Estructuras de Datos
Árboles

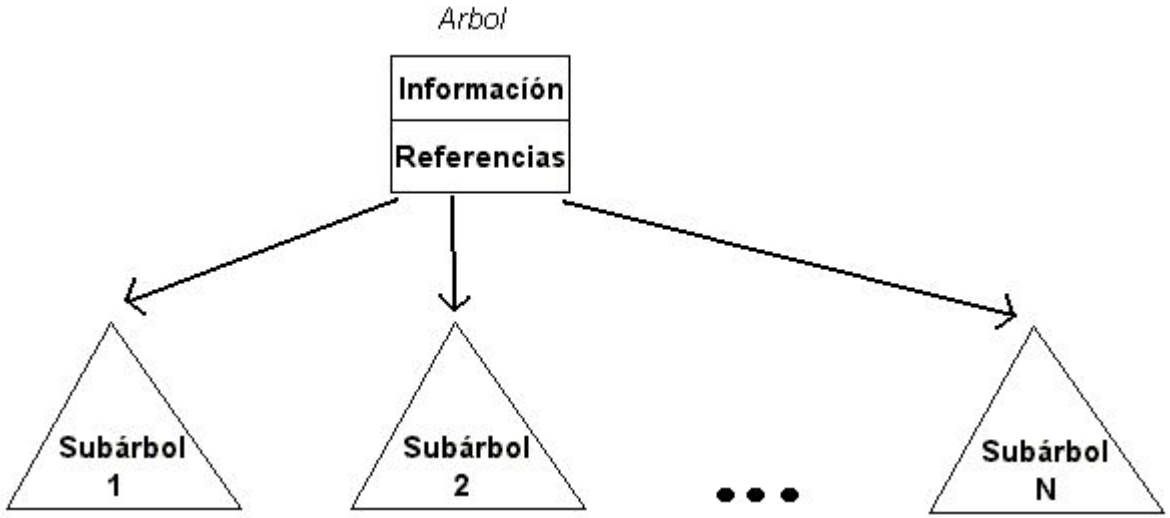
Definiciones

Un Árbol se define como una colección de nodos organizados en forma recursiva.

Cuando hay 0 nodos se dice que el árbol está vacío, en caso contrario el árbol consiste en un nodo denominado raíz, el cual tiene 0 o más referencias a otros árboles, conocidos como subárboles. Las raíces de los subárboles se denominan hijos de la raíz, y consecuentemente la raíz se denomina *padre* de las raíces de sus subárboles.

Definiciones

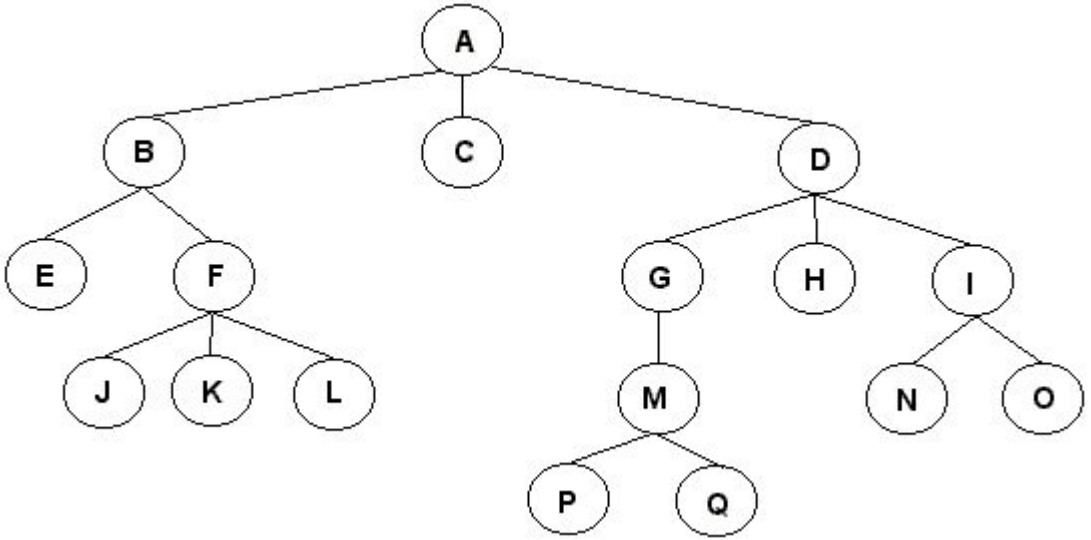
Gráficamente esta definición recursiva se muestra en la siguiente figura:



Los nodos que no poseen hijos se denominan hojas. Dos nodos que tienen el padre en común se denominan hermanos.

Definiciones

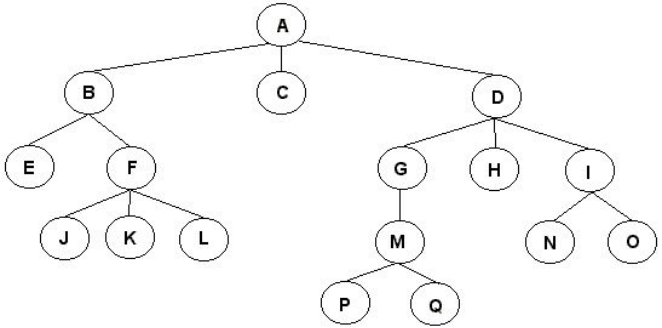
Si consideramos el siguiente árbol



Definiciones

Podemos afirmar que:

- A es la raíz del árbol.
- A es padre de B, C y D.
- E y F son hermanos, puesto que ambos son hijos de B.
- E, J, K, L, C, P, Q, H, N y O son las hojas del árbol.



Definiciones

Análogamente al concepto de árbol genealógico, podemos definir el concepto de ancestro y descendiente pero, para esto, necesitamos, previamente, darle forma a algunas ideas:

Un camino entre un nodo n_1 y un nodo n_k está definido como la secuencia de nodos $n_1, n_2, ..., n_k$ tal que n_i es padre de n_{i+1}, $1 <= i < k$.

El *largo del camino* es el número de referencias que componen el camino, que para el ejemplo son *k-1*.

Definiciones

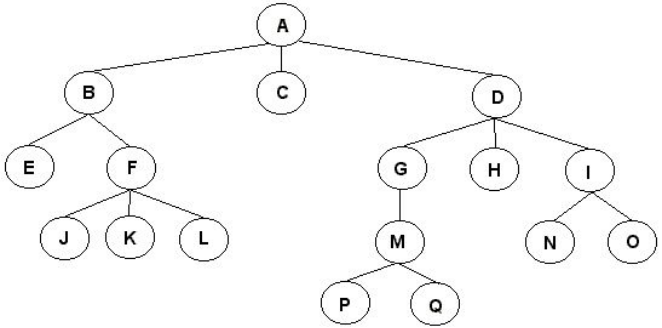
Convenimos que existe un camino desde cada nodo del árbol a sí mismo y es de largo 0.

*Una característica de los árboles que es que existe **un único** camino desde la raíz hasta cualquier otro nodo del árbol*.

Ahora sí podemos definir los conceptos de ancestro y descendiente: un nodo *n* es ancestro de un nodo *m* si existe un camino desde *n* a *m*; un nodo *n* es descendiente de un nodo *m* si existe un camino desde *m* a *n*.

Definiciones

Entonces podemos ver que:



- El camino desde A a J es único, lo conforman los nodos A-B-F-J y es de largo 3.
- D es ancestro de P, y por lo tanto P es descendiente de D.
- L no es descendiente de C, puesto que no existe un camino desde C a L.

Definiciones

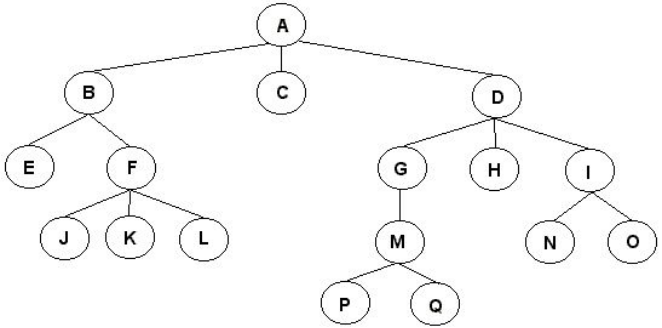
Se define la profundidad del nodo n_k como el largo del camino entre la raíz del árbol y el nodo n_k. Esto implica que la profundidad de la raíz es siempre 0.

La altura de un nodo n_k es el máximo largo de camino desde n_k hasta alguna hoja. Esto implica que la altura de toda hoja es 0.

La altura de un árbol es igual a la altura de la raíz, y tiene el mismo valor que la profundidad de la hoja más profunda. La altura de un árbol vacío se define como -1.

Definiciones

Con estas definiciones podemos concluir que:



- La profundidad de C es 1, de F es 2 y de Q es 4.
- La altura de C es 0, de F es 1 y de D es 3.
- La altura del árbol es 4 (largo del camino entre la raíz A y la hoja más profunda, P o Q).

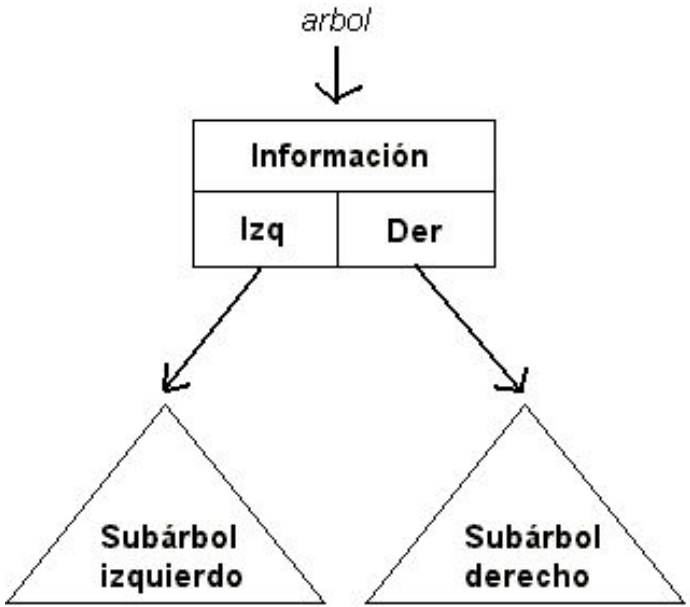
Estructuras de Datos
Árboles

Árbol Binario

Un árbol binario es un árbol en donde cada nodo posee exactamente *2 referencias* a subárboles. En general, dichas referencias se denominan izquierda y derecha, y consecuentemente se define el subárbol izquierdo y subárbol derecho del árbol.

Árbol Binario

La representación gráfica se vería así:



Árbol Binario

Yendo al código, podemos representar esta estructura de la siguiente forma:

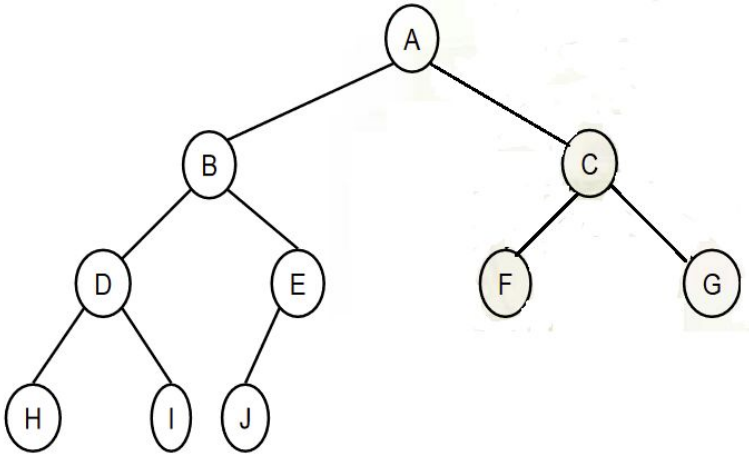
```
typedef struct _BTNodo {
  int dato;
  struct _BTNodo *left;
  struct _BTNodo *right;
} BTNodo;

typedef BTNodo *BTree;
```

vemos que, en este caso, la información a almacenar es un entero y, tengo dos punteros, uno al subárbol izquierdo y, otro al derecho.

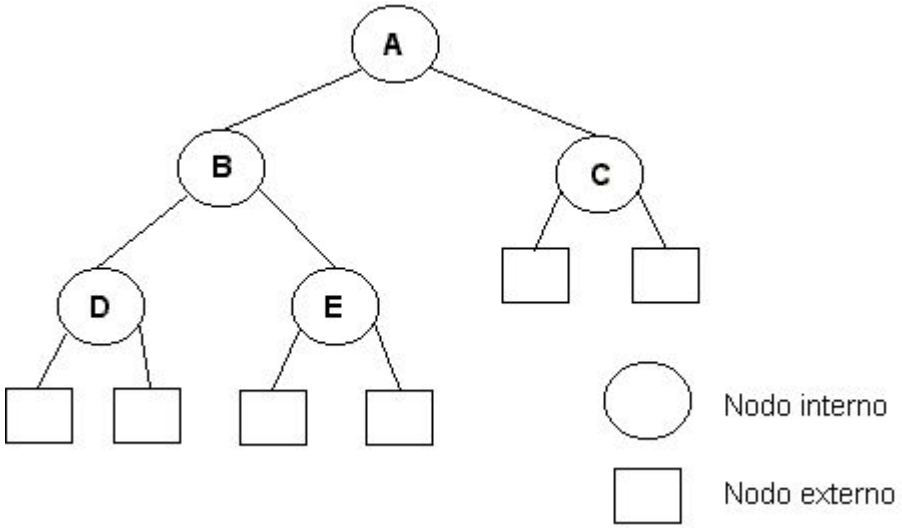
Árbol Binario

Un árbol binario de altura `k` está completo si está lleno hasta altura `k-1`, es decir, cada nodo tiene sus dos hijos y, el último nivel está ocupado de izquierda a derecha. Por ejemplo:



Árbol Binario

Los nodos en sí que conforman un árbol binario se denominan nodos internos, y todas las referencias que son *null* las llamamos nodos externos.



Árbol Binario - Propiedades

Podemos ver algunas propiedades de árboles binarios en base a esta definición:

1) Si se define i = número de nodos internos, e = número de nodos externos, entonces se tiene que:

$$e = i+1$$

Árbol Binario - Propiedades

Podemos ver algunas propiedades de árboles binarios en base a esta definición:

2) Sea *n = número de nodos internos*. Se define:

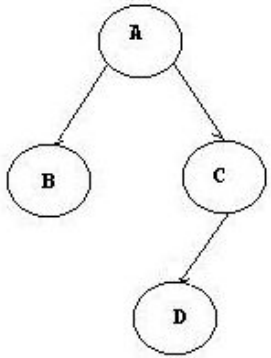
- I_n = suma del largo de los caminos desde la raíz a cada nodo interno (largo de caminos internos).
- E_n = suma del largo de los caminos desde la raíz a cada nodo externo (largo de caminos externos).

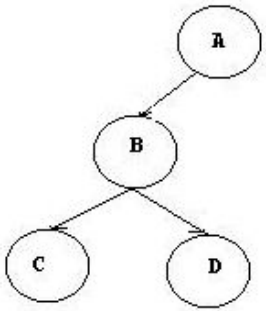
Se tiene que:

$$E_n = I_n + 2n$$

Árbol Binario - Propiedades

Diremos que dos árboles binarios son distintos si su estructura es distinta, es decir, si con la misma información existe, al menos, un nodo en diferente lugar.
En este caso podemos ver que el nodo C es hijo de A y en el árbol izquierdo y, es hijo de B en el árbol derecho.





Árbol Binario - Propiedades

Podemos ver algunas propiedades de árboles binarios en base a esta definición:

3) ¿Cuántos árboles binarios distintos (b_n) se pueden construir con *n* nodos internos? Para los primeros n números tenemos:

n	b_n
0	1
1	1
2	2
3	5

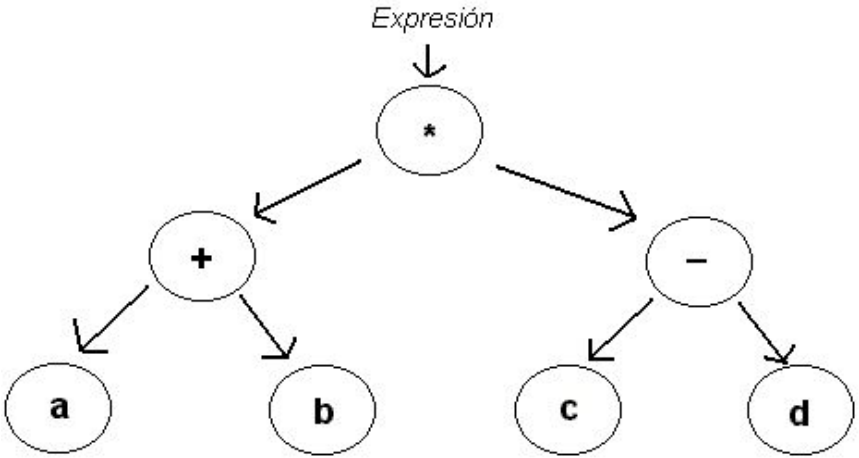
pero, generalizando, este número es: $$b_n = \frac{1}{n+1}\binom{2n}{n}$$

La serie de números que genera b_n se conoce como números de Catalan y converge a

$$b_n \approx \frac{4^n}{n\sqrt{\pi n}}$$

Árbol Binario - Árboles de Expresiones Aritméticas

Veremos un ejemplo de árboles binarios: los árboles de expresiones aritméticas.
En un árbol de expresiones las hojas corresponden a los *operandos* de la expresión (variables o constantes), mientras que los nodos restantes contienen *operadores*. Dado que los operadores matemáticos son binarios (o unarios como en el caso del operador negación), un árbol de expresiones resulta ser un árbol binario.



Árbol Binario - Árboles de Expresiones Aritméticas

Usando esta idea podemos construir un árbol para la expresión aritmética:

$$((7-a)/5)\times((a+b)\uparrow 3)$$

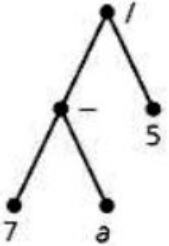
donde ↑ representa la potencia.

En este caso vamos a construir el árbol que representa la expresión de las hojas a la raíz.

Árbol Binario - Árboles de Expresiones Aritméticas

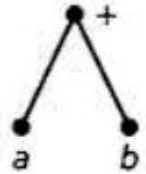
En primer lugar construimos un subárbol para la expresión `7-a`:

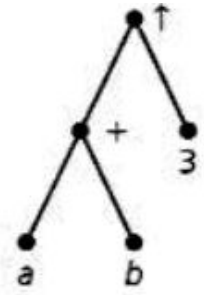
Una vez que lo tenemos, vamos a crear el subárbol de la expresión `(7-a)/5` que tiene al anterior como subárbol izquierdo.



Árbol Binario - Árboles de Expresiones Aritméticas

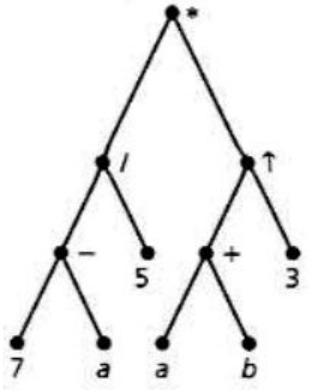
Luego, de modo similar, construimos los árboles binarios correspondientes a la expresión: (a+b)↑3.





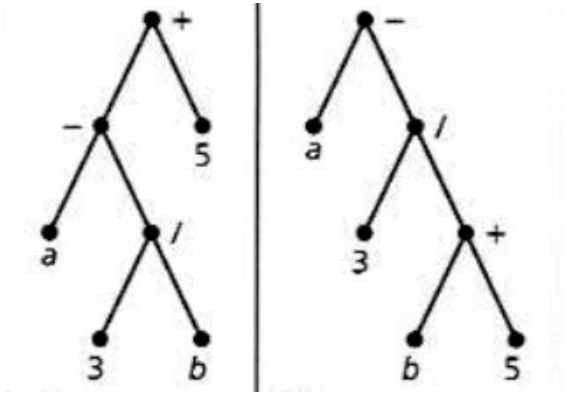
Árbol Binario - Árboles de Expresiones Aritméticas

Obteniendo, finalmente, el árbol de la expresión aritmética buscado:



Árbol Binario - Árboles de Expresiones Aritméticas

Los siguientes árboles binarios representan las expresiones: `(a-(3/b))+5` y `a-(3/(b+5))`



Árbol Binario - Recorridos

Un árbol de expresiones, se puede evaluar de la siguiente forma:

- Si la raíz del árbol es una constante o una variable se retorna el valor de ésta.
- Si la raíz resulta ser un operador, entonces recursivamente se evalúan los subárboles izquierdo y derecho, y se retorna el valor que resulta al operar los valores obtenidos de las evaluaciones de los subárboles con el operador respectivo.

Árbol Binario - Recorridos

Podemos ver que, al ser el árbol una estructura recursiva los algoritmos que se aplican sobre ellos suelen serlo también.

Esto es porque el algoritmo está escrito como una función que toma como dato la raíz del árbol que está procesando.

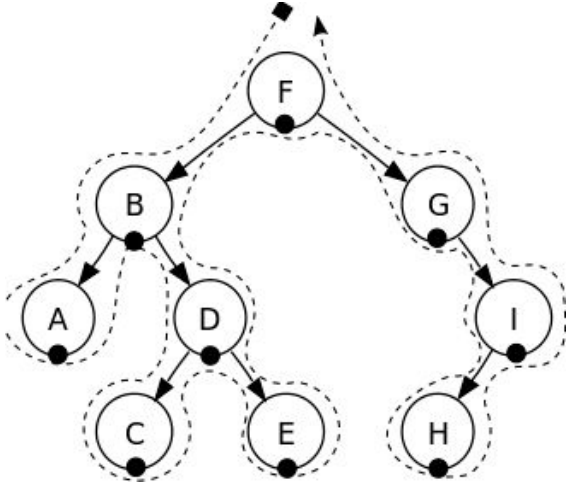
Árbol Binario - Recorridos

El primer algoritmo que vamos a ver para recorrer un árbol binario se denomina DFS (depth-first search) al que se llama Búsqueda en profundidad.

La idea de este recorrido es seguir un camino desde la raíz tan profundo como sea posible, cuando no es posible continuar este camino (por ejemplo cuando estamos en una hoja), se retrocede hasta el último nodo del camino en donde sea posible tomar una decisión diferente, y se avanza en ese otro sentido.

Árbol Binario - Recorridos

La siguiente imagen ilustra este recorrido.



Árbol Binario - Recorridos

Usando el recorrido podemos tener diferentes formas de visitar los vértices que forman un árbol binario. Las 3 formas usan DFS pero, lo que cambia en cada una de ellas es el momento en que se muestra la información del vértice en que estamos.

Debido a la expresión resultante obtenida es que estos recorridos se denominan: preorder, inorder y postorder.

A grandes rasgos, los mismos consisten en:

- *Preorden*: procesar la raíz - subárbol izquierdo - subárbol derecho.
- *Inorden*: subárbol izquierdo - procesar la raíz - subárbol derecho.
- *Postorden*: subárbol izquierdo - subárbol derecho - procesar la raíz.

La expresión que se obtiene con el recorrido en *postorden* se conoce como *notación polaca inversa*.

Árbol Binario - Recorridos

Algoritmo $preorder(s)$

Si s es vacío $Entonces$

$Retornar$

$Mostrar$ el valor de s

$L \coloneqq hijo\ izquierdo\ de\ s$

$preorder(L)$

$R \coloneqq hijo\ derecho\ de\ s$

$preorder(R)$

Árbol Binario - Recorridos

Ahora, analizaremos el algoritmo para algunos casos sencillos:

1) Si el árbol binario es vacío, nada se procesa pues, en este caso, el algoritmo termina simplemente en la línea 2.

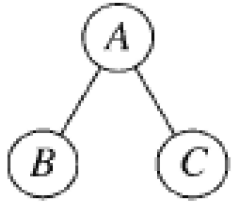
Árbol Binario - Recorridos

2) En este caso consideremos que ya contamos con un vértice. Es decir:

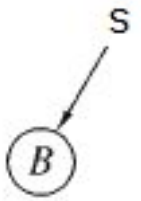
Supongamos que la entrada consta de un árbol con un único vértice. Hacemos s igual a la raíz y llamamos a $preorder(s)$. Como s no es vacío pasamos a la línea 3 donde mostramos el valor de la raíz. En la línea 5, llamamos a $preorder$ con s siendo el hijo izquierdo (vacío) de la raíz. Sin embargo hemos dicho que en la entrada de un árbol vacío $preorder$ no muestra información alguna. De manera análoga, en la línea 7 utilizamos un árbol vacío como entrada de $preorder$ y nada se procesa. Así cuando la entrada consta de un árbol con un único vértice, mostramos la raíz y terminamos.

Árbol Binario - Recorridos

3) Ahora, supongamos que la entrada es el siguiente árbol:



Hacemos s igual a la raíz y llamamos a $preorder(s)$. Como s no es vacío pasamos a la línea 3 donde mostramos el valor de la raíz (A). En la línea 5, llamamos a $preorder$ con s siendo el hijo izquierdo de la raíz:

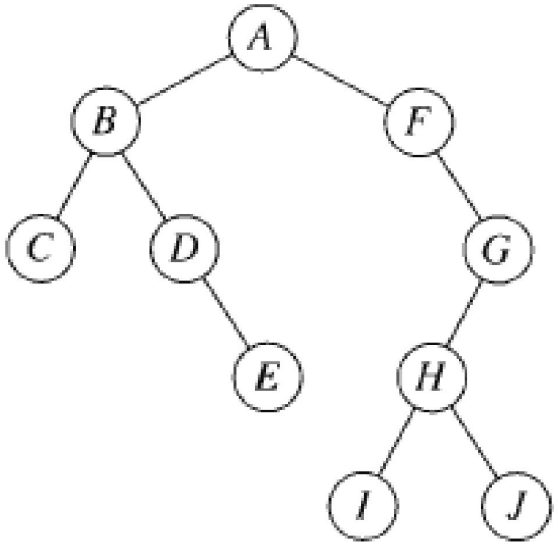


Acabamos de ver que si la entrada de $preorder$ consta de un único vértice, $preorder$ muestra ese vértice. Así, a continuación, mostramos el vértice B. De manera análoga en la línea 7, mostramos el vértice C.

Cuando el algoritmo termina, la salida obtenida es : ABC.

Árbol Binario - Recorridos

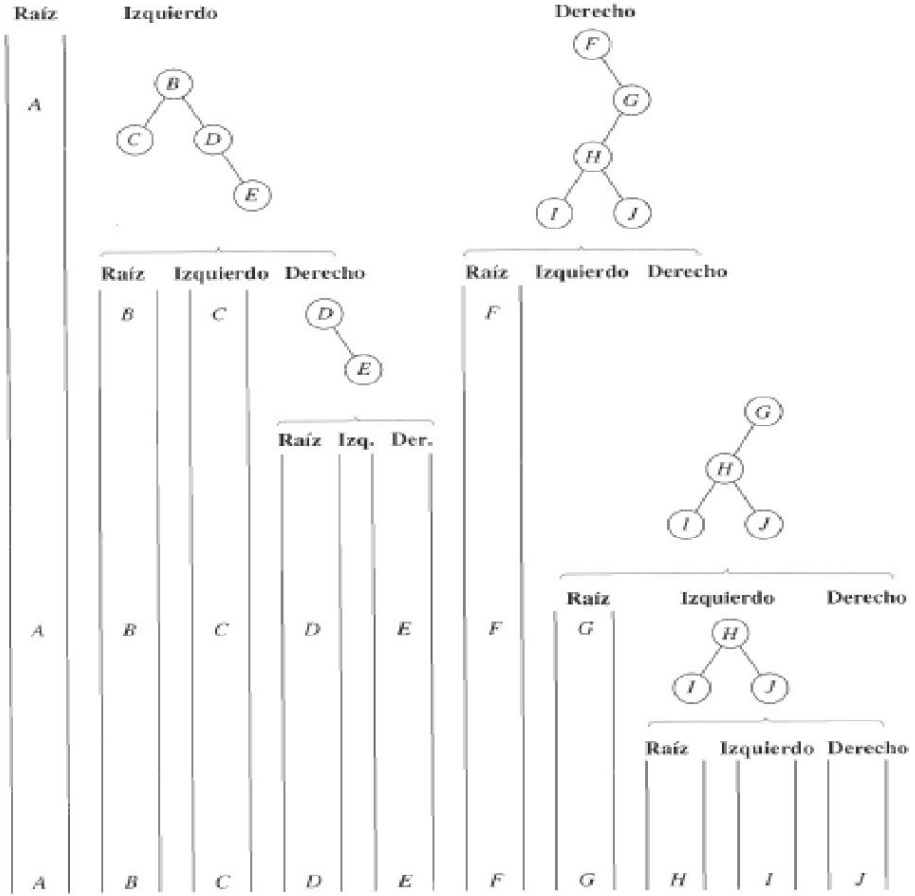
¿Cuál sería la salida del algoritmo $preorder$ con el siguiente árbol?



Árbol Binario - Recorridos

Si vamos siguiendo las líneas $3-7$ $(raíz/izquierda/derecha)$, del algoritmo $preorder$ el recorrido se muestra como indica la figura.

Así, la salida del algoritmo es : $ABCDEFGHIJ$.



Árbol Binario - Recorridos

Los recorridos $inorder$ y $postorder$ se obtienen solamente cambiando la posición de la línea 3 (raíz). Los prefijos pre, in y $post$ se refieren a la posición de la raíz en el recorrido; es decir, $preorder$ significa que primero va la raíz, $inorder$ quiere decir que la raíz va luego de mostrar el subárbol de la izquierda y, $postorder$ significa que la raíz va al final luego de mostrar el subárbol de la izquierda y, el de la derecha.

Árbol Binario - Recorridos

Algoritmo $inorder(s)$

Si s es vacío $Entonces$

$Retornar$

$L \coloneqq hijo\ izquierdo\ de\ s$

$inorder(L)$

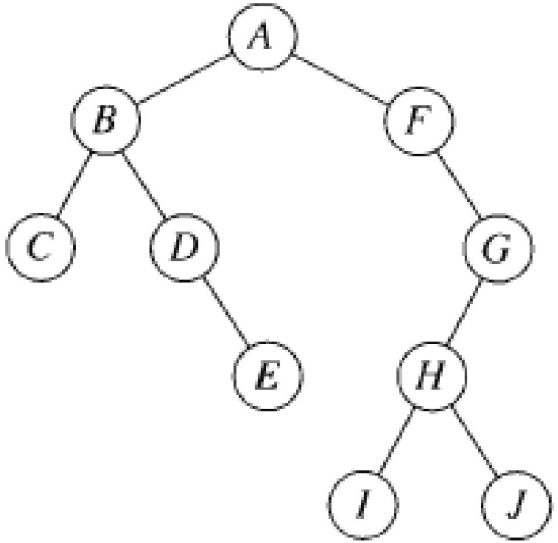
$Mostrar$ el valor de s

$R \coloneqq hijo\ derecho\ de\ s$

$inorder(R)$

Árbol Binario - Recorridos

¿Cuál sería la salida del algoritmo $inorder$ con el siguiente árbol?



Si vamos siguiendo las líneas $3 - 7$ $(izquierda/raíz/derecha)$, del algoritmo $inorder$ obtenemos como salida: $CBDEAFIHJG$.

Árbol Binario - Recorridos

Algoritmo $postorder(s)$

Si s es vacío $Entonces$

$Retornar$

$L \coloneqq hijo\ izquierdo\ de\ s$

$postorder(L)$

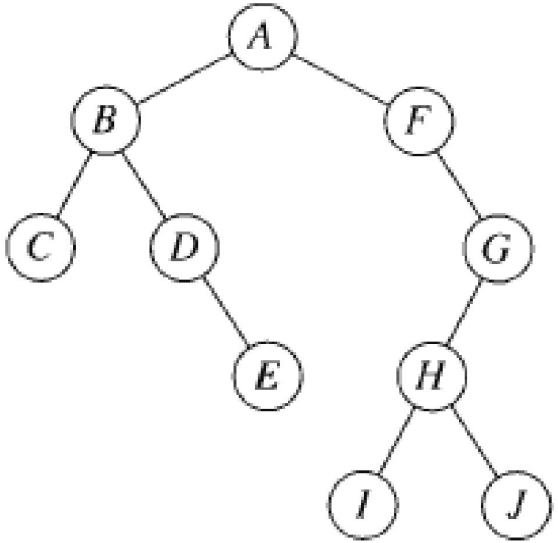
$R \coloneqq hijo\ derecho\ de\ s$

$postorder(R)$

$Mostrar$ el valor de s

Árbol Binario - Recorridos

¿Cuál sería la salida del algoritmo $postorder$ con el siguiente árbol?



Si vamos siguiendo las líneas $3-7$ $(izquierda/derecha/raíz)$, del algoritmo $postorder$ obtenemos como salida: $CEDBIJHGFA$.

Árbol Binario - Recorridos

Si volvemos al árbol del slide 10 cuando vimos expresiones aritméticas, y aplicamos los recorridos dados tendríamos:

en preorden

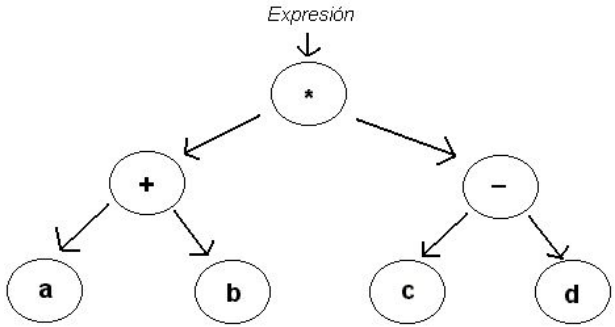
** + a b - c d*

en inorden

*a + b * c - d*

en postorden

*a b + c d - **

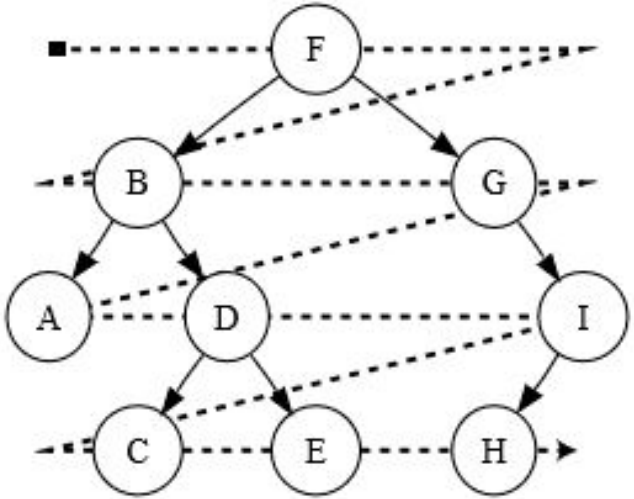


Árbol Binario - Recorridos

Otra forma de recorrer un árbol binario es usando el algoritmo BFS (Breadth-first search) el cual se conoce como Búsqueda por extensión o “a lo ancho”.
Es decir, se recorre por niveles.

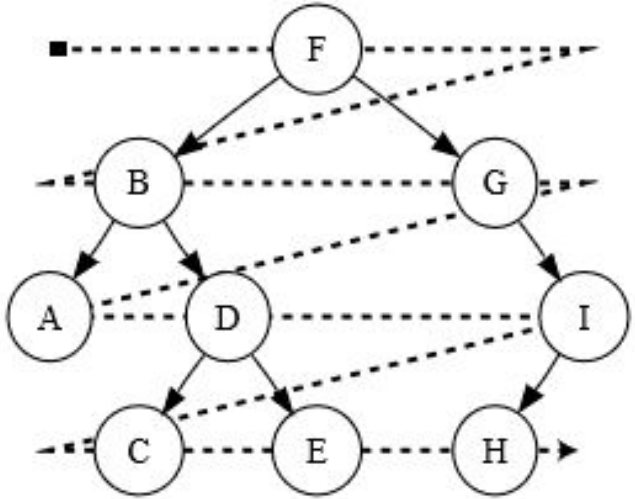
Árbol Binario - Recorridos

El siguiente árbol muestra el recorrido BFS



Árbol Binario - Recorridos

Como podemos ver comenzamos desde la raíz y luego por niveles, de izquierda a derecha, se va recorriendo el árbol.



Estructuras de Datos
Árboles

Tópicos

- Búsqueda binaria
- Árboles de búsqueda binaria
  - Búsqueda en un ABB
  - Inserción en un ABB
  - Eliminación en un ABB

Búsqueda binaria

Supongamos que se dispone del arreglo a, de tamaño n en el cual queremos buscar un elemento x. El mismo puede estar o no presente en el arreglo.
Para realizar esto podemos usar una Búsqueda Lineal, es decir, vamos recorriendo secuencialmente cada elemento comparándolo con x hasta que sea encontrado o hasta que todos los elementos hayan sido comparados.

Búsqueda binaria

Pensemos ahora qué sucedería si ahora nuestro arreglo a, de tamaño n tiene almacenado un conjunto de elementos ordenados de menor a mayor.
¿Podemos seguir haciendo de la Búsqueda Lineal? La respuesta es que sí pero, con la mejora que, en caso de encontrar un elemento mayor al buscado finalice. En ese caso ya sabemos que el elemento buscado x no se encuentra en el arreglo a.

Búsqueda binaria

Sin embargo existe una variante de la Búsqueda Lineal llamada Búsqueda binaria o dicotómica sobre arreglos ordenados la cual consiste en:

para buscar un elemento `x` dentro del arreglo `a`, de tamaño `n` se debe:

- buscar el índice de la posición media del arreglo (en donde se busca el elemento) que se denotará `m`. Inicialmente, `m = n/2`.
- si `a[m]=x` se encontró el elemento (fin de la búsqueda), en caso contrario se sigue buscando en el lado derecho o izquierdo del arreglo dependiendo si `a[m]<x` ó `a[m]>x` respectivamente.

Búsqueda binaria

La variable `i` representa el primer casillero del arreglo en donde es posible que se encuentre el elemento x, la variable `j` representa el último casillero del arreglo hasta donde x puede pertenecer, con `j>=i`.

Inicialmente, `i=0` y `j=n-1`.

Búsqueda binaria

En cada iteración,

- Si el conjunto es vacío (`j-i < 0`), o sea si `j < i`, entonces el elemento `x` no está en el conjunto (búsqueda infructuosa).
- En caso contrario, `m = (i+j)/2`.
  Si `x=a[m]`, el elemento fue encontrado (búsqueda exitosa).
  Si `x<a[m]` se modifica `j=m-1`, sino se modifica `i=m+1` y se sigue iterando.

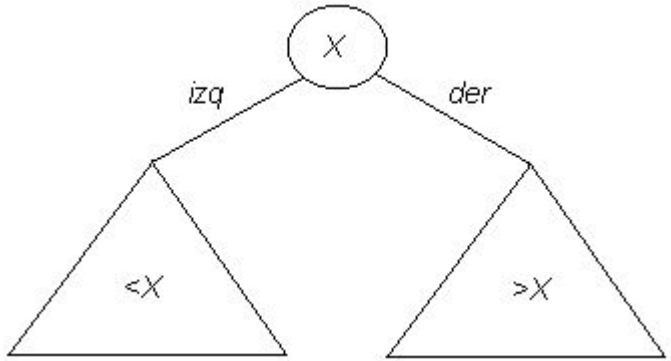
Búsqueda binaria

A continuación se muestra una versión iterativa del algoritmo. Se deja para pensar la versión recursiva del mismo.

```
#define NO_ENCONTRADO -1
int binary_search(int* a, int n, int x) {
  int i=0, j=n-1, m, posicion = NO_ENCONTRADO;
  while (i<=j && posicion == NO_ENCONTRADO) {
    m=(i+j)/2;
    if (x==a[m]) posicion = m;
    else if (x<a[m]) j=m-1;
    else i=m+1;
  }
  return posicion;
}
```

Árboles de búsqueda binaria

Un Árbol de búsqueda binaria, en adelante ABB, es un árbol binario en donde todos los nodos cumplen la siguiente propiedad (sin perder generalidad se asumirá que los elementos almacenados son números enteros): si el valor del elemento almacenado en un nodo N es X, entonces todos los valores almacenados en el subárbol izquierdo de N son menores que X, y los valores almacenados en el subárbol derecho de N son mayores que X.



Búsqueda en un ABB

Esta operación retorna un puntero al nodo en donde se encuentra el elemento buscado, x, o NULL si dicho elemento no se encuentra en el árbol. La estructura del árbol facilita la búsqueda:

- Si el árbol está vacío, entonces el elemento no está y se retorna NULL.
- Si el árbol no está vacío y el elemento almacenado en la raíz es X, se encontró el elemento y se retorna un puntero a dicho nodo.
- Si X es menor que el elemento almacenado en la raíz se sigue buscando recursivamente en el subárbol izquierdo, y si X es mayor que el elemento almacenado en la raíz se sigue buscando recursivamente en el subárbol derecho.

Búsqueda en un ABB

Como podemos ver esta función es recursiva y, podemos escribirla de la siguiente manera tomando la definición de BTree comentada en la presentación anterior:

```
BTree btree_search(BTree nodo, int dato){
    if (btree_empty(nodo))
        return NULL;
    if (nodo->dato == dato)
        return nodo;
    if (nodo->dato>dato)
        return btree_search(nodo->left, dato);
    else
        return btree_search(nodo->right, dato);
}
```

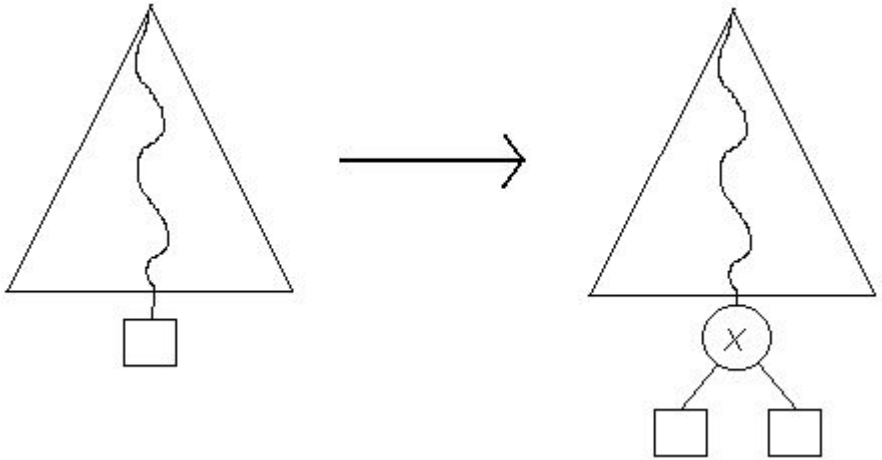
Búsqueda en un ABB

Otra versión sería, usando el if ternario:

```
BTree btree_search(BTree nodo, int dato){
    if (btree_empty(nodo) || nodo->dato == dato) return nodo;
    return nodo->dato>dato?btree_search(nodo->left, dato):btree_search(nodo->right, dato);
}
```

Inserción en un ABB

Para insertar un elemento X en un ABB, es equivalente a buscarlo, ya que nos posicionamos en el lugar donde debería haber estado usando la propiedad de ABB.



Inserción en un ABB

El código sería:

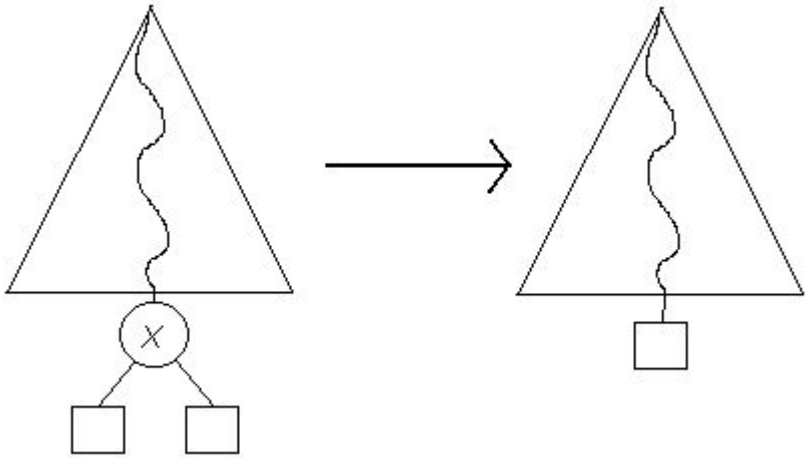
```
BTree btree_insert(BTree nodo, int dato) {
    if (btree_empty(nodo)) {
        nodo = malloc(sizeof(BTNodo));
        nodo->left = NULL;
        nodo->right = NULL;
    }
    else if (nodo->dato > dato)
        nodo->izq = btree_insert(nodo->izq, dato);
    else
        nodo->der = btree_insert(nodo->der, dato);

    return nodo;
}
```

Eliminación en un ABB

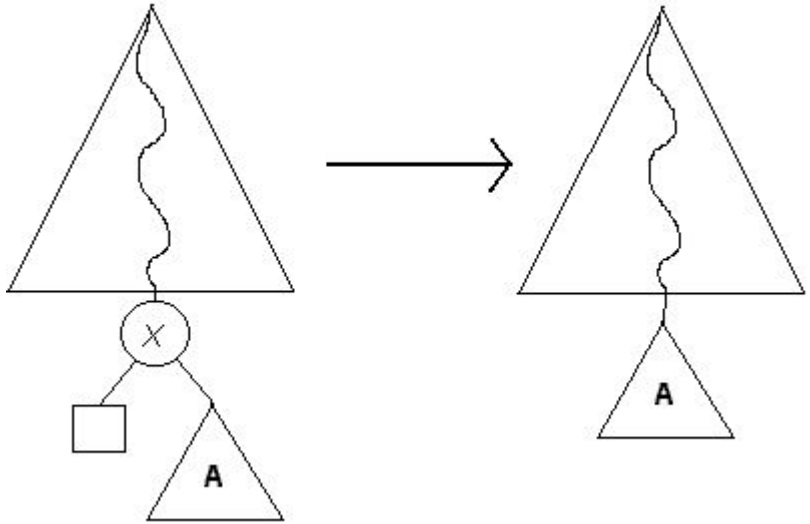
Primero se realiza una búsqueda del elemento a eliminar, digamos X. Si la búsqueda fue infructuosa no se hace nada, en caso contrario hay que considerar los siguientes casos posibles:

1. Si X es una hoja sin hijos, se puede eliminar inmediatamente.



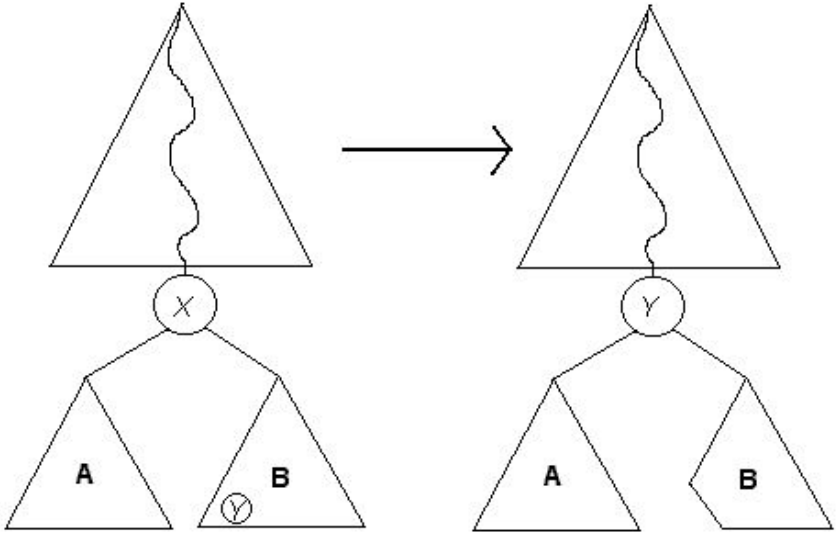
Eliminación en un ABB

2. Si X tiene un solo hijo, entonces se cambia la referencia del padre a X para que ahora referencie al hijo de X.



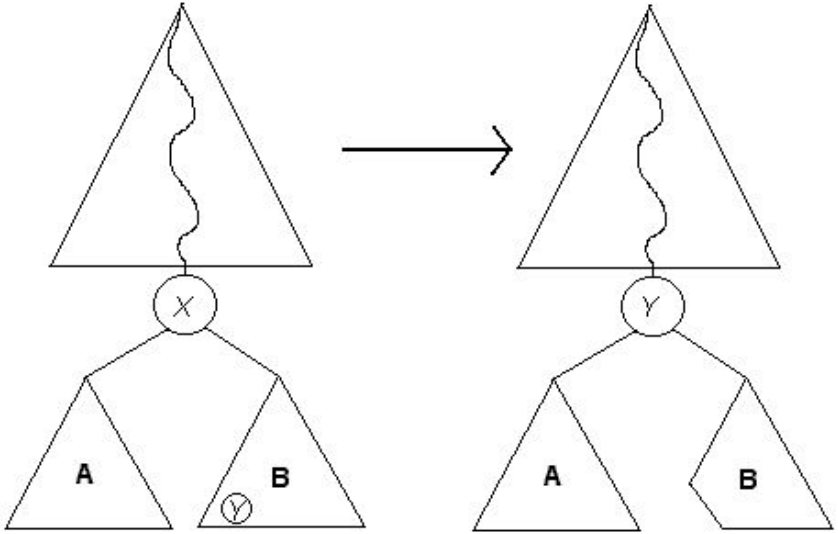
Eliminación en un ABB

3. Si X tiene dos hijos, es el caso complicado, tenemos que analizar qué dato se puede ubicar en el lugar de X para mantener la propiedad de ABB y que implique la menor cantidad de cambios posible.
   Si tomamos a Y = mínimo nodo del subárbol derecho de X, es decir, sería el nodo más a la izquierda del subárbol derecho de X.



Eliminación en un ABB

3. Este nodo Y tiene la característica de ser menor que todos los que están en el subárbol derecho de X y, mayor que los que están en el subárbol izquierdo de X. Es decir, lo podemos ubicar directamente en el lugar de X y seguir preservando la propiedad de ABB.
   Análogamente, se podría hacer tomando el máximo nodo del subárbol izquierdo de X.



Estructuras de Datos
Árboles

Cola de Prioridad

Una Cola de prioridad es una variante del tipo de datos Cola que mencionamos anteriormente. La característica que lo distingue es que los datos poseen una clave perteneciente a un conjunto ordenado.

Cola de Prioridad

Las funciones que posee permiten

- *insertar* nuevos elementos;
- *extraer el máximo* (o el mínimo, en caso de que la estructura se organice con un criterio de orden inverso).

Frecuentemente los valores de las claves se ven como prioridades, con lo cual la estructura permite insertar elementos de cualquier prioridad, y extraer el de mayor prioridad.

Cola de Prioridad

La *cola de prioridad* puede ser implementada con:

- Una lista ordenada:
  - Inserción: O(n) para poder insertar tenemos que recorrer la lista, en orden secuencial, nodo a nodo, hasta encontrar el lugar que le corresponda.
  - Extracción de máximo: O(1) estaría en el primer lugar de la lista al cual accedemos en forma directa.
- Una lista desordenada:
  - Inserción: O(1) insertamos el dato en cualquier lugar.
  - Extracción de máximo: O(n) debemos hallar el máximo en una lista desordenada, esto implica recorrer toda la lista.

Cola de Prioridad

¿Podemos usar arrays para implementar una Cola de prioridad? Analicemos un poco esto.
Supongamos que usamos un array ordenado.

Si queremos insertar un dato tenemos que recorrer el array, esto lo podemos hacer usando búsqueda binaria para posicionarnos en el lugar adecuado pero después, tendríamos que desplazar todos los datos siguientes para poder mantener el orden de prioridades. Por lo tanto, si bien, encontraríamos más rápido la posición donde insertar (O(log n)) tendríamos que desplazar todos los elementos restantes del arreglo.

Debido a esto es que no es una buena opción pensar en un array ordenado.

Cola de Prioridad

¿Podemos usar arrays para implementar una Cola de prioridad? Supongamos que usamos un array sin orden.

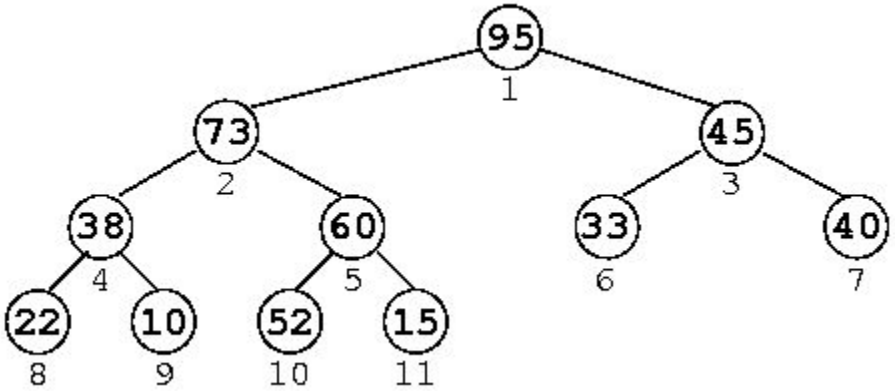
En este caso:

- Inserción: O(1) insertamos el dato en cualquier lugar.
- Extracción de máximo: O(n) debemos hallar el máximo en un array desordenado, esto implica recorrer todo el array.

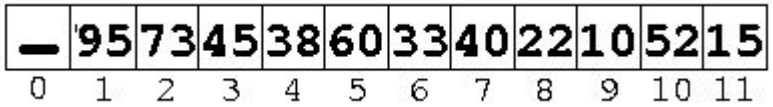
Heap

Un Heap es un árbol binario completo, que permite su almacenamiento en un arreglo sin usar punteros. ¿Cómo se puede hacer esto? Lo veremos a continuación pero, recordemos que un árbol binario completo tiene todos sus niveles llenos, excepto posiblemente el de más abajo, y en este último los nodos están lo más a la izquierda posible.

Supongamos que tenemos el siguiente árbol binario completo:



La numeración por niveles (indicada bajo cada nodo) son las posiciones en donde cada elemento sería almacenado en el arreglo. Dejaremos la primera posición (índice 0) sin usar, para simplificar algunos cálculos que veremos a continuación. En el caso del ejemplo dado, el arreglo sería:



Heap

La característica que permite que un Heap se pueda almacenar sin punteros es que, si se utiliza la numeración por niveles indicada, entonces la relación entre padres e hijos es:

$$\text{Hijos del nodo } j = \{2*j,\ 2*j+1\}$$
$$\text{Padre del nodo } k = \text{floor}(k/2)$$

Heap

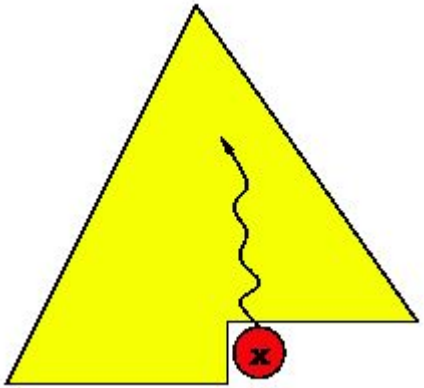
Un Heap puede utilizarse para implementar una Cola de prioridad almacenando los datos de modo que las claves estén siempre ordenadas de arriba a abajo (a diferencia de un árbol de búsqueda binaria, que ordena sus claves de izquierda a derecha). En otras palabras, el padre debe tener siempre mayor prioridad que sus hijos (ver el slide 8).

Inserción en un Heap

La inserción se realiza agregando el nuevo elemento en la primera posición libre del heap, esto es, el próximo nodo que debería aparecer en el recorrido por niveles o, equivalentemente, un casillero que se agrega al final del arreglo.

Inserción en un Heap

Después de agregar este elemento, la *estructura* del Heap se preserva, pero la restricción de orden no tiene por qué cumplirse. Para resolver este problema, si el nuevo elemento es mayor que su padre (recordemos que, por la propiedad del Heap , el padre es mayor que el otro hijo), se intercambia con él, y ese proceso se repite mientras sea necesario. Una forma de describir esto es diciendo que el nuevo elemento "trepa" en el árbol hasta alcanzar el nivel correcto según su prioridad.



Inserción en un Heap

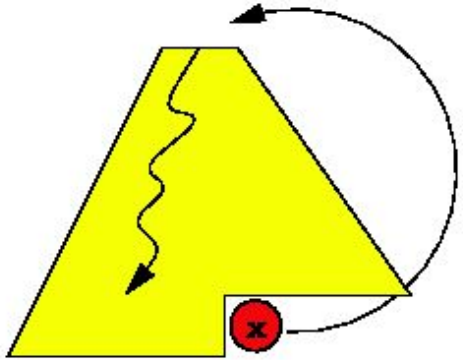
El siguiente bloque de código muestra el proceso de inserción de un nuevo elemento x:

```
//n representa la cantidad de elementos del heap
a[++n]=x;
for(j=n; j>1 && a[j]>a[j/2]; j/=2)
  { //intercambiamos con el padre
    t=a[j];
    a[j]=a[j/2];
    a[j/2]=t;
  }
```

El proceso de inserción, en el peor caso, toma un tiempo proporcional a la altura del árbol, esto es, O(log n).

Extracción del máximo en un Heap

El máximo evidentemente está en la raíz del árbol (casillero 1 del arreglo). Al sacarlo de ahí, podemos imaginar que ese lugar queda vacante. Para llenarlo, tomamos al *último* elemento del Heap y lo trasladamos al lugar vacante. En caso de que no esté bien ahí de acuerdo a su prioridad (¡que es lo más probable!), lo hacemos descender intercambiándolo siempre con el mayor de sus hijos. Decimos que este elemento "se hunde" hasta su nivel de prioridad.



Extracción del máximo en un Heap

El siguiente fragmento de programa implementa este algoritmo:

```
int esMayor =1;
m=a[1];  //La variable m lleva el máximo
a[1]=a[n--];  //Movemos el último a la raíz y achicamos el heap
j=1;
while(2*j<=n && esMayor) //mientras tenga algún hijo
{
    k=2*j; //el hijo izquierdo
    if(k+1<=n && a[k+1]>a[k])
        k=k+1;  //el hijo derecho es el mayor
    if(a[j]>a[k])
        esMayor = 0;  //es mayor que ambos hijos
    else {
        t=a[j];
        a[j]=a[k];
        a[k]=t;
        j=k;   //lo intercambiamos con el mayor hijo
    }
}
```

Este algoritmo también demora un tiempo proporcional a la altura del árbol en el peor caso, esto es, O(log n).

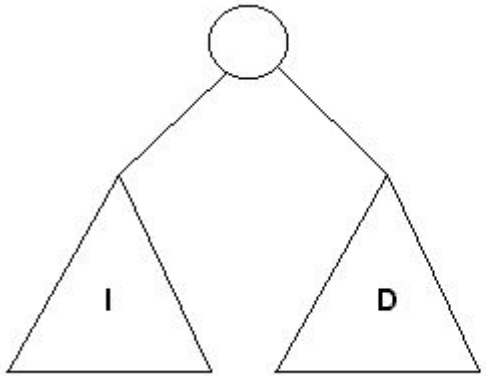
Estructuras de Datos
Árboles

Tópicos

- Árboles AVL
  - Inserción en un AVL
    - Rotación Simple
    - Rotación Doble
  - Eliminación en un AVL

Árboles AVL

Un ABB se dice que es un Árbol AVL si para todo nodo interno la diferencia de altura de sus dos árboles hijos es menor o igual que 1.



$$|altura(I) - altura(D)| \leq 1$$

Es decir que si definimos

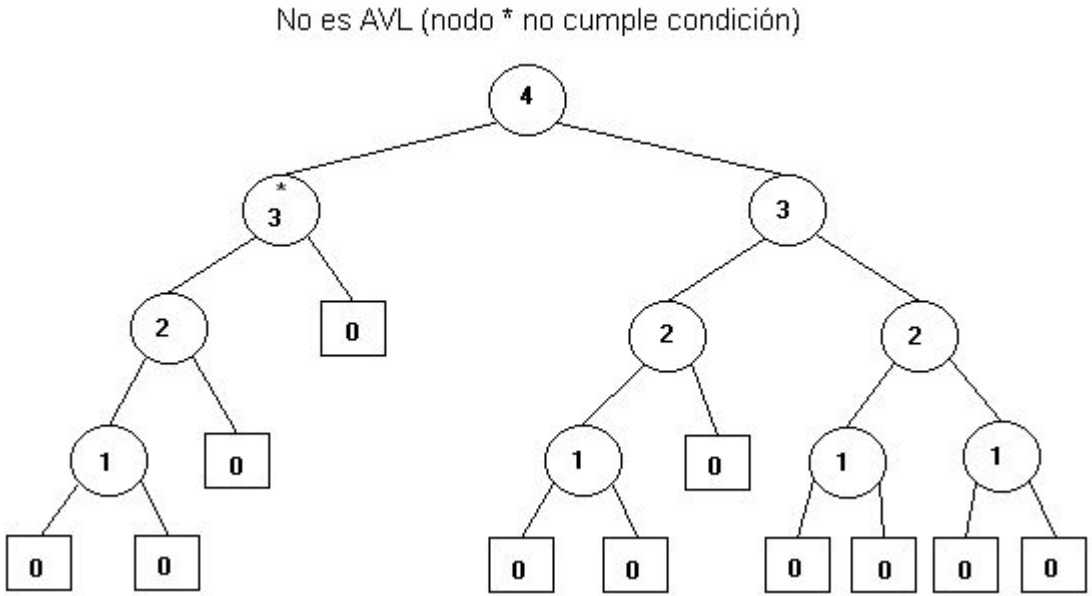
```
int btree_balance_factor(Btree N){
    return btree_altura(N->right)-btree_altura(N->left);
}
```

donde `btree_altura` nos devuelve la altura del árbol.

En el caso de que el argumento corresponda a un nodo en un Árbol AVL el valor de retorno va a estar en el conjunto `{-1, 0, 1}`.

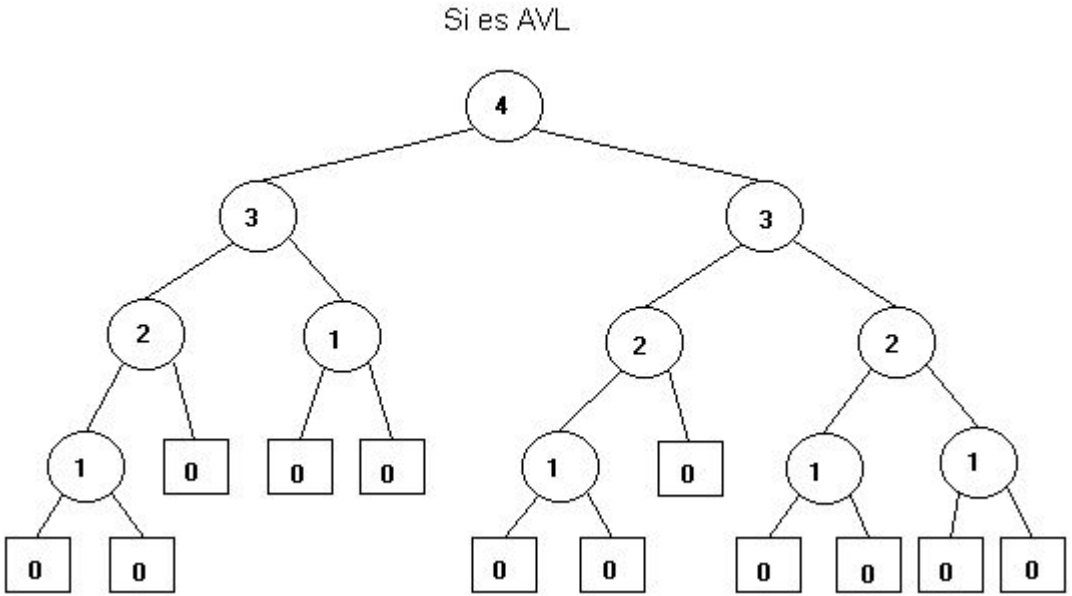
Árboles AVL

Por ejemplo: (el número dentro de cada nodo indica su altura)



Árboles AVL

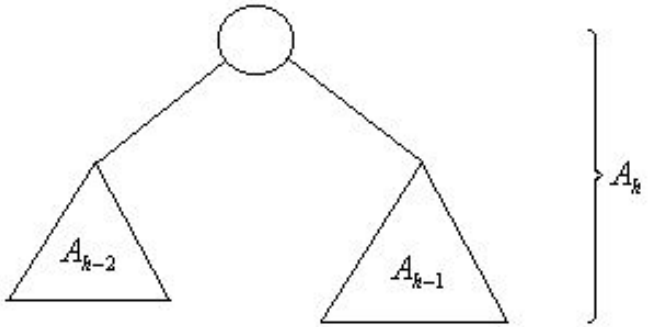
Por ejemplo: (el número dentro de cada nodo indica su altura)



Árboles AVL

Para una altura h dada, ¿cuál es la cantidad mínima de nodos que se necesitan para construir un Árbol AVL que alcance esa altura?

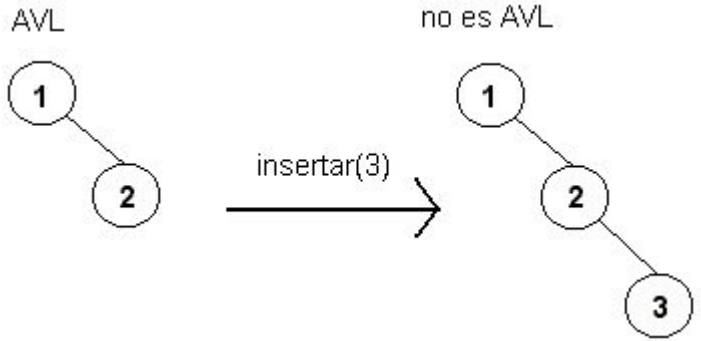
A partir de esta pregunta, y el hecho que en un Árbol AVL de estas características la diferencia de altura de sus hijos en todos los nodos tiene que ser 1 (para minimizar la cantidad de nodos), se puede demostrar que la altura de un Árbol AVL es del orden de $\log n$, donde n es la cantidad de nodos.



Inserción en un AVL

La inserción en un Árbol AVL se realiza de la misma forma que en un ABB, con la salvedad que hay que modificar la información de la altura de los nodos que se encuentran en el camino entre el nodo insertado y la raíz del árbol.

El problema potencial que se puede producir después de una inserción es que el árbol con el nuevo nodo no mantenga la propiedad:



Inserción en un AVL

En el ejemplo de la figura, la condición de balance se pierde al insertar el número 3 en el árbol, por lo que es necesario restaurar de alguna forma dicha condición. Esto siempre es posible de hacer a través de una modificación simple en el árbol, conocida como rotación.

Inserción en un AVL

Supongamos que después de la inserción de un elemento X el nodo desbalanceado más profundo en el árbol es N.

Esto quiere decir que la diferencia de altura entre los dos hijos de N tiene que ser 2, puesto que antes de la inserción el árbol estaba balanceado.

Inserción en un AVL

El problema pudo ser ocasionado al insertar el elemento en una de estas cuatro posibles opciones:

1. El elemento X fue insertado en el subárbol izquierdo del hijo izquierdo de N.
2. El elemento X fue insertado en el subárbol derecho del hijo izquierdo de N.
3. El elemento X fue insertado en el subárbol izquierdo del hijo derecho de N.
4. El elemento X fue insertado en el subárbol derecho del hijo derecho de N.

Dado que el primer y último caso son simétricos, así como el segundo y el tercero, sólo hay que preocuparse de dos casos principales: una inserción "hacia afuera" con respecto a N (primer y último caso) o una inserción "hacia adentro" con respecto a N (segundo y tercer caso).

Inserción en un AVL

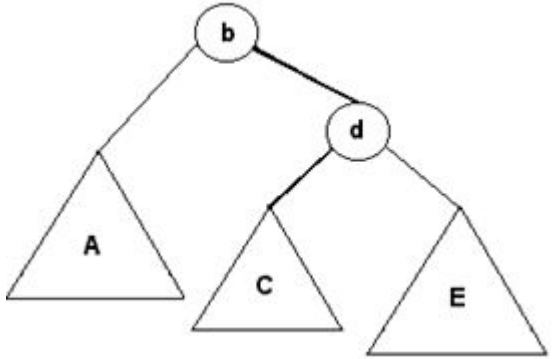
El desbalance por inserción "hacia afuera" con respecto a N se soluciona con una rotación simple que puede ser a izquierda o derecha dependiendo de dónde estaba el desbalance.

La inserción "hacia adentro" requiere de una rotación doble, es decir, dos rotaciones simples.

Comencemos viendo como serían las rotaciones simples y luego, las dobles.

Inserción en un AVL – Rotación simple

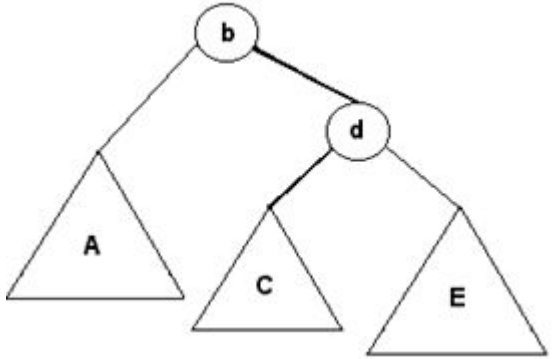
Supongamos que tenemos un Árbol AVL al que se le inserta un nodo de forma tal que deja de cumplir la condición. Es decir, tenemos este árbol, que representa nuestro caso 4.



El elemento X fue insertado en E y dejó de cumplir la condición, ya que hay 2 niveles de diferencia entre el subárbol izquierdo y el derecho de b.

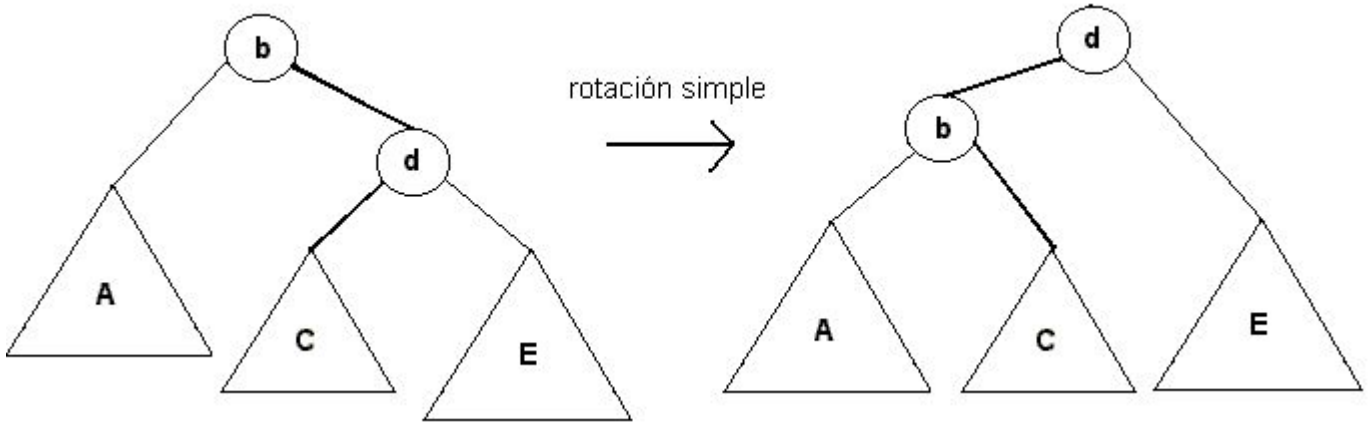
Inserción en un AVL – Rotación simple

Para recuperar la condición de balance se necesitaría bajar A en un nivel y subir E en un nivel. Esto lo vamos a lograr cambiando las referencias derecha de b e izquierda de d, quedando este último como nueva raíz del árbol.



Inserción en un AVL – Rotación simple

Es decir, nuestro árbol se transforma usando una rotación simple a izquierda. Podemos verlo así:



Inserción en un AVL – Rotación simple

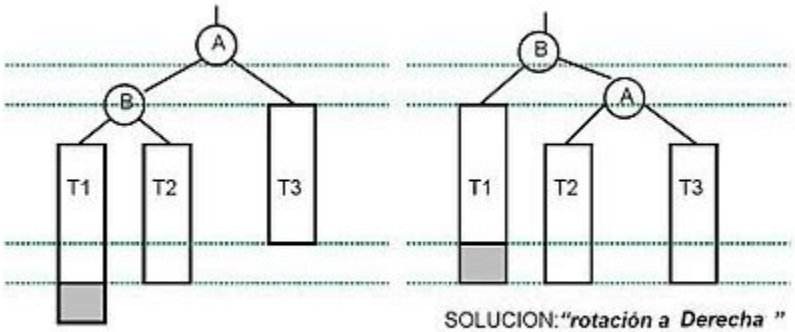
Resumiendo, supongamos que antes de la inserción, la altura de nuestro Árbol AVL era de C+1.

Con el movimiento que hicimos el nuevo árbol tiene la misma altura que antes de insertar el elemento, es decir, C+1. Esto implica que no puede haber nodos desbalanceados más arriba en el árbol, por lo que es necesaria una sola rotación simple para devolver la condición de balance al árbol.

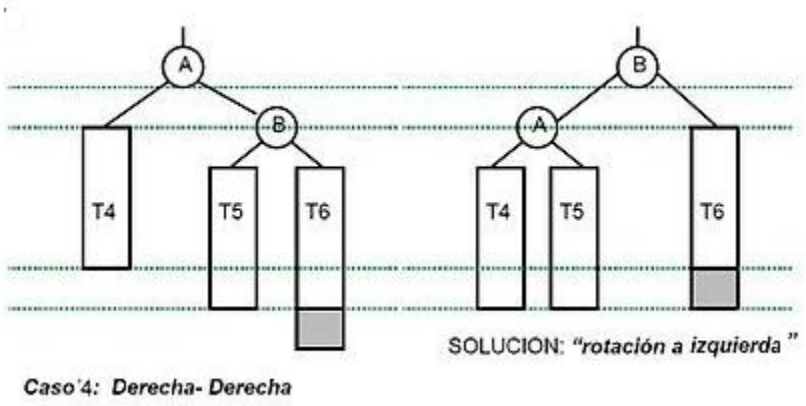
El árbol obtenido, de esta manera, es un Árbol AVL.

Inserción en un AVL – Rotación simple

Resumiendo, nuestras rotaciones simples a derecha e izquierda respectivamente serían:



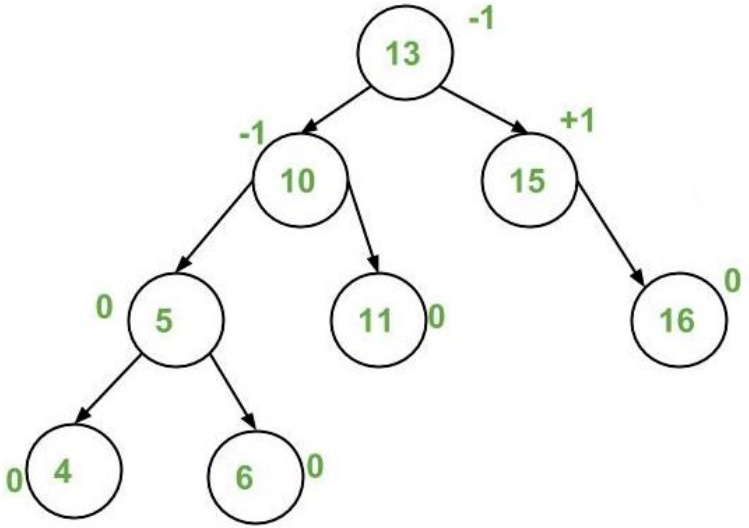
Caso 1: Izquierda-Izquierda



Caso 4: Derecha- Derecha

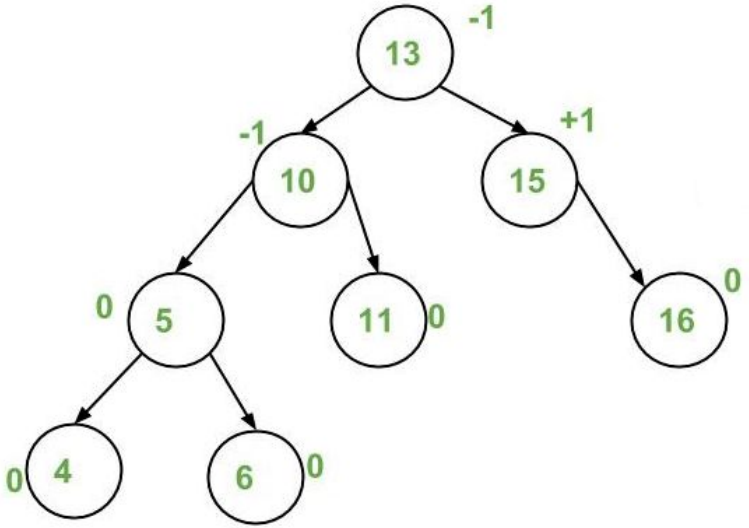
Inserción en un AVL – Rotación simple

Veamos un ejemplo práctico de esto. Supongamos que tenemos el siguiente Árbol AVL.



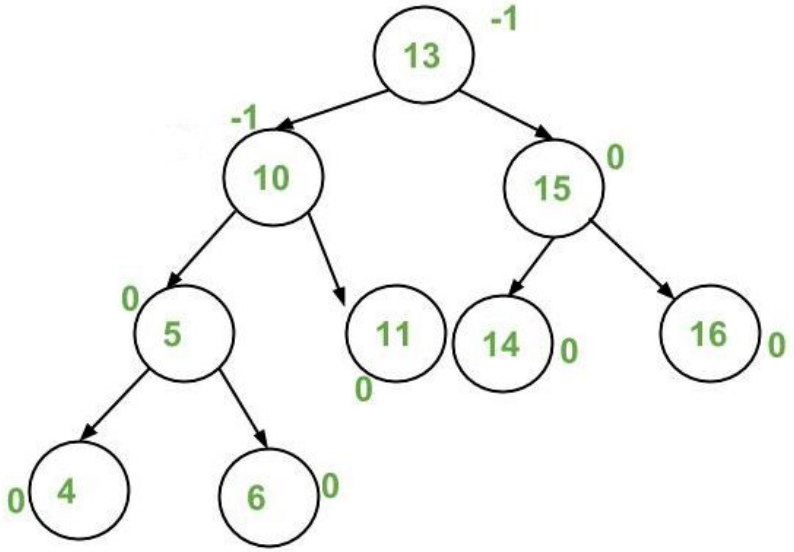
Inserción en un AVL – Rotación simple

En cada nodo se almacena un valor entero y, el número que aparece al costado es el resultado de la función `btree_balance_factor`.



Inserción en un AVL – Rotación simple

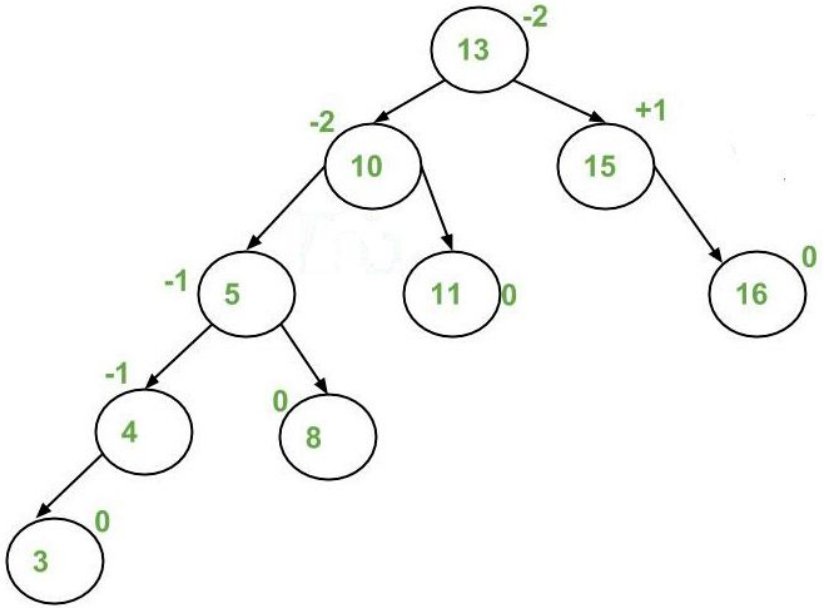
Supongamos que insertamos el valor 14 al árbol. Lo que obtenemos sería:



El cual sigue estando balanceado. O sea, no tenemos que aplicar ninguna rotación.

Inserción en un AVL – Rotación simple

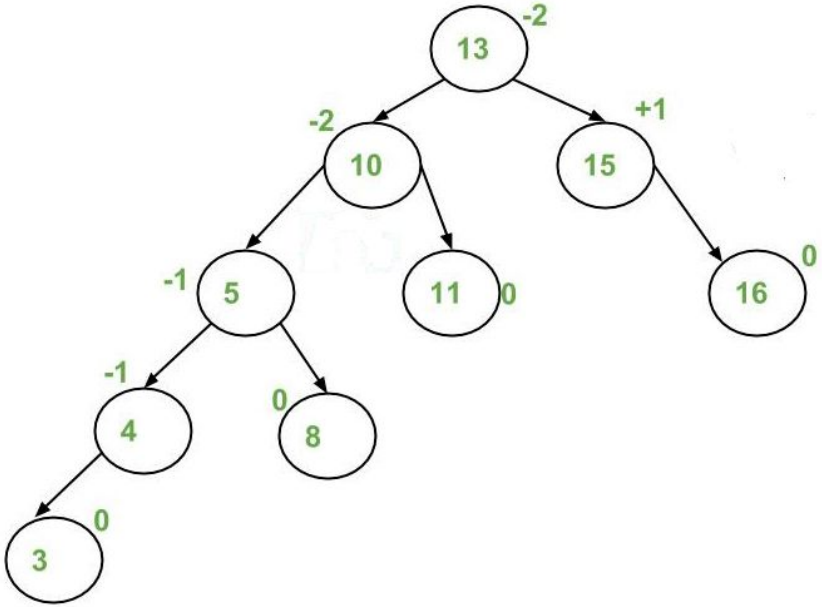
Pero, qué pasaría si, en lugar del 14 insertamos el valor 3. El árbol obtenido sería:



El cual NO está balanceado.

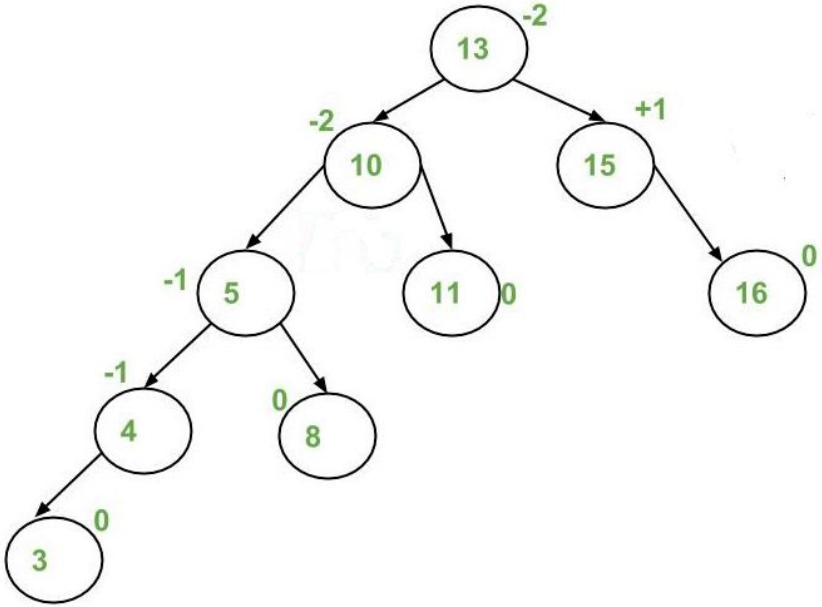
Inserción en un AVL – Rotación simple

Para poder balancearlo tenemos que identificar el nodo más profundo que está desbalanceado. Ese nodo es el que tiene el valor 10.



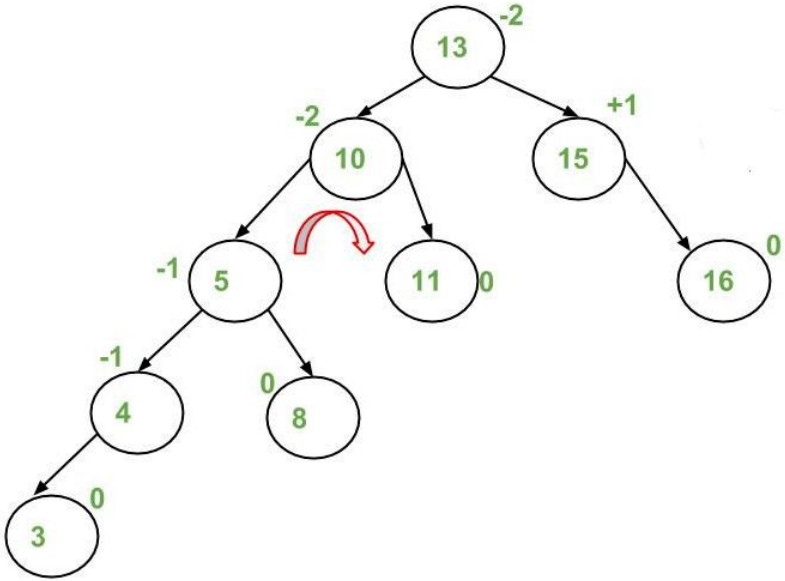
Inserción en un AVL – Rotación simple

Si lo analizamos podemos ver que estamos en el caso 1 (de los mencionados en el slide 11), ya que el desbalance está en el subárbol izquierdo del hijo izquierdo del nodo. O sea que nuestra solución debería ser una rotación a derecha.



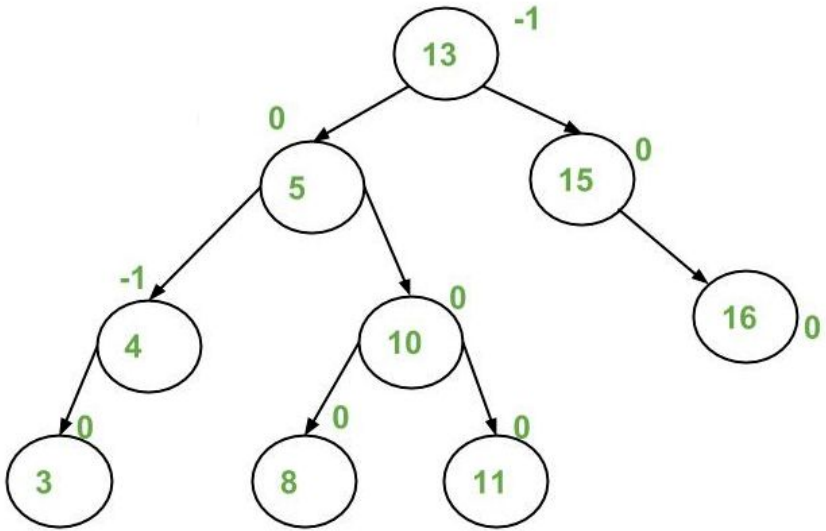
Inserción en un AVL – Rotación simple

Es decir, dado que hay dos niveles más a izquierda que a derecha, tenemos que hacer una rotación a derecha para poder subir el Nodo con el 5 y bajar el Nodo con el 10.



Inserción en un AVL – Rotación simple

Obteniéndose el siguiente árbol el cual es un Árbol AVL.



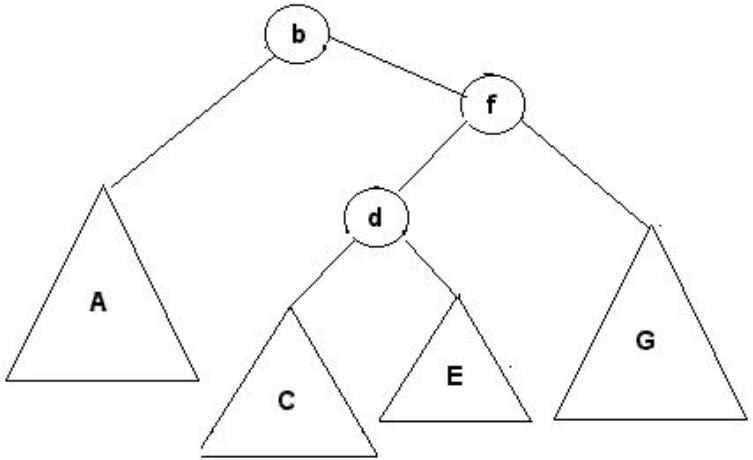
Inserción en un AVL – Rotación doble

Volviendo a nuestra motivación original, cuando insertamos nos quedan analizar estas dos situaciones:

2. El elemento X fue insertado en el subárbol derecho del hijo izquierdo de N.
3. El elemento X fue insertado en el subárbol izquierdo del hijo derecho de N.

Inserción en un AVL – Rotación doble

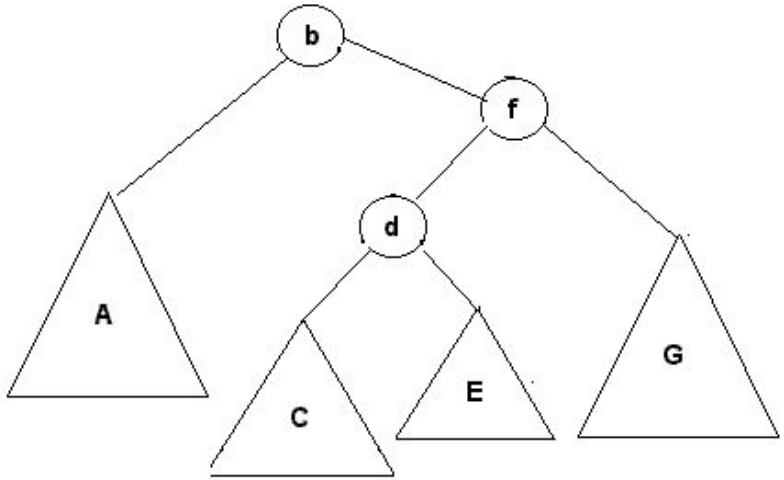
Veamos este ejemplo que corresponde al caso 3. Supongamos que nuestro elemento fue insertado en el subárbol C lo que produce el desbalance. Es decir, en el subárbol izquierdo del hijo derecho de b.



Inserción en un AVL – Rotación doble

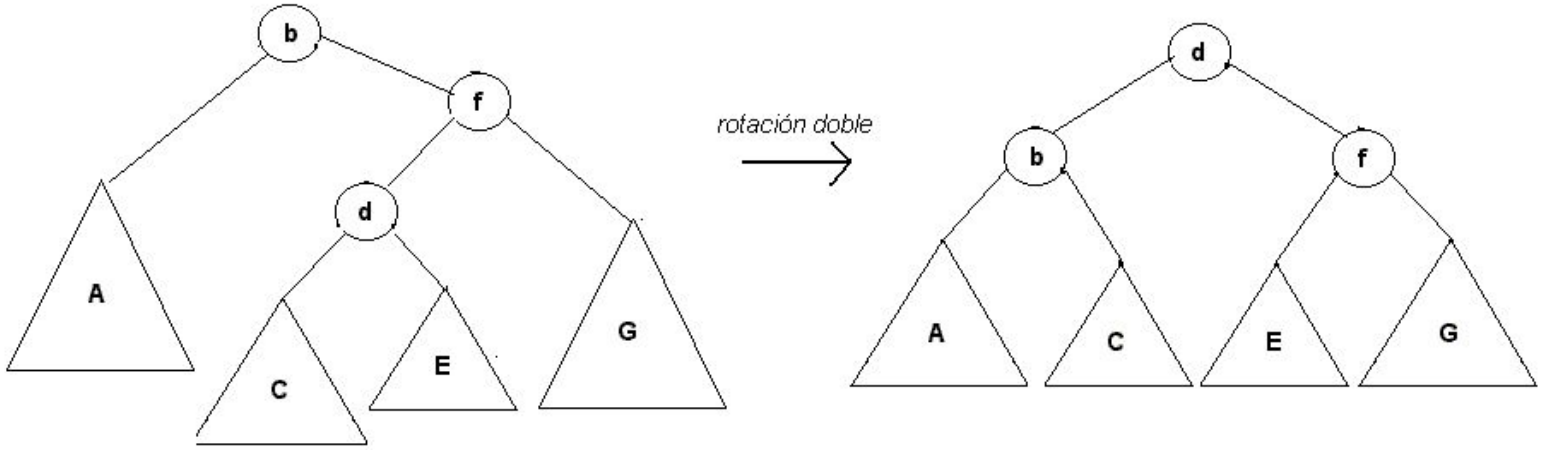
O sea, tenemos un desbalance producido por una inserción "hacia adentro" con respecto a b.

Para recuperar el balance del árbol es necesario subir C y bajar A pero, como están en diferentes ramas tenemos primero que subir a d para luego poder bajar a b.



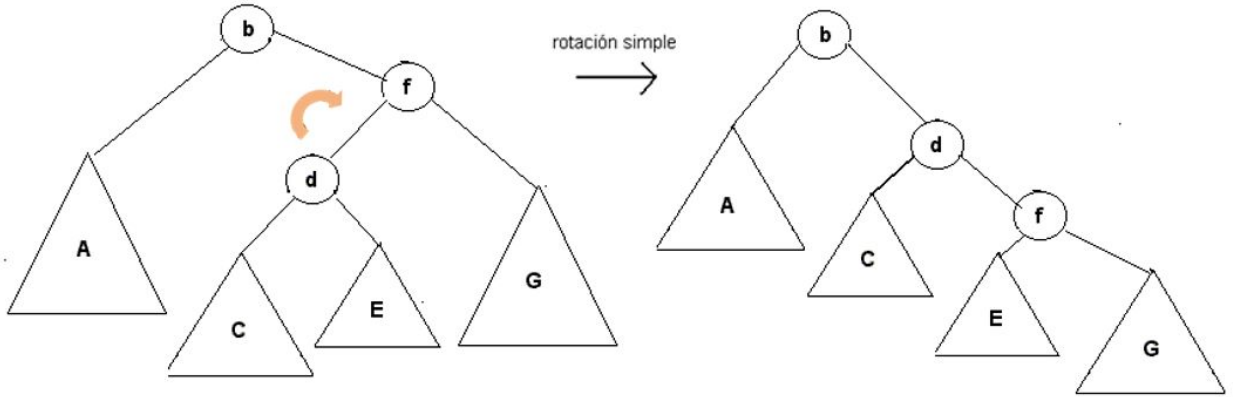
Inserción en un AVL – Rotación doble

Por este motivo necesitamos aplicar dos rotaciones simples: la primera entre d y f, y la segunda entre d, ya rotado, y b, obteniéndose el resultado de la figura.
Vamos a ver el paso a paso de las rotaciones para transformar nuestro árbol en un Árbol AVL.



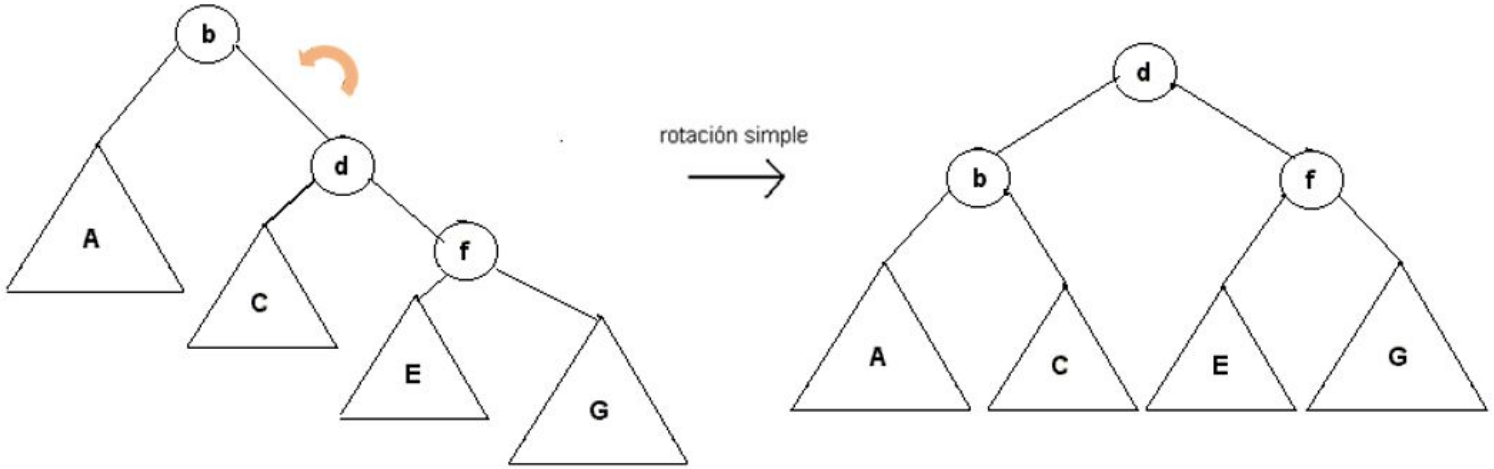
Inserción en un AVL – Rotación doble

Paso 1: aplicamos una rotación simple a derecha para llegar a un estado intermedio y subir C.



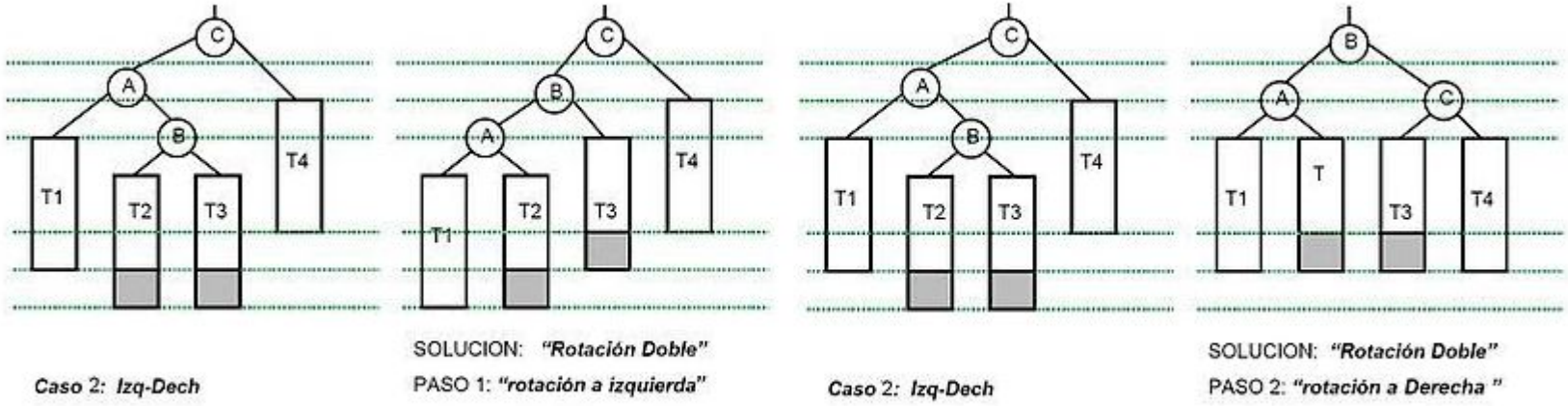
Inserción en un AVL – Rotación doble

Paso 2: volvemos a aplicar una rotación simple pero a izquierda para bajar A y finalmente alcanzar nuestro objetivo: tener un Árbol AVL.



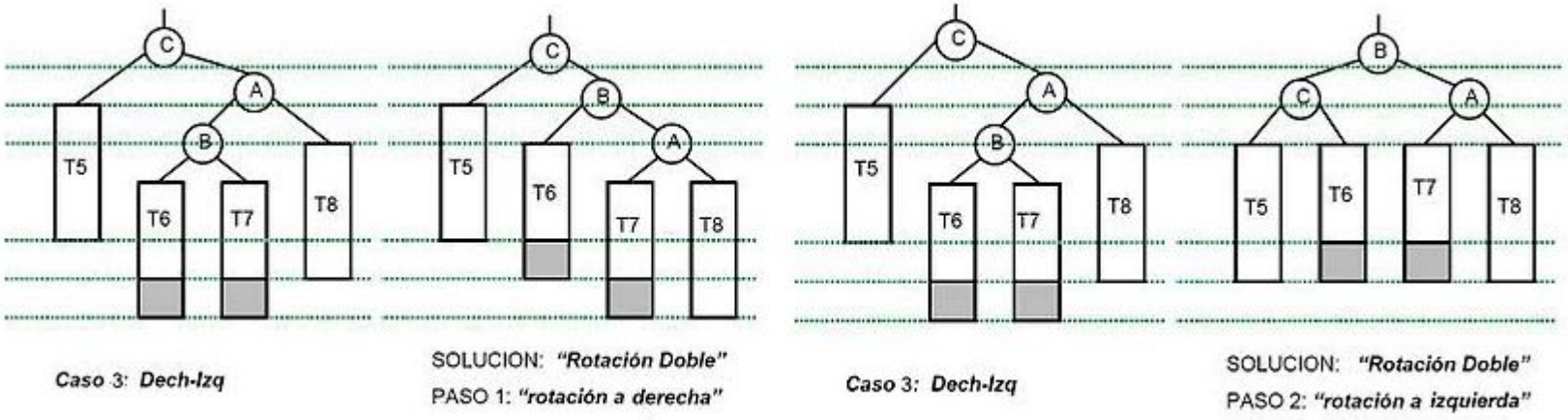
Inserción en un AVL – Rotación doble

Acá tenemos ejemplificados los casos



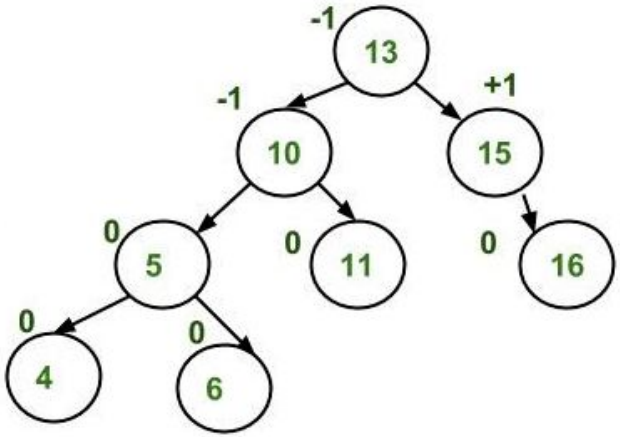
Inserción en un AVL – Rotación doble

También se puede hacer primero una rotación a izquierda y, luego una a derecha:



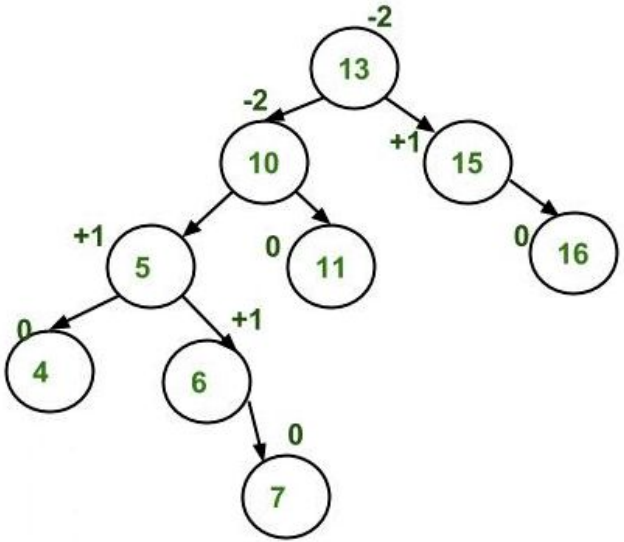
Inserción en un AVL – Rotación doble

Veamos un ejemplo práctico del uso de rotaciones dobles. Supongamos que tenemos el siguiente Árbol AVL.



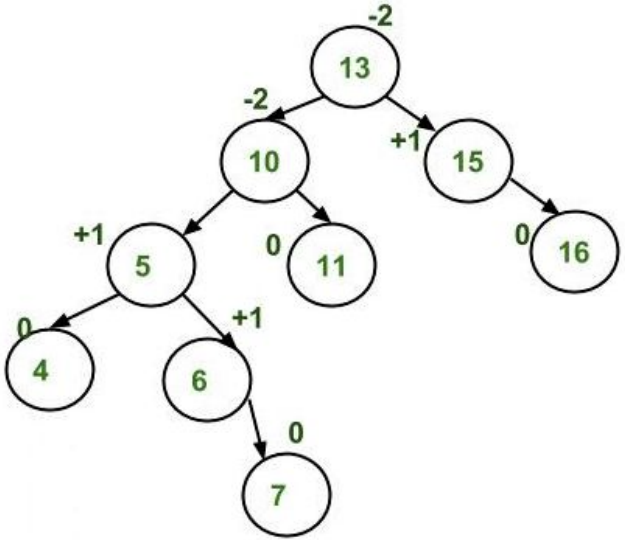
Inserción en un AVL – Rotación doble

Si insertamos el valor 7 obtenemos el siguiente árbol, el cuál está desbalanceado.



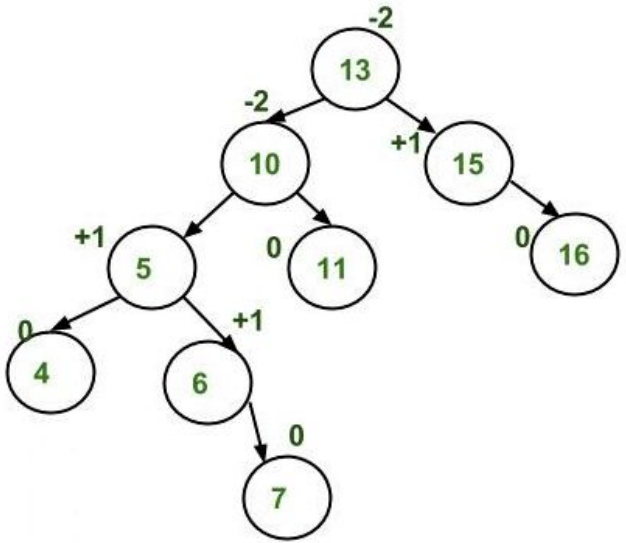
Inserción en un AVL – Rotación doble

El nodo más profundo desbalanceado es el que tiene el valor 10. Vemos que estamos en el caso 2 (slide 11), el nodo fue insertado en el subárbol derecho del hijo izquierdo del nodo 10.



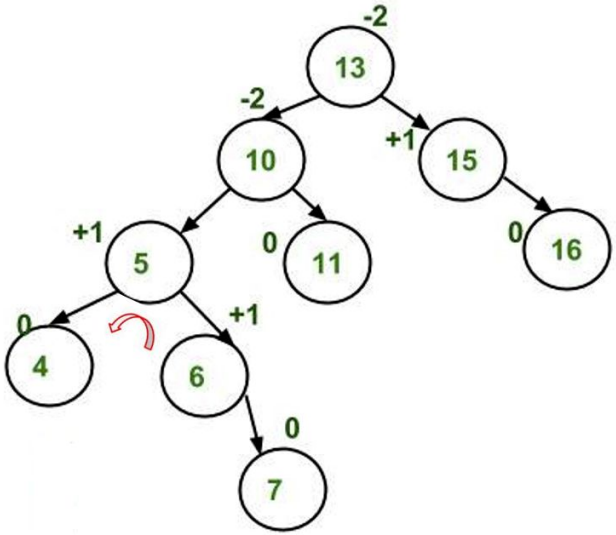
Inserción en un AVL – Rotación doble

Para resolverlo tenemos entonces que hacer una rotación doble.
En este caso sería, primero a izquierda para subir el nodo con el 7 (esto se realiza sobre el nodo 5) y, luego a derecha sobre el nodo 10.



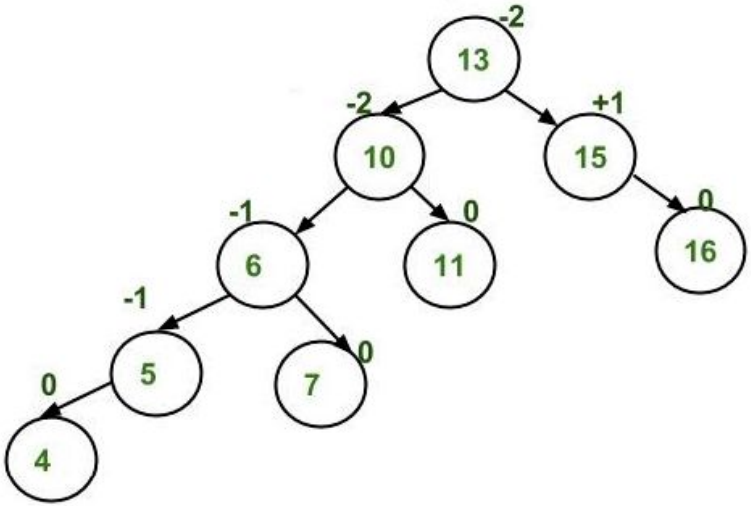
Inserción en un AVL – Rotación doble

Comencemos con la primera rotación simple a izquierda.



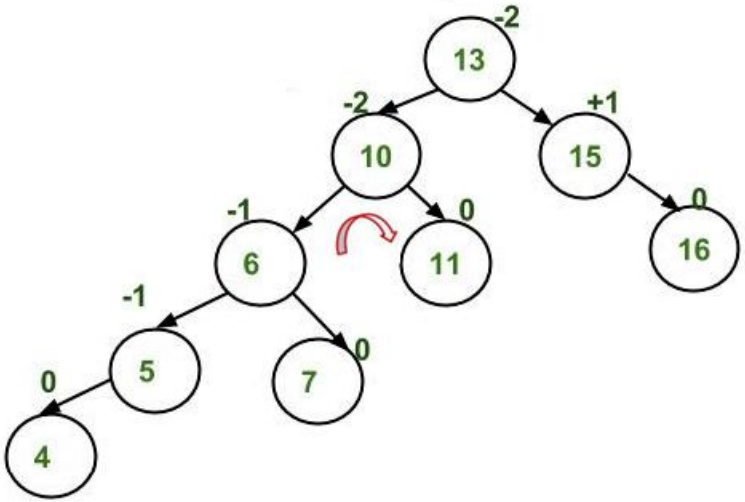
Inserción en un AVL – Rotación doble

El resultado de esta rotación a izquierda sería:



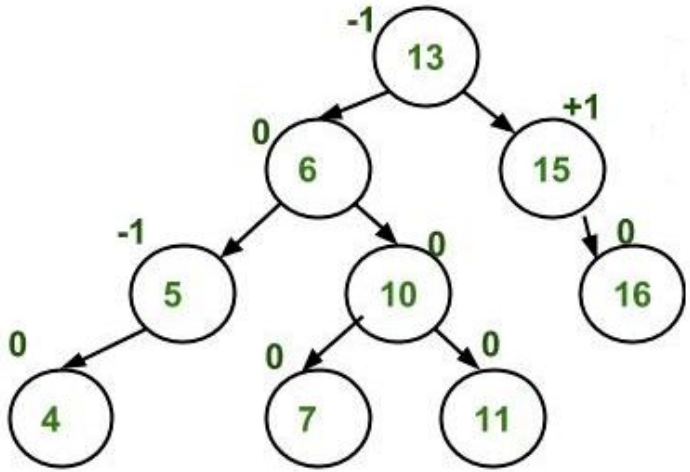
Inserción en un AVL – Rotación doble

Ahora, tenemos que hacer la rotación a derecha sobre el 10 ya que el desbalance está en el subárbol izquierdo del hijo izquierdo del nodo.



Inserción en un AVL – Rotación doble

Finalmente, luego de esta rotación, obtenemos el siguiente árbol que sí es un Árbol AVL.



Eliminación en un AVL

La eliminación en árbol AVL se realiza de manera análoga a un ABB, pero también es necesario verificar que la condición de balance se mantenga una vez eliminado el elemento.

En caso que dicha condición se pierda, será necesario realizar una rotación simple o doble dependiendo del caso, pero es posible que se requiera más de una rotación para reestablecer el balance del árbol.

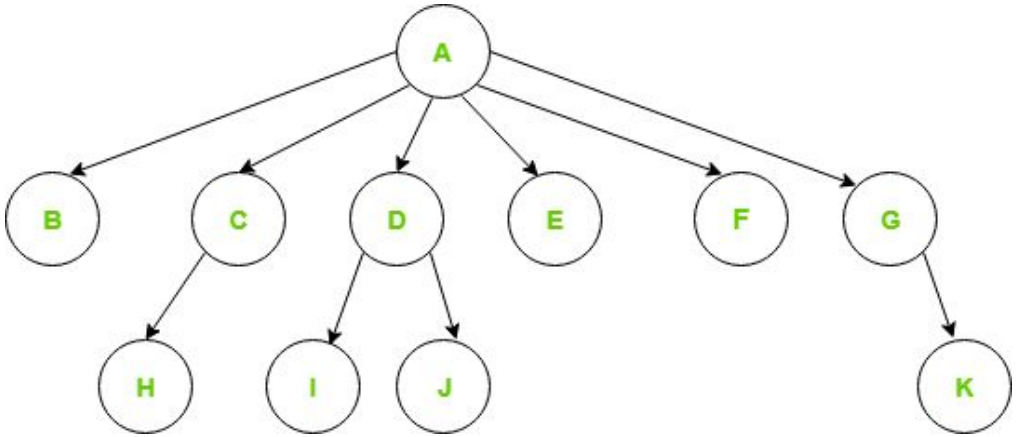
Estructuras de Datos
Árboles

Árboles Generales

En un Árbol general cada nodo puede poseer un número indeterminado de hijos. Esto nos genera ciertas complicaciones a la hora de pensar en su implementación.
Vamos a ver dos enfoques para resolver este problema.

Árboles Generales

Nuestro primer enfoque tiene como idea el hecho de que al no saber cuántos hijos tiene cada nodo, se utilice una estructura dinámica para almacenarlos.
Por ejemplo, podríamos pensar en una lista de hijos.



Árboles Generales

Es decir, tendríamos una implementación de este tipo. Con un dato genérico y usando nuestra declaración de listas genéricas, de la siguiente forma:

```
typedef struct _GTNodo {
  void* dato;
  struct GList childs;
  } GTNodo;

typedef GTNodo *GTree;
```

De esta manera, podemos implementar un Árbol general, que además, es genérico.

Árboles Generales

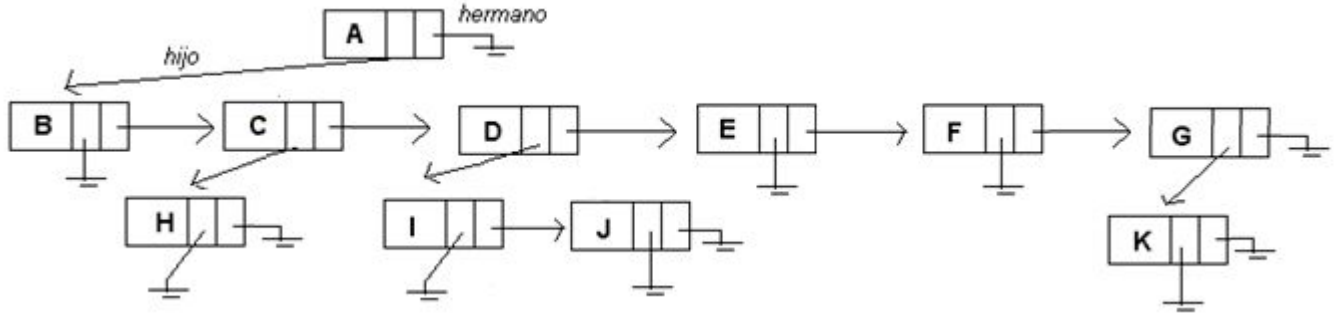
Nuestro segundo enfoque plantea la siguiente implementación de los nodos.

Como no se sabe de antemano cuántos hijos tiene un nodo en particular se utilizan dos referencias, una a su primer hijo y otra a su hermano más cercano.

La raíz del árbol necesariamente tiene la referencia a su hermano como *null*.

Árboles Generales

Es decir, tendríamos una implementación que generaría un árbol de esta manera, teniendo en cuenta el ejemplo del slide 3.



Árboles Generales

Notemos que, con este enfoque, todo árbol general puede representarse como un árbol binario donde el puntero a derecha apunta al hermano y, el puntero a izquierda apunta a su primer hijo.
En esta implementación, la raíz no tendría hermanos, por lo que sería siempre *null*.

Si se permite que la raíz del árbol tenga hermanos, lo que se conoce como *bosque*, entonces tendríamos que el conjunto de los bosques generales es isomorfo al conjunto de los árboles binarios.

En efecto, las propiedades vistas en los árboles binarios se siguen cumpliendo en los árboles generales.