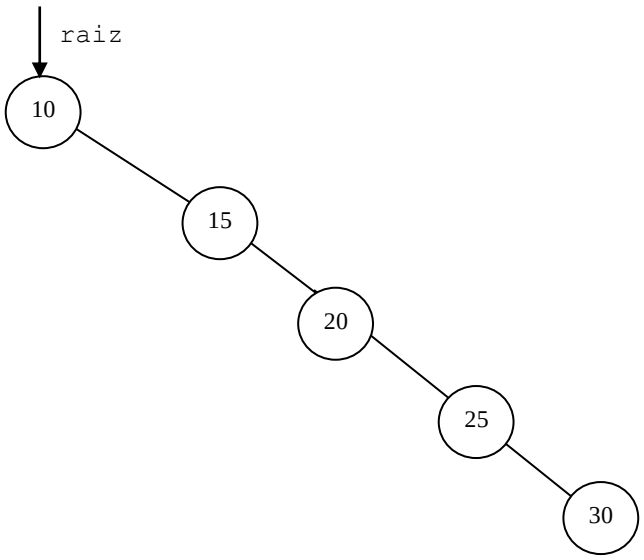
Ficha 13

Árboles AVL

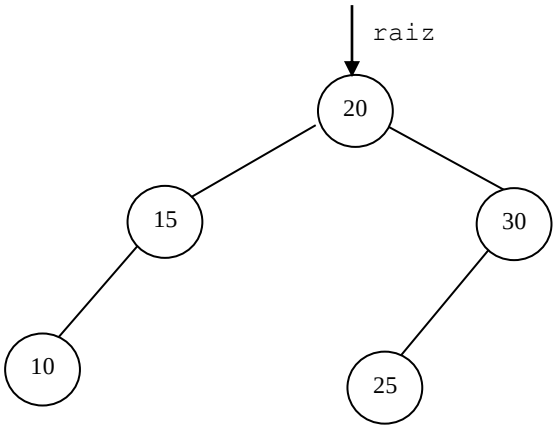
1.] Arboles de búsqueda equilibrados: Equilibrio AVL.

El algoritmo de inserción visto para árboles de búsqueda es fuertemente dependiente de la secuencia en la que vienen las claves a ser insertadas. El mismo algoritmo puede producir árboles cuyas formas sean completamente diferentes, de acuerdo a cómo vengan las claves. Por ejemplo, si entregamos las claves en esta secuencia: {10, 15, 20, 25, 30} el algoritmo de inserción generará el siguiente árbol, cuya altura es h = 5, igual al número de nodos del árbol:



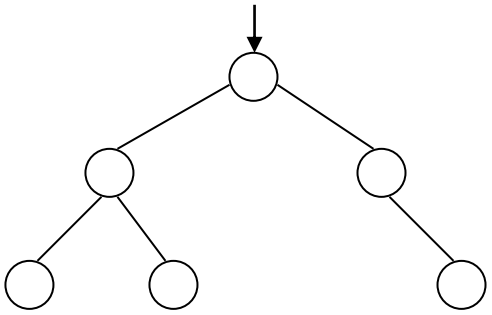
El árbol es de búsqueda, pero está organizado de forma tan desbalanceada que ha degradado en una lista simple... Pero el caso es que si pretendemos buscar una clave en este árbol, nuestro tiempo de búsqueda será O(n) y ya no O(log(n)). ¿Cuál es el problema con este árbol? ¿Qué entendemos por *desbalance*? Intuitivamente, podemos ver que el árbol anterior tiene altura mucho mayor a la que podría tener si organizáramos los nodos de otra forma. Las mismas claves podrían producir este otro árbol (de altura h = 3) si fueran ingresadas en esta otra secuencia: {20, 15, 10, 30, 25} (ver figura en página siguiente).

En este árbol, ubicar una clave llevará como máximo tres comparaciones, mientras que en el anterior tendremos cinco. Como puede verse, la altura del árbol determina la cantidad máxima de comparaciones que tendremos que realizar. Si esa altura fuera la mínima posible, el orden de búsqueda sería O(log(n)), pues las claves estarían distribuidas aproximadamente por mitades entre cada subárbol. Pero si permitimos que la altura del árbol crezca de manera indiscriminada, perderemos el orden de búsqueda logarítmico y nos aproximaremos al orden lineal típico de las listas.

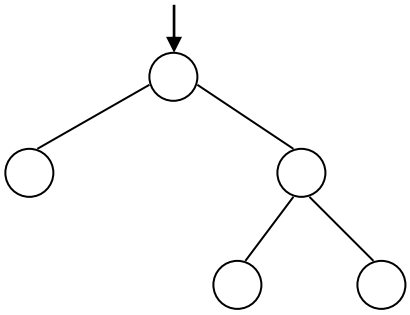


Es claro, por lo tanto, que el proceso de localización de una clave en un árbol de búsqueda se ve favorecido si la altura del árbol es mínima (esto se debe a que en la búsqueda se realiza una y solo una comparación en cada nivel del árbol, entonces mientras menos niveles tenga el árbol, más rápida será la búsqueda). En ese sentido, lo mejor que podría ocurrir es que un árbol de búsqueda tenga *equilibrio perfecto*.

Decimos que un árbol binario tiene *equilibrio perfecto*, si para cada uno de sus nodos se cumple que la diferencia entre la cantidad de nodos de sus subárboles izquierdo y derecho es a lo sumo igual a uno. Esto significa, intuitivamente, que para construir el árbol debe preverse que cada nivel se vaya llenando en forma completa antes de pasar al siguiente nivel y por lo tanto, el árbol tendrá la mínima altura posible para su número de nodos:



a.) Arbol en equilibrio perfecto, h = 3

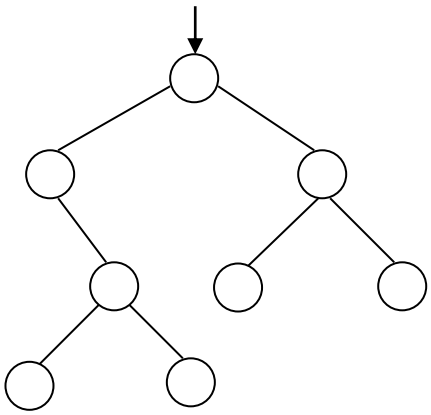


b.) Arbol SIN equilibrio perfecto, h = 3

Notar que un árbol puede tener altura mínima, pero no cumplir la regla de la distribución pareja de nodos y por lo tanto no tener equilibrio perfecto (ver figura b). Por otra parte, la regla debe cumplirse para cada nodo del árbol (y no sólo para el nodo raiz), por lo cual árbol de la página siguiente tampoco tiene equilibrio perfecto (el nodo raiz cumple la regla, pues tiene cuatro nodos a la izquierda y tres a la derecha, pero el hijo izquierdo de la raiz está desequilibrado: no tiene nodos a la izquierda, contra tres nodos que tiene a la derecha: la diferencia es mayor a uno).

Un algoritmo de inserción en un árbol de búsqueda que restaure siempre el equilibrio perfecto rara vez será eficiente, ya que después de hacer una inserción la restauración del equilibrio es una operación muy compleja y el tiempo que se ganaría buscando velozmente, se perdería re-equilibrando el árbol luego de cada inserción.
Sin embargo, el algoritmo de inserción que vimos para la clase TreeSearch no puede dejarse como está, pues puede producir árboles degradados en listas con mucha facilidad. Si luego de una inserción no es eficiente rebalancear el árbol para dejarlo en equilibrio perfecto, debemos pensar

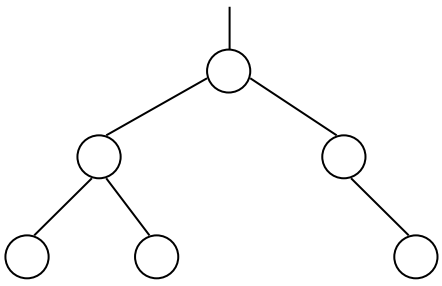
en alguna alternativa que cueste menos en términos de operaciones de rebalanceo, aunque eso pueda significar algún deterioro en el tiempo de búsqueda.



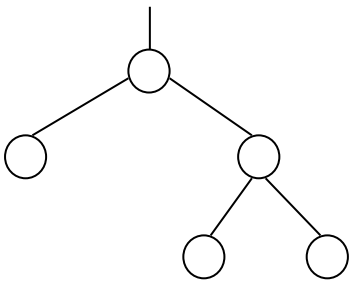
c.) Arbol SIN equilibrio perfecto: el nodo raiz cumple la regla, pero no su hijo izquierdo...

Una mejora sería introducir una definición más amplia del concepto de equilibrio, que sea más sencilla de cumplir. Una definición de equilibrio, en el sentido amplio o imperfecto citado arriba, fue postulada por los matemáticos rusos G. M. Adelson-Velskii y E. M. Landis en 1965. El criterio de equilibrio definido se denomina hoy equilibrio AVL, tomando las iniciales de los apellidos de sus descubridores. La definición es la siguiente:

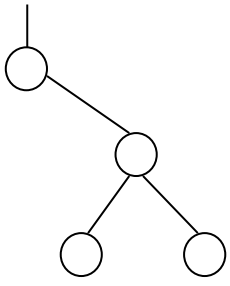
*Un árbol binario está en **equilibrio AVL**, si y sólo si para cada uno de sus nodos, ocurre que las **alturas** de sus dos subárboles difieren cuando mucho en uno.*



Ejemplo c.)
Arbol en equilibrio perfecto, que también es AVL.



Ejemplo d.)
Arbol SIN equilibrio perfecto, pero que sin embargo es AVL.



Ejemplo e.)
Sin equilibrio alguno.

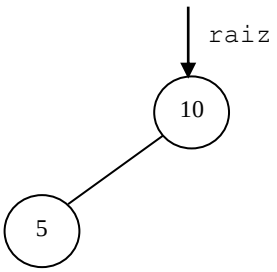
Notar que todo árbol en equilibrio perfecto es también AVL (por ejemplo, el árbol de la figura c está en equilibrio perfecto y también es AVL). El árbol de la figura e no es AVL porque para el nodo raíz no se cumple la regla: el subárbol izquierdo tiene altura 0, y el derecho altura 2: la diferencia es 2, y la regla no se cumple.

Observar además que el árbol de la figura d no tiene equilibrio perfecto, aún cuando su altura es mínima. Un algoritmo que busque siempre el equilibrio perfecto, rechazaría ese arbol y obligaría a realizar operaciones de rebalanceo, siendo ello innecesario pues el árbol tiene altura mínima. Sin embargo, un algoritmo que busque mantener el equilibrio AVL aceptaría ese árbol como válido, sin hacer ninguna operación de re-equilibrado. Por lo que se ve, mantener un árbol binario de

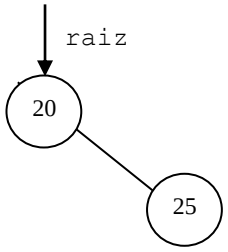
búsqueda en equilibrio AVL es relativamente más simple y por añadidura, los mismos Adelson-Velskii y Landis demostraron que un árbol de búsqueda AVL nunca será más de un 45% más alto que su correspondiente perfectamente equilibrado. Esto implica que una operación de búsqueda, inserción o borrado de una clave, se hará con razonable eficiencia.

2.] Inserción y borrado en árboles AVL.

Si analizamos con cuidado el proceso de insertar claves en un árbol de búsqueda, veremos que hay dos situaciones que provocan desequilibrio y que deben considerarse en detalle si se desea modificar el algoritmo para que mantenga el equillibrio; y otras dos que son simétricas a estas. Supongamos los siguientes árboles simples:



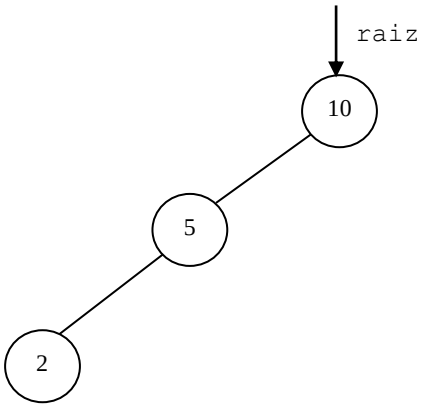
Ejemplo f.)



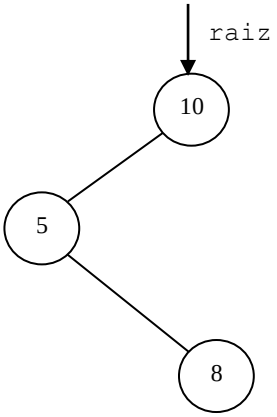
Ejemplo g.)

Si en el árbol del ejemplo f se inserta una clave mayor a 10, no habrá problema alguno ya que el árbol incluso ganará equilibrio. Pero si en ese árbol se inserta una clave menor a 10 (por ejemplo un 2 o un 8), la misma provocará que el nodo que contiene al 10 pierda el equilibrio (ver figuras en página siguiente).

En ambos casos, el nodo 10 ha perdido el equilibrio AVL, pues la altura de su subárbol izquierdo es dos, mientras que la del subárbol derecho es 0: la diferencia de alturas es mayor a uno. Si pretendemos que el algoritmo de inserción se recupere de estos desequilibrios, entonces debemos hacer que luego de insertar la clave sea capaz de verificar si algún nodo ha perdido el equilibrio, y en ese caso, realizar las operaciones de re-equilibrado que sean necesarias. Esas operaciones de re-equilibrado se suelen designar como rotaciones, y para cada tipo de desequilibrio se aplica una rotación particular.

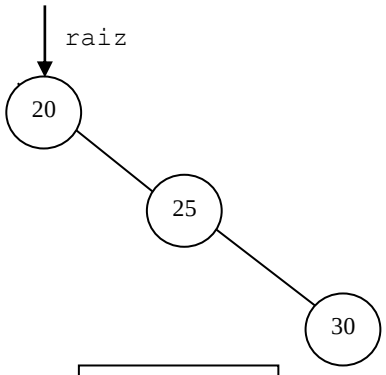


h.) después de insertar el 2

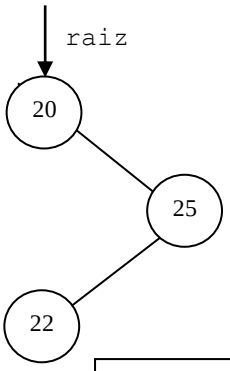


i.) después de insertar el 8

El hecho es que los ejemplos *h* e *i* anteriores muestran dos de los cuatro tipos de desequilibrio que pueden darse al insertar una clave. Los otros dos, son exactamente los mismos pero reflejados hacia la derecha (ver ejemplos j y k), por lo cual, realmente basta con estudiar la solución para los dos casos originales, y las otras dos salen por simetría.

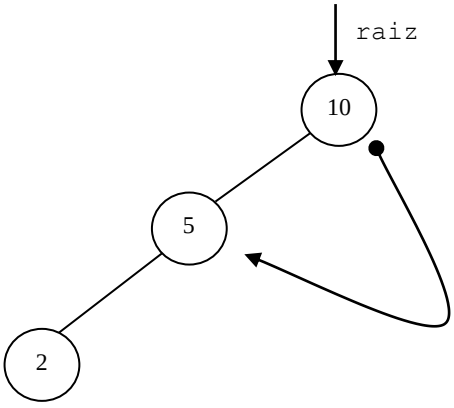


j.) después de insertar el 30

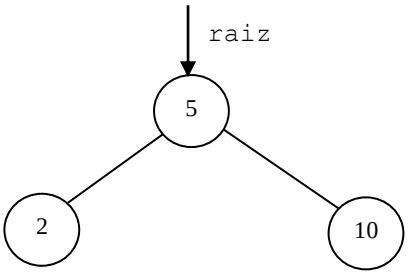


k.) después de insertar el 22

Las rotaciones más sencillas se dan para los casos de desequilibrio en diagonal (ejemplos h y j) Si observamos el ejemplo h, para restaurar el equilibrio en el árbol sólo debemos rotar el nodo 10 de forma que quede como hijo derecho del 5. A esta rotación se la suele designar como rotación derecha simple:

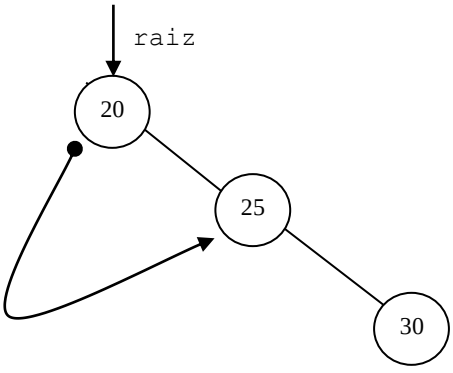


Antes de la rotación derecha simple

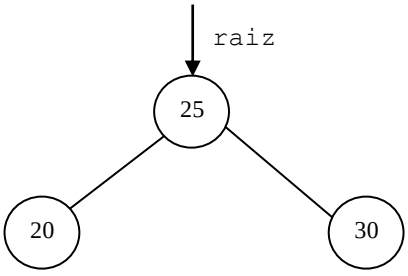


Después de la rotación derecha simple.

Del mismo modo, en el ejemplo j, sólo debemos rotar el 20 para que quede como hijo izquierdo del 25. Esta rotación se suele designar como rotación izquierda simple:

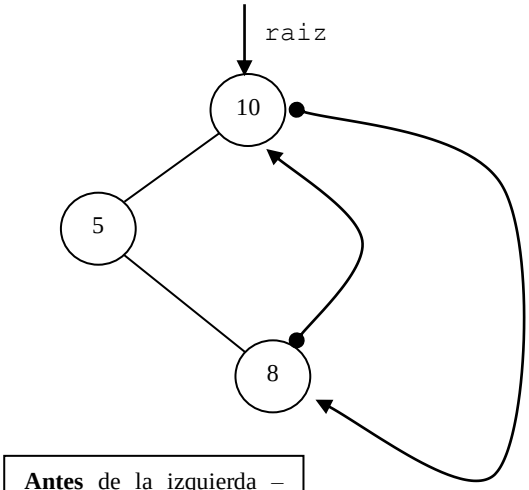


Antes de la rotación izquierda simple

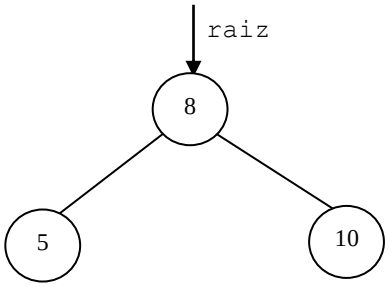


Después de la rotación izquierda simple

Un análisis semejante puede hacerse para los casos de desequilibrio en ángulo (ver ejemplos i y k). El caso del ejemplo i puede resolverse rotando el 8 hacia arriba y a la izquierda para que quede como padre del 5, y luego rotar el 10 hacia abajo y a la derecha para que quede como hijo derecho del 8. Esta rotación se suele designar como rotación izquierda – derecha doble:



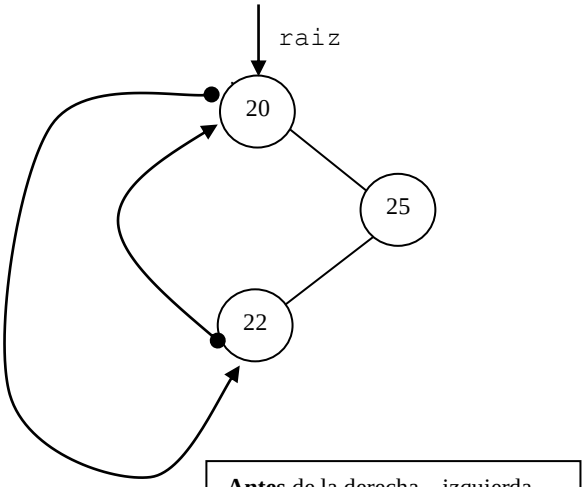
Antes de la izquierda – derecha doble.



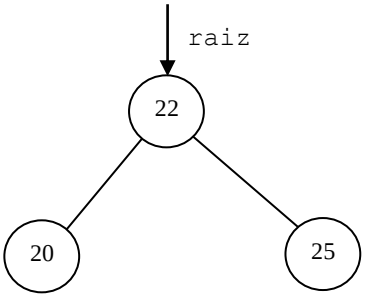
Después de la izquierda – derecha doble

Finalmente, el caso del ejemplo k se resuelve de forma similar pero simétrica (hacia el otro lado), con una rotación derecha –izquierda doble (ver figura en página siguiente).

El desarrollo del algoritmo de inserción y equilibrado depende fuertemente de la forma como se determine o almacene en el árbol la información referida al equilibrio de cada nodo. Una forma de trabajar sería que el algoritmo calcule el estado de equilibrio cada vez que haga una inserción, pero eso lleva a una gran pérdida de tiempo (habría que calcular las alturas de todos los subárboles mientras se recorre el árbol).



Antes de la derecha – izquierda doble.

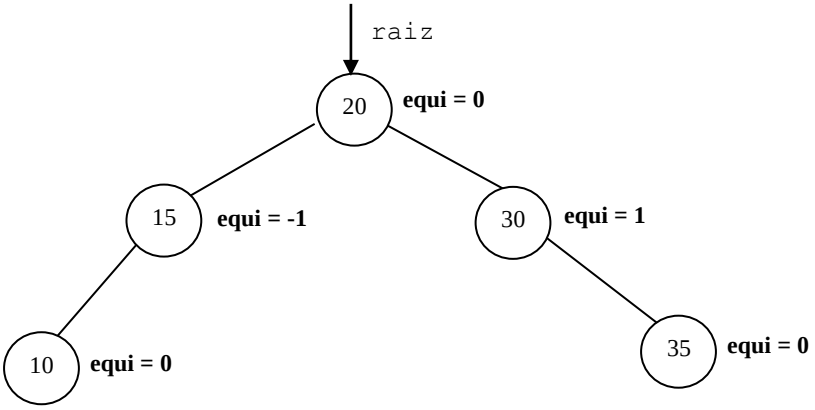


Después de la derecha – izquierda doble

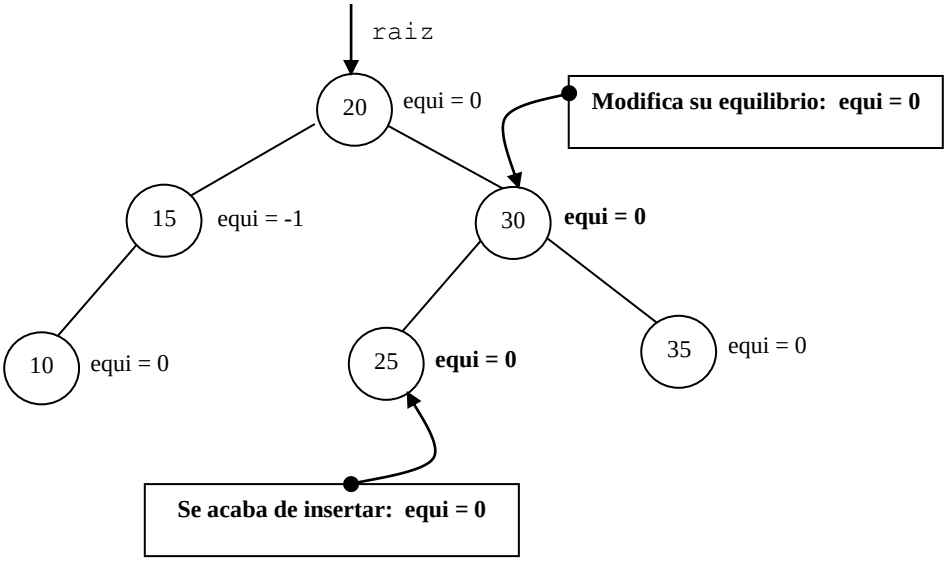
Una solución más aceptable (y por cierto, más tradicional) consiste en guardar en cada nodo el factor de equilibrio (en un atributo *equi*) del mismo, entendiendo como tal a la diferencia entre la altura del subárbol derecho (*hd*) menos la del izquierdo (*hi*), con lo cual:

```
equi  =  hd - hi
```

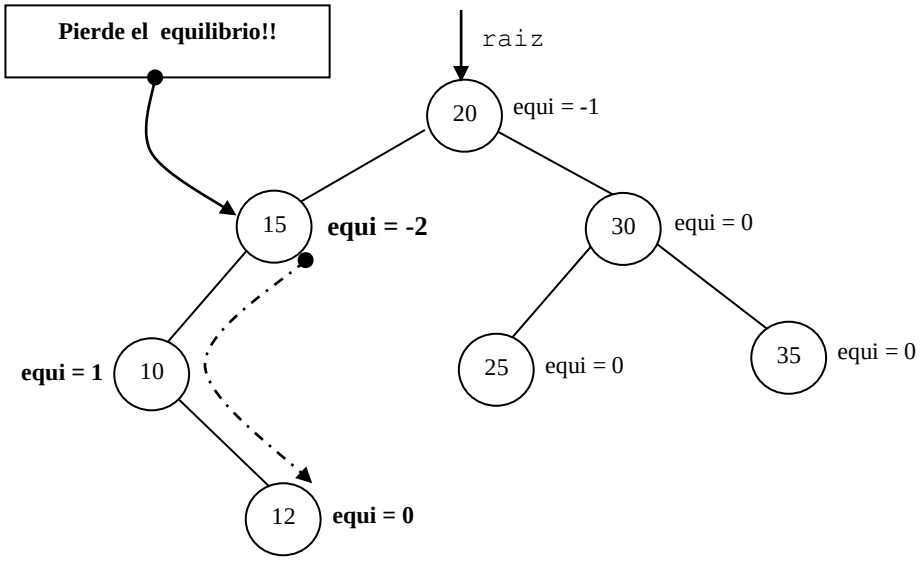
y es evidente que en un árbol AVL, el valor de *equi* para cada nodo será 1, 0, ó -1. Veamos esto en un ejemplo: en el siguiente árbol AVL, se muestra el factor equi de cada nodo:



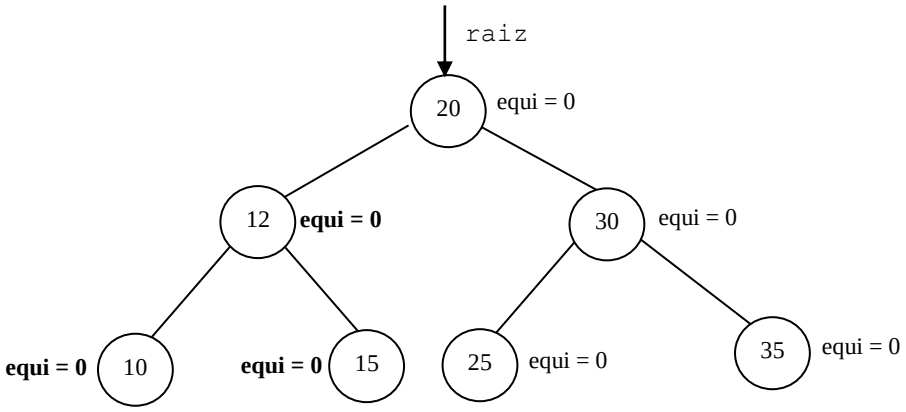
La idea es que al insertar un nodo, su factor equi será igual a cero (ya que el nodo se inserta como nodo hoja), pero su inserción puede cambiar los factores equi en los nodos anteriores en el camino de búsqueda de la clave insertada. Por ejemplo, si en el árbol anterior se inserta el valor 25 (que va a la izquierda del 30), quedaría así:



Como se ve en este caso, luego de insertar 25 todos los nodos del árbol mantienen sus valores de equilibrio entre -1 y 1, con lo cual ningún nodo se desbalancea (de hecho, el nodo 30 incluso ha mejorado su estado de equilibrio). Pero, ¿qué ocurre si se inserta un valor 12 (que iría a la derecha del 10)? En este caso, ocurriría un desequilibrio en ángulo en el nodo 15, y haría falta una rotación doble sobre el mismo:



Si los factores de equilibrio estuvieran guardados en cada nodo, entonces este desequilibrio era predecible al momento de insertar el 12, porque el factor equi del 15 tenía un valor -1, lo cual quería decir que la altura de su árbol izquierdo era superior en 1 a la del derecho. Y si la altura del izquierdo crece, el 15 se desequilibra. Notemos que podría estar desequilibrado algún nodo más hacia arriba del 15, pero esto sería consecuencia del desequilibrio del 15. Si este último se reequilibra con una rotación, sus antecesores se reequilibrarían a su vez. Esto quiere decir que al insertar una clave, nunca será necesario efectuar más de una rotación (simple o doble) para restaurar el equilibrio del árbol completo. En el árbol anterior, una rotación izquierda – derecha doble restaura el equilibrio, quedando así:



En suma: para poder desarrollar el algoritmo de inserción, debemos redefinir el nodo del árbol, de manera que contenga un campo equi para su factor de equilibrio y por otro lado, el algoritmo debe seguir los siguientes pasos básicos:

i.) Seguir el camino de búsqueda, y determinar si la clave está o no en el árbol. Si ya existía, cortar el proceso.

ii.) Si la clave no existía, insertar el nuevo nodo como una hoja, con *equi = 0*.

iii.) Volver atrás en el camino de búsqueda, y comprobar el *equi* de cada nodo en ese retorno. En el primer nodo que se encuentre desequilibrado, aplicar la rotación que sea necesaria. Esa rotación devolverá el equilibrio a todo el árbol.

Como la idea es no sólo recorrer hacia adelante el camino de búsqueda de la clave a insertar, sino también recorrerlo hacia atrás al finalizar la inserción para ver si algún nodo en ese camino perdió el equilibrio, es necesario que el algoritmo recuerde por cuáles nodos bajó, para luego poder regresar... y ya sabemos que eso puede hacerse muy naturalmente usando recursividad. Para insertar la clave, un método recursivo recorrerá el camino de búsqueda de la misma, de forma que la propia recursión irá almacenando las direcciones de los nodos por los que pase. Cuando alguna llamada recursiva logre terminar (habiendo insertado un nodo), automáticamente comenzarán a extraerse del stack segment las direcciones de los nodos almacenados, y en cada uno de ellos se verificará el equilibrio.

El algoritmo es extenso y un tanto complejo, pero su funcionamiento se comprende bien si se hace una prueba de escritorio exhaustiva. En el modelo *TSBAVLSearchTree*, presentado anexo a esta ficha, se implementa la idea completa a través de métodos de la clase *TSBAVLSearchTree*. Esta clase deriva de la clase *TSBSearchTree* que ya hemos analizado, por lo cual sólo debemos redefinir los métodos de inserción y borrado: el resto de los métodos son exactamente los mismos que en la clase *TSBSearchTree*. Para incluir el factor de equilibrio como atributo, se define dentro de la clase *TSBAVLSearchTree* una clase interna *AVLTreeNode* que deriva desde la clase *TreeNode* (que a

su vez era interna de *TSBSearchTree*). De esta forma, la nueva clase sólo incorporar el atributo *factor* para almacenar el factor de equilibrio y aprovecha el resto desde su clase base. En la clase *TSBAVLSearchTree* se usan referencias polimórficas de la clase *TreeNode*, y en los casos en que sea necesario se hacen conversiones mediante casting explícito a objetos de la clase *AVLTreeNode*.

En la clase *TSBAVLSearchTree*, están incluidos también los métodos que permiten borrar un nodo del árbol AVL. Por lo que se vió al hacer borrado en árboles de búsqueda, es de esperar que también en los árboles AVL el borrado sea más complicado que la inserción. Básicamente, se procede igual que en el borrado común: si el nodo no tiene hijos, o tiene uno solo, borrarlo es fácil; pero si tiene los dos, se reemplaza por su mayor descendiente izquierdo. Luego de esto, si algún nodo perdió el equilibrio, se aplican las rotaciones adecuadas. A diferencia del algoritmo de inserción, en el que a lo sumo se aplica una rotación, en el borrado puede llegar a ser necesaria una rotación en cada nodo del camino de búsqueda. Dejamos el análisis de los detalles de implementación para los estudiantes.