.

# Marco Teórico

# Arboles

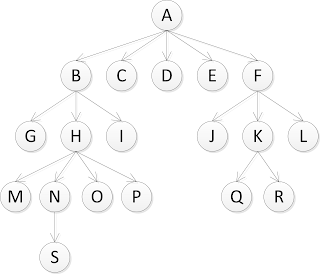
## Conversión de árboles generales en árboles binarios

Un árbol es una estructura de datos muy importante en el mundo de la programación, sin embargo el tipo de árbol que se usa comúnmente por sus propiedades es el árbol binario (árbol donde cada nodo tiene a lo máximo 2 nodos descendientes o nodos  hijo). Es muy natural definir arboles n-arios (arboles donde los nodos pueden tener cualquier cantidad de hijos) y para usar las propiedades de los arboles binarios hacer la respectiva conversión.**[1]**

[](http://4.bp.blogspot.com/-P9uF6y6JlU0/T7kwtK_GmBI/AAAAAAAAAnA/MMgNo0v5s44/s1600/Arboles.png)

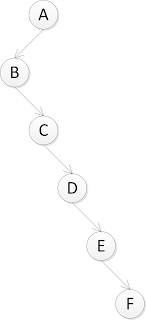
**Figura No.1 “Transformación”**

Si tenemos como inicio al siguiente árbol n-ario, el algoritmo de conversión es muy sencillo y se explicara gráficamente a continuación:

[](http://1.bp.blogspot.com/-PNc-AHL7xj0/T7k1D8QJMmI/AAAAAAAAAnM/1Q7cf72MNcY/s1600/Arboles.png)

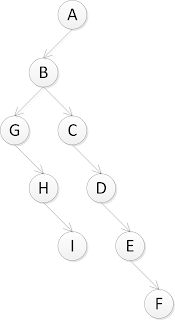
**Figura No.2 “Árbol N-ario”**

Empezamos por la raíz, la raíz sigue siendo la misma para el árbol binario, a continuación el hijo mas de la izquierda de la raíz para a ser hijo inmediato izquierdo de la raíz y los demás hijos de la raíz se van colocando por orden. **[1]**

[](http://3.bp.blogspot.com/-6cs9MyA4Hw4/T7k1lTCAT9I/AAAAAAAAAnU/3Uuy0yoAsl8/s1600/Arboles.png)El segundo hijo de la raíz pasa a ser hijo derecho del primer hijo, y asi sucesivamente con los demás hijos que pasan a ser hijos derechos siempre se su hermano anterior. **[1]**

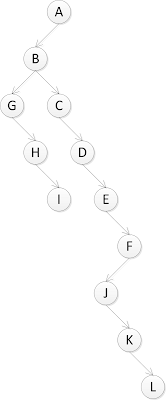
**Figura No.3 ”Realineación”**

Luego pasamos a hacer lo mismo con los hijos de la raíz. Empezamos con el nodo B (en este ejemplo) y colocamos a su primer hijo de mas a la izquierda cono su hijo inmediato a la izquierda y los demás a la derecha de manera consecutiva. **[1]**

[](http://4.bp.blogspot.com/-JFF5siHdiFI/T7k4R_wVYGI/AAAAAAAAAng/qhKwu7IHOFE/s1600/Arboles.png)

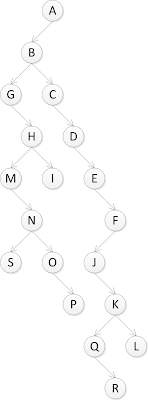
**Figura No.3” Realineación Secundaria”**

Pasamos al siguiente nodo (Nodo F en este ejemplo) al otro hijo que tienes hijos, y hacemos lo mismo. **[1]**

[](http://2.bp.blogspot.com/-POh3Nh3Ru3g/T7k5RUAoZ3I/AAAAAAAAAn4/_539L3Oth3I/s1600/Arboles.png)

**Figura No.3” Transformación Parcial”**

Terminado esto pasamos a los siguientes nodos hijo, culminando así y llegando al siguiente árbol. **[1]**

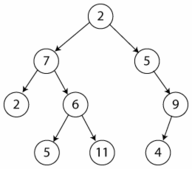
[](http://1.bp.blogspot.com/-KU4pibpPerE/T7k7OCC_CfI/AAAAAAAAAoA/M8_nKGMr8ic/s1600/Arboles.png)

**Figura No.5” transformación Completa”**

Como podemos observar el algoritmo es sencillo, lo que falta por hacer es implementarlo en un lenguaje de programación si lo piden estará en poco tiempo publicado por este medio. **[1]**

## Arboles Binarios

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);

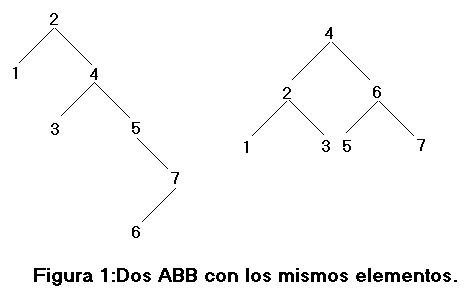
} *// al vaciarse la cola se han visitado todos los nodos del árbol*

}

## 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*. **[3]**

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: **[3]**

#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));

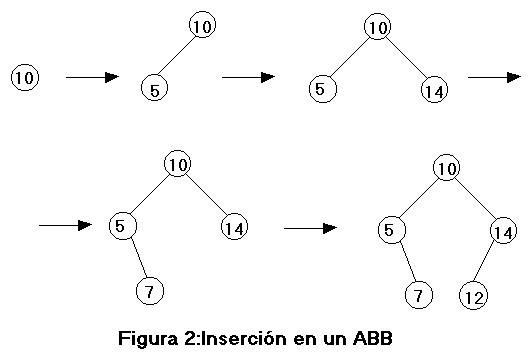
else

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. **[3]**

El resultado es el que muestra la figura



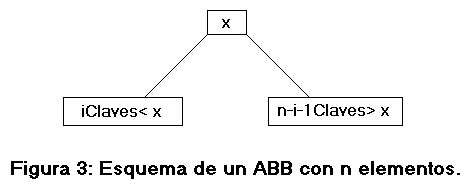
## Análisis en la eficiencia de las operaciones

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.

## Árbol Binario (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.

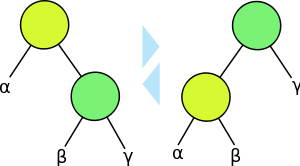
-1 -> el nodo está equilibrado y su subárbol izquierdo es un nivel más alto.

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

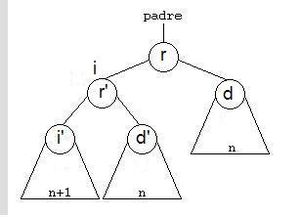
[](https://commons.wikimedia.org/wiki/File:BinaryTreeRotations.svg)

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}] .

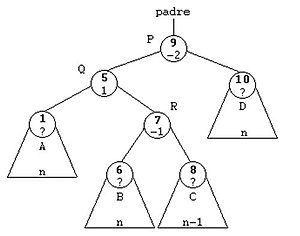
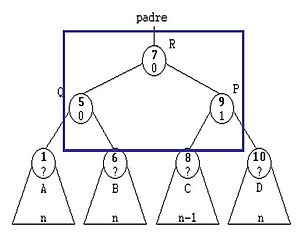
eq rotIzq(arbolBin(R1, I, arbolBin(R2, I2, D2))) ==

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.

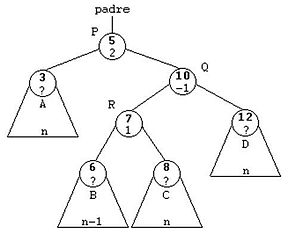
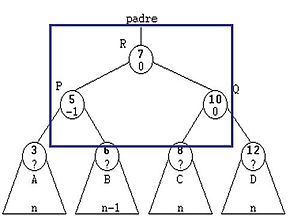
##### Rotación doble a la derecha

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.

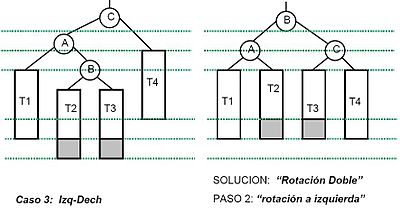
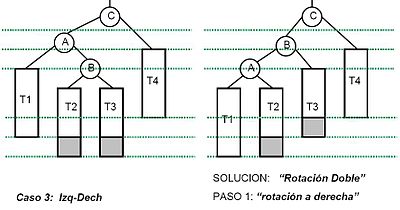
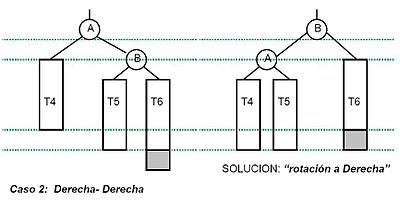
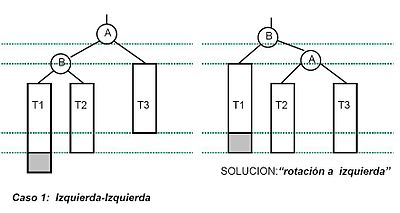
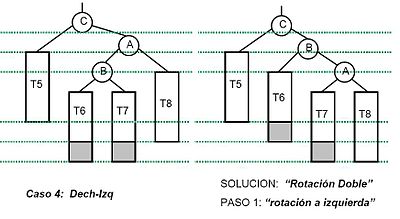
[](https://commons.wikimedia.org/wiki/File:ROTACIONIZDA2.jpg)[](https://commons.wikimedia.org/wiki/File:ROTACIONIZQ2.jpg)

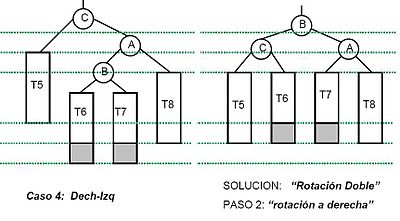
#### Inserción

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.

# Grafos

## Representaciones matriciales y basadas en listas

**Representación de grafos**

Existen diferentes formas de representar un  grafo, y hay muchos métodos para almacenarlos en una computadora. La estructura de datos usada dependerá de las características del grafo, y el algoritmo usado para manipularlo. Entre las más comunes esta las listas y matrices, con frecuencia se usa una combinación de ambas. **[5]**

### Estructura de lista

Lista de incidencia: El grafo está representado por una matriz de A (aristas) por V (vértices), donde [arista, vértice] contiene la información de la arista (conectado o no conectado). **[5]**

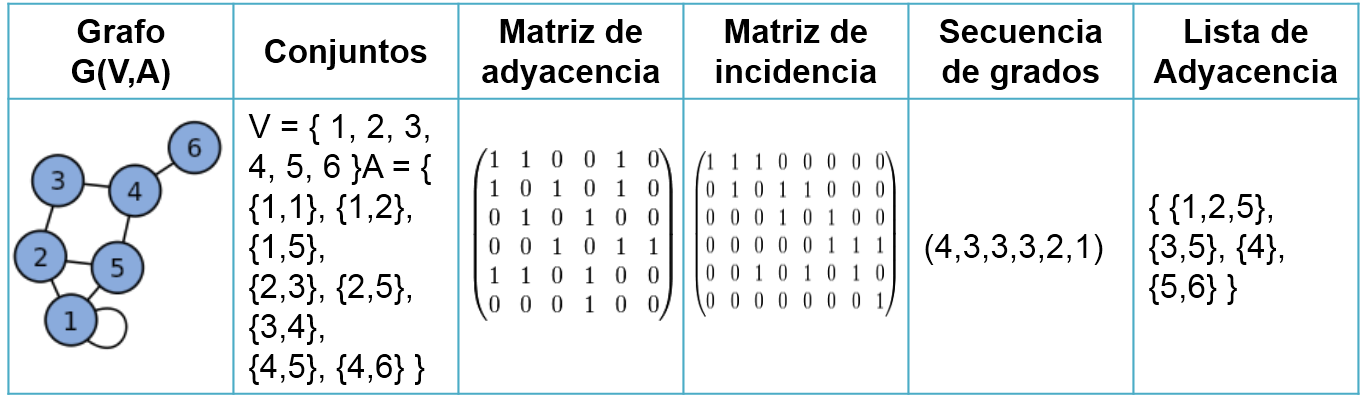
Lista de adyacencia: El grafo está representado por un arreglo de listas de adyacencia. Para un vértice i, la lista de adyacencia está formada por todos los vértices adyacentes a i. Puede construirse en tiempo lineal, y las inserciones pueden hacerse al principio de cada lista, con lo que se asegura tiempo constante. **[5]**

Lista de grados: También llamada secuencia de grados o sucesión gráfica de un una secuencia de números, que corresponde a los grados de los vértices del grafo. **[5]**

### Estructuras matriciales

Matriz de adyacencia: El grafo está representado por una matriz cuadrada M de tamaño n^2, donde n es el número de vértices. Si hay una arista entre un vértice x y un vértice y, entonces el elemento m\_ {x, y} es 1, de lo contrario, es 0. **[5]**

Matriz de incidencia: El grafo está representado por una matriz de A (aristas) por V (vértices), donde [vértice, arista] contiene la información de la arista (1 - conectado, 0 - no conectado) **[5]**

[](https://sites.google.com/site/matematicasmoralesgalindo/6-2-representacion-de-los-grafos/tabls.png?attredirects=0)

## Búsquedas de amplitud

En [Ciencias de la Computación](https://es.wikipedia.org/wiki/Ciencias_de_la_Computaci%C3%B3n), Búsqueda en anchura (en [inglés](https://es.wikipedia.org/wiki/Idioma_ingl%C3%A9s) BFS - Breadth First Search) es un [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) de [búsqueda no informada](https://es.wikipedia.org/wiki/B%C3%BAsquedas_no_informadas) utilizado para recorrer o buscar elementos en un [grafo](https://es.wikipedia.org/wiki/Grafo) (usado frecuentemente sobre [árboles](https://es.wikipedia.org/wiki/%C3%81rbol_(programaci%C3%B3n))). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol. **[6]**

Formalmente, BFS es un [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia [heurística](https://es.wikipedia.org/wiki/Heur%C3%ADstica). **[6]**

Si las aristas tienen pesos negativos aplicaremos el [algoritmo de Bellman-Ford](https://es.wikipedia.org/wiki/Algoritmo_de_Bellman-Ford) en alguna de sus dos versiones. **[6]**

### Procedimiento

* Dado un [vértice](https://es.wikipedia.org/wiki/V%C3%A9rtice_(Teor%C3%ADa_de_grafos)) fuente **s**, *Breadth-first search* sistemáticamente explora los vértices de G para “descubrir” todos los vértices alcanzables desde **s**.
* Calcula la distancia (menor número de vértices) desde **s** a todos los [vértices](https://es.wikipedia.org/wiki/V%C3%A9rtice_(Teor%C3%ADa_de_grafos)) alcanzables.
* Después produce un [árbol](https://es.wikipedia.org/wiki/%C3%81rbol_(teor%C3%ADa_de_grafos)) **BF** con raíz en **s** y que contiene a todos los [vértices](https://es.wikipedia.org/wiki/V%C3%A9rtice_(Teor%C3%ADa_de_grafos)) alcanzables.
* El camino desde **s** a cada vértice en este recorrido contiene el mínimo número de [vértices](https://es.wikipedia.org/wiki/V%C3%A9rtice_(Teor%C3%ADa_de_grafos)). Es el camino más corto medido en número de vértices.
* Su nombre se debe a que expande uniformemente la frontera entre lo descubierto y lo no descubierto. Llega a los nodos de distancia k, sólo tras haber llegado a todos los nodos a distancia k-1.

### Código

* La nomenclatura adicional utilizada es: Q = Estructura de datos cola

**BFS**(grafo G, nodo\_fuente s)

{

*// recorremos todos los vértices del grafo inicializándolos a NO\_VISITADO,*

*// distancia INFINITA y padre de cada nodo NULL*

**for** u ∈ V[G] **do**

{

estado[u] = NO\_VISITADO;

distancia[u] = INFINITO; /\* distancia infinita si el nodo no es alcanzable \*/

padre[u] = NULL;

}

estado[s] = VISITADO;

distancia[s] = 0;

padre[s] = NULL;

CrearCola(Q); /\* nos aseguramos que la cola está vacía \*/

Encolar(Q, s);

**while** !vacía(Q) **do**

{

*// extraemos el nodo* ***u*** *de la cola* ***Q*** *y exploramos todos sus nodos adyacentes*

u = extraer(Q);

**for** v ∈ adyacencia[u] **do**

{

**if** estado[v] == NO\_VISITADO **then**

{

estado[v] = VISITADO;

distancia[v] = distancia[u] + 1;

padre[v] = u;

Encolar(Q, v);

}

}

}

}

\*Falta recorrer vértices no adyacentes directa o indirectamente al vértice origen "s",

pues la cola queda vacía sin los adyacentes restantes.

* El tiempo de ejecución es O(|V|+|E|). Nótese que cada nodo es puesto a la cola una vez y su lista de adyacencia es recorrida una vez también.

## Búsqueda en profundidad

Una **Búsqueda en profundidad** (en [inglés](https://es.wikipedia.org/wiki/Idioma_ingl%C3%A9s) DFS o *Depth First Search*) es un [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) de [búsqueda no informada](https://es.wikipedia.org/wiki/B%C3%BAsquedas_no_informadas) utilizado para recorrer todos los nodos de un [grafo](https://es.wikipedia.org/wiki/Grafo) o [árbol (teoría de grafos)](https://es.wikipedia.org/wiki/%C3%81rbol_(teor%C3%ADa_de_grafos)) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa ([Backtracking](https://es.wikipedia.org/wiki/Backtracking)), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado. **[7]**

Análogamente existe el [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) de [búsqueda en anchura](https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura) (BFS o *Breadth First Search*).

### Evaluación

**Completitud**: DFS es completo si y solo si usamos búsqueda basada en grafos en espacios de estado finitos, pues todos los nodos serán expandidos. **[7]**

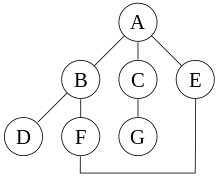
**Optimalidad**: DFS en ningún caso asegura la optimalidad, pues puede encontrar una solución más profunda que otra en una rama que todavía no ha sido expandida. **[7]**

**Complejidad tempora**l: en el peor caso, es {\displaystyle O(b^{m})}, siendo b el factor de ramificación (número promedio de ramificaciones por nodo) y m la máxima profundidad del espacio de estados. **[7]**

**Complejidad espacial**: {\displaystyle O(b^{d})}, siendo b el factor de ramificación y d la profundidad de la solución menos costosa, pues cada nodo generado permanece en memoria, almacenándose la mayor cantidad de nodos en el nivel meta. **[7]**

#### Ejemplo

Para el grafo siguiente:

[](https://commons.wikimedia.org/wiki/File:Graph.traversal.example.svg)

Una búsqueda en profundidad empezando en el nodo A, con la suposición que las aristas a la izquierda son escogidas antes de las aristas a la derecha, el algoritmo va a visitar los nodos en esta orden: A, B, D, F, E, C, G. Se puede notar que si el algoritmo no recuerde los nodos ya visitados, el algoritmo podría continuar en una vuelta infinita A, B, D, F, E, A, B, D, F, E, etc. sin visitar C o G**.[7]**

Para evitar esta vuelta infinita, puede usar técnicas como [búsqueda en profundidad iterativa](https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad_iterativa). **[7]**

### Pseudocódigo

#### Pseudocódigo para grafos

**DFS**(grafo G)

**PARA CADA** vértice *u* ∈ V[G] **HACER**

estado[*u*] ← NO\_VISITADO

padre[*u*] ← NULO

tiempo ← 0

**PARA CADA** vértice *u* ∈ V[G] **HACER**

**SI** estado[u] = NO\_VISITADO **ENTONCES**

DFS\_Visitar(*u*,*tiempo*)

**DFS\_Visitar**(nodo *u*, int *tiempo*)

estado[*u*] ← VISITADO

tiempo ← tiempo + 1

d[u] ← tiempo

**PARA CADA** *v* ∈ Vecinos[*u*] **HACER**

**SI** estado[*v*] = NO\_VISITADO **ENTONCES**

padre[*v*] ← *u*

DFS\_Visitar(*v*,*tiempo*)

estado[*u*] ← TERMINADO

tiempo ← tiempo + 1

*f[u]* ← tiempo

#### Código para grafos

*/\*\**

*Algorithm: DFS (Graph)*

*\*/*

#include *<bits/stdc++.h>*

#include *<stack>*

#define oo 1005

**using** **namespace** std;

int N, A, K, D[oo]; *///N: Cant. de Nodo A: Cant. de Aristas K: Cant. de Instrucciones*

bool Mk[oo];

vector<int> V[oo];

stack<int> Q;

char S; *///Caracter Separador*

void DFS ()

{

, newNod = 0, ady = 0;

Q.push(0);

**while**(!Q.empty())

{

nodo = Q.top();

Q.pop();

vector<int>::iterator i;

**for**(i = V[nodo].begin(); i != V[nodo].end(); i++)

{

ady = \*i;

**if**(Mk[ady])

**continue**;

Mk[ady] = true;

D[ady] = D[nodo] + 1;

Q.push(ady);

}

}

}

int main ()

{

freopen("DFS.in", "r", stdin);

freopen("DFS.out", "w", stdout);

int C\_1, C\_2;

cin >> N >> A >> K;

**for**(int i = 0; i < A; i++)

{

cin >> C\_1 >> C\_2;

V[C\_1].push\_back(C\_2);

V[C\_2].push\_back(C\_1);

}

D[0] = 0;

Mk[0] = true;

DFS();

**for**(int i = 0; i < K; i++)

{

cin >> C\_1;

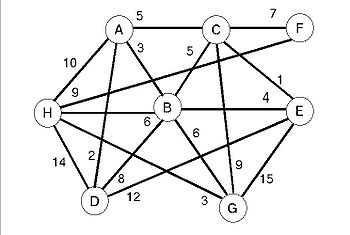
cout << D[C\_1] << "**\n**";

}

**return** 0;

}

## Problema del camino más corto

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

En la [teoría de grafos](https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos), el **problema del camino más corto** es el problema que consiste en encontrar un camino entre dos [vértices](https://es.wikipedia.org/wiki/V%C3%A9rtice_(Teor%C3%ADa_de_grafos)) (o nodos) de tal manera que la suma de los pesos de las [aristas](https://es.wikipedia.org/wiki/Arista_(Teor%C3%ADa_de_grafos)) que lo constituyen es mínima. Un ejemplo de esto es encontrar el camino más rápido para ir de una ciudad a otra en un mapa. En este caso, los vértices representarían las ciudades y las aristas las carreteras que las unen, cuya ponderación viene dada por el tiempo que se emplea en atravesarlas**.[8]**

### Definición

El problema del camino más corto puede ser definido para [grafos](https://es.wikipedia.org/wiki/Grafo) [no dirigidos](https://es.wikipedia.org/wiki/Grafo#Grafo_no_dirigido), [dirigidos](https://es.wikipedia.org/wiki/Grafo_dirigido) o mixtos. La siguiente es una definición para grafos no dirigidos, en el caso de grafos dirigidos la definición de camino requiere que los vértices adyacentes estén conectados por una apropiada arista dirigida**.[8]**

Dos vértices son adyacentes cuando poseen una arista común. Un [camino](https://es.wikipedia.org/wiki/Camino_(teor%C3%ADa_de_grafos)) en un grafo no dirigido es una secuencia de vértices {\displaystyle P=(v\_{1},v\_{2},\ldots ,v\_{n})\in V\times V\times \ldots \times V}tal que todo vértice {\displaystyle v\_{i}} es adyacente con el vértice {\displaystyle v\_{i+1}}. Un camino {\displaystyle P}se dice que es de longitud {\displaystyle n-1} si va desde {\displaystyle v\_{1}} hasta **[8]**{\displaystyle v\_{n}}

Sea {\displaystyle e\_{i,j}} la arista incidente con los vértices {\displaystyle v\_{i}} y {\displaystyle v\_{j}} dada una función de variable real ponderada {\displaystyle f:E\rightarrow \mathbb {R} } y un grafo no dirigido {\displaystyle G}, el camino más corto desde {\displaystyle v} hasta {\displaystyle v'} es el camino {\displaystyle P=(v\_{1},v\_{2},\ldots ,v\_{n})} sobre todos los posibles {\displaystyle n} que minimiza la suma. {\displaystyle \sum \_{i=1}^{n-1}f(e\_{i,i+1}).} Cuando cada arista en el grafo tiene un peso unitario o {\displaystyle f:E\rightarrow \{1\}}, hallar el camino más corto es equivalente a encontrar el camino con menor número de aristas. **[8]**

El problema es también conocido como el **problema de los caminos más cortos entre dos nodos**, para diferenciarlo de las siguientes generalizaciones:

**El problema de los caminos más cortos desde un origen**, en el cual tenemos que encontrar los caminos más cortos de un vértice origen *v* a todos los demás vértices del grafo. **[8]**

**El problema de los caminos más cortos con un destino**, en el cual tenemos que encontrar los caminos más cortos desde todos los vértices del grafo a un único vértice destino, esto puede ser reducido al problema anterior invirtiendo el orden. **[8]**

**El problema de los caminos más cortos entre todos los pares de vértices**, el cual tenemos que encontrar los caminos más cortos entre cada par de vértices (*v*, *v'*) en el grafo. **[8]**

# Conclusión.

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]:**

I. Gregorio and G. Estrada, “Transformación en Árboles Binarios Estructura de Datos” [Online]. Available: https://garciagregorio.webcindario.com/ed/transformacion\_en\_arboles\_binarios.pdf. [Accessed: 13-Jan-2021]

‌**[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]

**[5]**

“GRAFOS,” *Decsai.ugr.es*, 2021. [Online]. Available: https://decsai.ugr.es/~jfv/ed1/tedi/cdrom/docs/grafos.htm. [Accessed: 14-Jan-2021]

**[6]**

Colaboradores de los proyectos Wikimedia, “Búsqueda en anchura,” *Wikipedia.org*, 23-May-2005. [Online]. Available: https://es.wikipedia.org/wiki/B%C3%BAsqueda\_en\_anchura. [Accessed: 14-Jan-2021]

**[7]:**

Colaboradores de los proyectos Wikimedia, “Búsqueda en profundidad,” *Wikipedia.org*, 08-May-2005. [Online]. Available: https://es.wikipedia.org/wiki/B%C3%BAsqueda\_en\_profundidad. [Accessed: 14-Jan-2021]

‌**[8]:**

Colaboradores de los proyectos Wikimedia, “Problema del camino más corto,” *Wikipedia.org*, 18-May-2009. [Online]. Available: https://es.wikipedia.org/wiki/Problema\_del\_camino\_m%C3%A1s\_corto. [Accessed: 14-Jan-2021]

‌