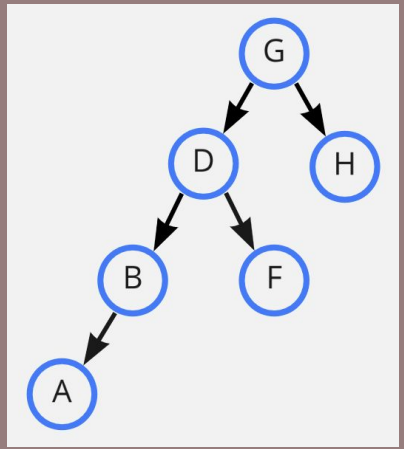
Árboles Binarios de Búsqueda Balanceados

Árbol Binario de Búsqueda Balanceado

Un árbol binario de búsqueda T es balanceado si y sólo si para todo nodo de T se cumple que la distancia máxima del nodo a alguna de sus hojas es a lo más **c** veces la distancia mínima del vértice a alguna de sus hojas, donde **c** es una constante fija.

Este árbol es 3-balanceado.



ED
ESTRUCTURAS DE DATOS

Rotaciones

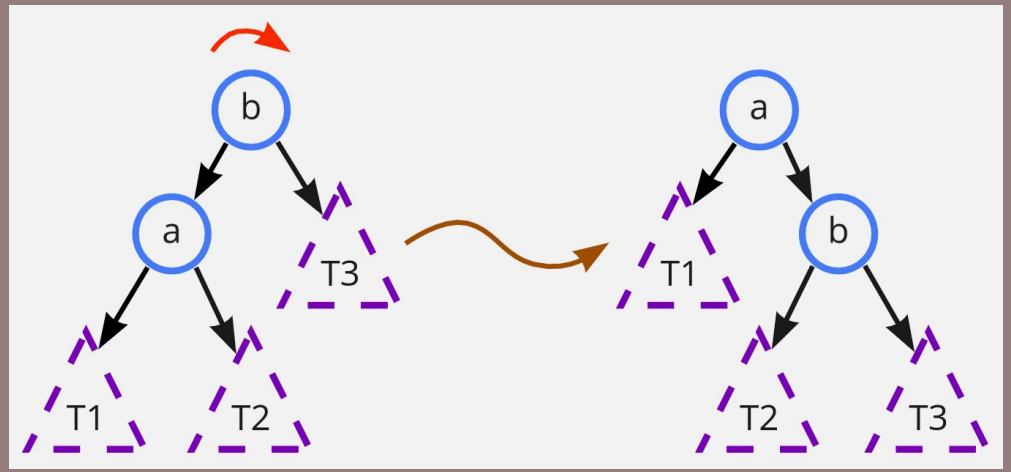
Para balancear un BST hay que reacomodarlo, por lo que se tienen funciones de rotación.

Un nodo se puede rotar hacia la derecha o hacia la izquierda, el único requisito es que tenga un hijo en la dirección opuesta a la que se va a rotar.

ED
ESTRUCTURAS DE DATOS

Rotar a la derecha

Al rotar a la derecha un nodo, su hijo izquierdo se vuelve su padre y el subárbol derecho del que antes era su hijo, se vuelve su subárbol izquierdo.

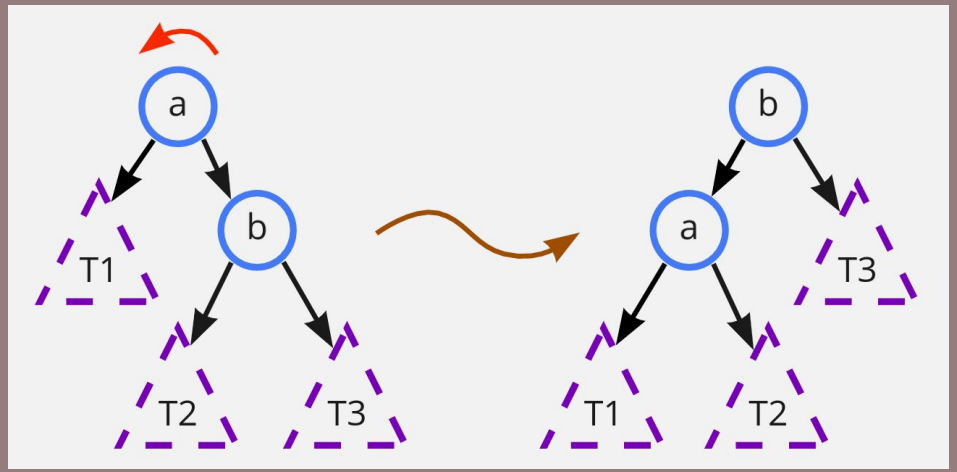


Nota que el orden del BST se preserva.

ED
ESTRUCTURAS DE DATOS

Rotar a la izquierda

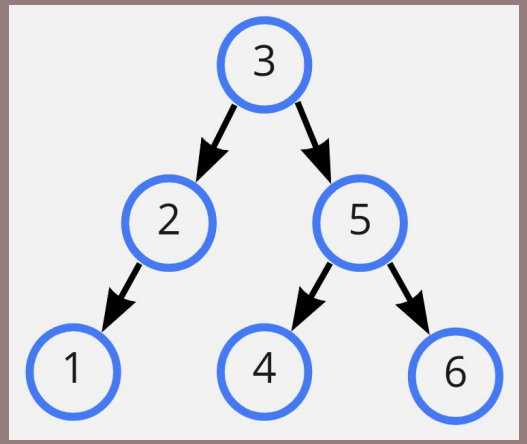
Al rotar a la izquierda un nodo, su hijo derecho se vuelve su padre y el subárbol izquierdo del que antes era su hijo, se vuelve su subárbol derecho.



Nota que el orden del BST también se preserva.

Ejercicio

Rota a la derecha el nodo 3 y luego rota a la izquierda ese mismo nodo en el siguiente BST:



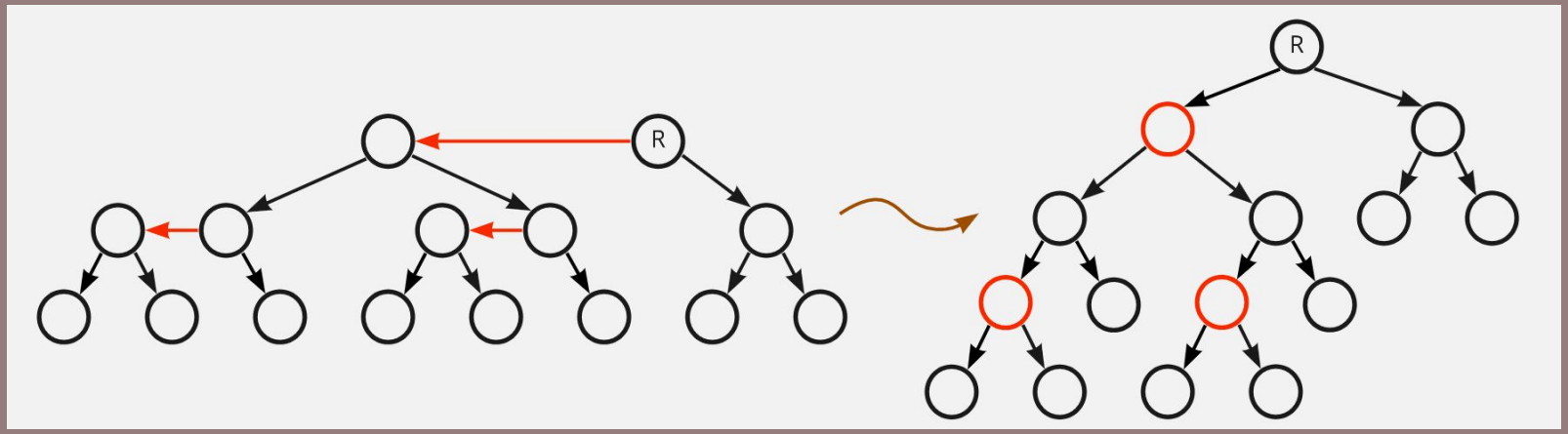
Árboles B-simétricos - 1972 Bayer

Un BST es B-simétrico si cumple las siguientes reglas:

- Puede haber aristas horizontales o descendientes.
- Los caminos desde la raíz a cualquier hoja tienen el mismo número de aristas descendientes.
- No hay 2 aristas horizontales adyacentes.

Estos árboles son 2-balanceados.

En 1978, Guibas y Sedgewick replantearon los árboles B-simétricos e hicieron distinguibles a los vértices en lugar de las aristas.



Un nodo es rojo si la arista de su padre es horizontal, negro si es descendiente.

ED
ESTRUCTURAS DE DATOS

Árboles Rojinegros - 1978 Bayer

Un árbol binario de búsqueda **T** es rojinegro si cumple las siguientes propiedades:

1. Todos los vértices son **rojos** o **negros**.
2. La raíz es **negra**.
3. Las hojas vacías (*null*) son **negras**.
4. No hay dos vértices **rojos** consecutivos (los vértices **rojos** solo tienen hijos **negros**).
5. Para todo vértice **v** ϵ **T**, todas las trayectorias de **v** a alguna de sus hojas descendientes tienen el mismo número de vértices **negros**.

ED
ESTRUCTURAS DE DATOS

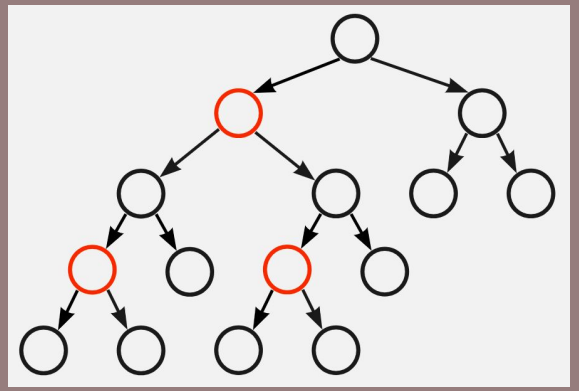
Simulador de Árboles Rojinegros

Pedro Ulises Cervantes González - Facultad de Ciencias

Búsqueda en Árboles Rojinegros

Como un árbol rojinegro es un BST, el algoritmo de búsqueda sigue siendo el mismo.

Un árbol rojinegro es 2-balanceado, así que podemos decir que $T(n) \in O(\log n)$.



ED
ESTRUCTURAS DE DATOS

Inserción en Árboles Rojinegros

Al agregar un nuevo nodo en un árbol rojinegro le asignaremos el color rojo.

Sin embargo, hacer esto puede violar las propiedades 2 o 4, por lo que hay que hacer una corrección que llamaremos *rebalanceo*.

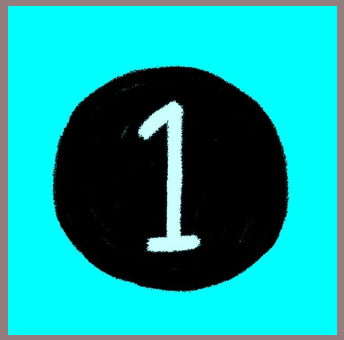
Algoritmo para agregar un elemento **e** a un árbol rojinegro:

1 Insertamos **e** al árbol como un BST cualquiera.
2 El color asignado a **v**, el nodo donde se guarda **e**, es rojo.
3 Rebalanceamos sobre **v**.

* A partir de aquí es conveniente que cada nodo tenga una referencia a su padre.

ED
ESTRUCTURAS DE DATOS

Rebalanceo de un nodo rojo v.



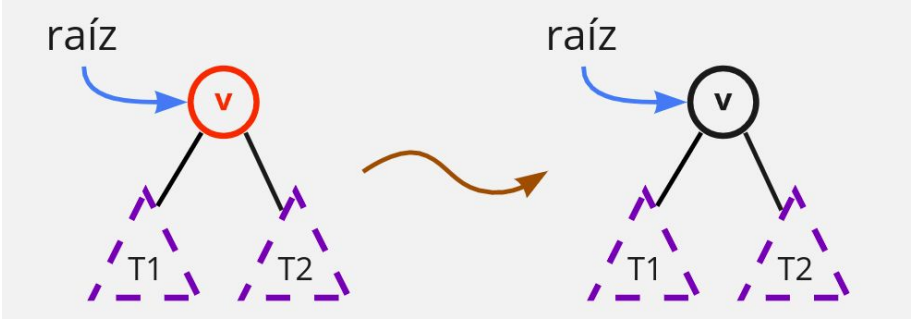
Caso 1. El nodo v no tiene padre:

1 Se colorea v de **negro.**

2 Terminamos.

Explicación:

Si v no tiene padre, quiere decir que es la raíz del árbol, por lo que la propiedad que se está violando es la 2.
Al colorearlo de **negro** se corrige esa propiedad.



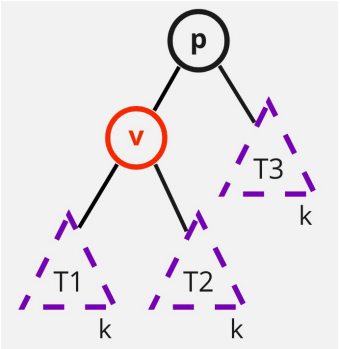
Rebalanceo de un nodo rojo v.

Sea **p** el padre de **v**.

Caso 2. El nodo **p** es **negro**:
1 Terminamos.

Explicación:

Recordemos que las propiedades que estamos corrigiendo son la 2 y la 4 solamente.
Nuestro balanceo funciona de abajo hacia arriba (en casos posteriores se verá mejor esto) por lo que podemos decir que **v** no tiene conflictos con T1 o T2, siendo las raíces de estos árboles de color negro.
Por lo tanto no hay ninguna violación y nuestro árbol ya está balanceado.

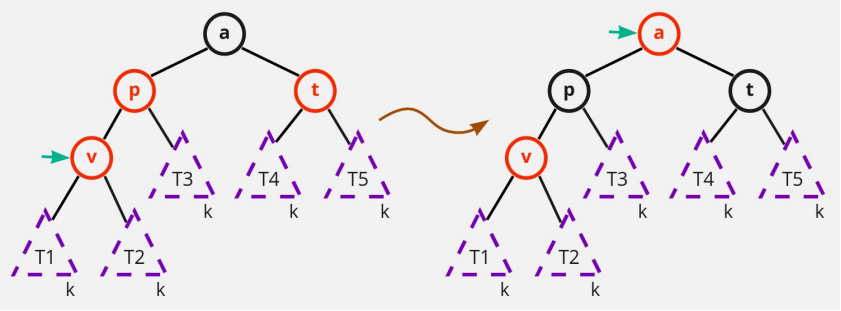


Rebalanceo de un nodo rojo v.

El padre **p** es **rojo**.
Sea **a** el padre de **p**, entonces **a** es **negro**.
Sea **t** el hermano de **p**.

Caso 3. El nodo **t** es **rojo**:
1 Se recolorean **p** y **t** de **negro** y **a** de **rojo**.
2 Se aplica recursión. Ahora se rebalancea el nodo **a**.

Explicación:



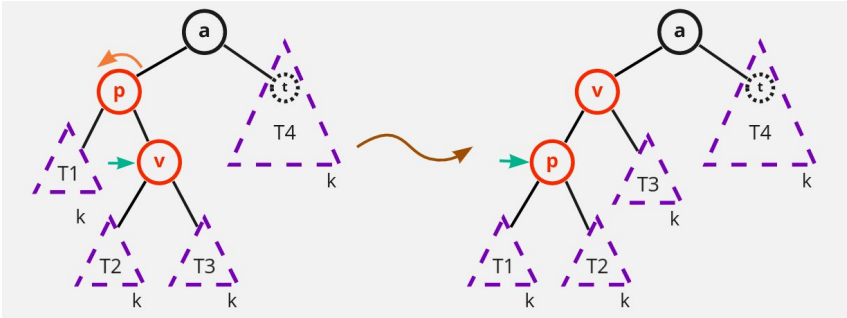
Al recolorear de esta manera se resuelve el conflicto entre **p** y **v**, además sigue habiendo k+1 vértices negros en cualquier camino, por lo que no se rompe la propiedad 5.
Al cambiar el color de **a** tal vez violemos la propiedad 2 o 4, por lo que el rebalanceo debe hacerse desde ahí.

Rebalanceo de un nodo rojo v.

Caso 4. Los nodos v y p están cruzados. Esto es, p es hijo izquierdo y v derecho, o bien, p es hijo derecho y v izquierdo:

1 Se rota p en su dirección. Si es hijo izquierdo se rota a la izquierda y si es derecho a la derecha.
2 El rebalanceo sigue sobre p, pasando al caso 5.

Explicación:



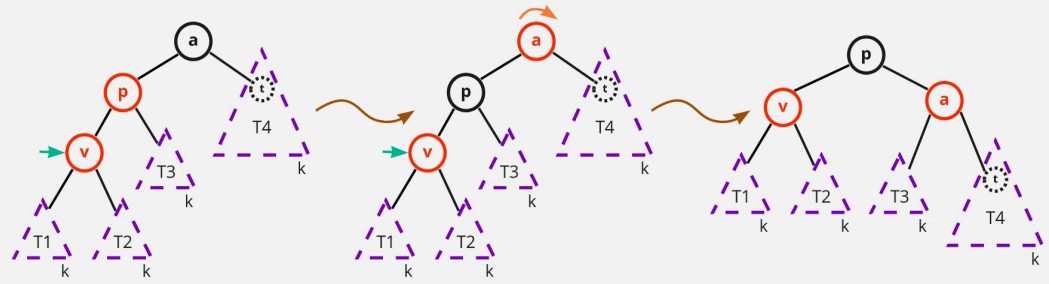
Al hacer la rotación no se resuelve el conflicto de la propiedad 4. En realidad esto es una preparación para el caso 5, por eso el rebalanceo se sigue sobre p.

Rebalanceo de un nodo rojo v.

Caso 5. Los nodos **v** y **p** no están cruzados:

1 Se recolorea **p** de **negro** y **a** de **rojo**.
2 Se rota **a** en la dirección contraria a **v**. Si **v** es hijo izquierdo se rota a la derecha y si es derecho a la izquierda.
3 Terminamos.

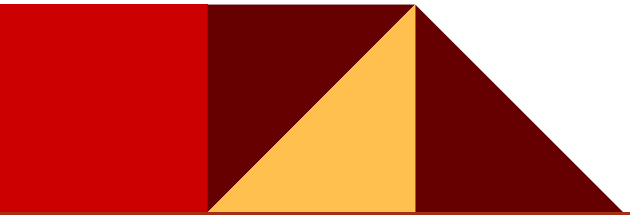
Explicación:



Al recolorear **a** y **p** resolvemos los conflictos de la propiedad 4, puesto que **t** es de color negro.
Al rotar **a**, nos aseguramos que no haya conflicto ascendente de la propiedad 4 y conservamos k+1 nodos negros en cualquier camino.

Ejercicio

Inserta en un árbol rojinegro los elementos 11, 22, 33, 5 y 7.



Eliminación en un BST

Para eliminar tenemos este algoritmo:

1 Buscar al nodo **v** que contiene el elemento a eliminar.
2 a) Si es hoja, se remueve del árbol.
b) Si tiene solo un hijo, se reemplaza **v** con su hijo.
c) Si tiene dos hijos:

Sea **u** el máximo en $T_i(v)$

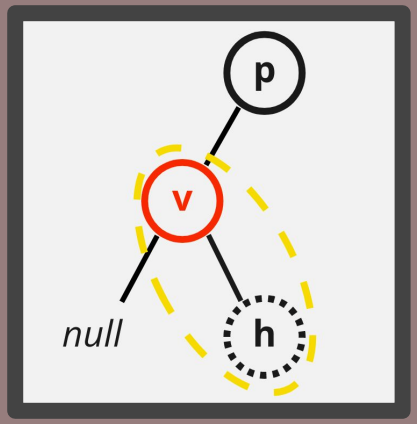
Sea **u** el mínimo en $T_d(v)$

Intercambiar el valor de **v** por el de **u**.
Eliminar **u**.

Podemos considerar que para eliminar una hoja, se puede hacer mediante un ascenso, uno en el que quien asciende es *null*.

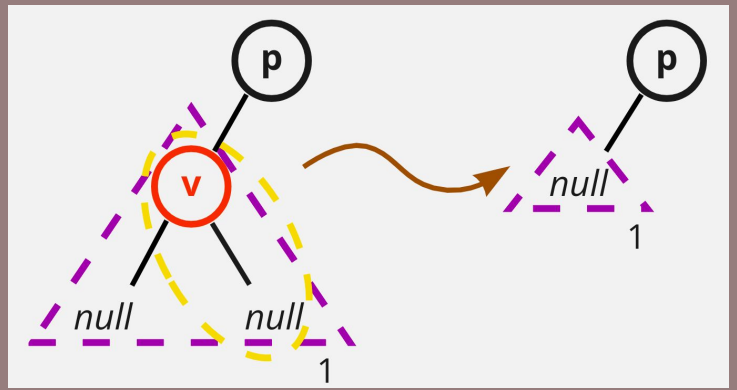
Por lo que el último paso de una eliminación es un ascenso en el que al menos uno de los hijos es *null*.

Exploremos qué casos pueden ocurrir después de ese ascenso en un árbol rojinegro. Para ello nombraremos como **v** al nodo por eliminarse, **p** su padre y **h** el hijo que ascenderá:



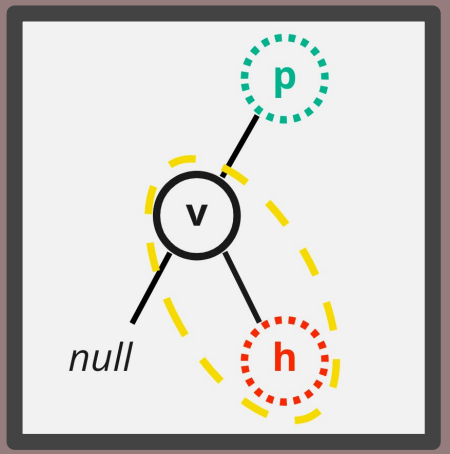
Caso 1. **v** es **rojo** y **h** es **negro**:

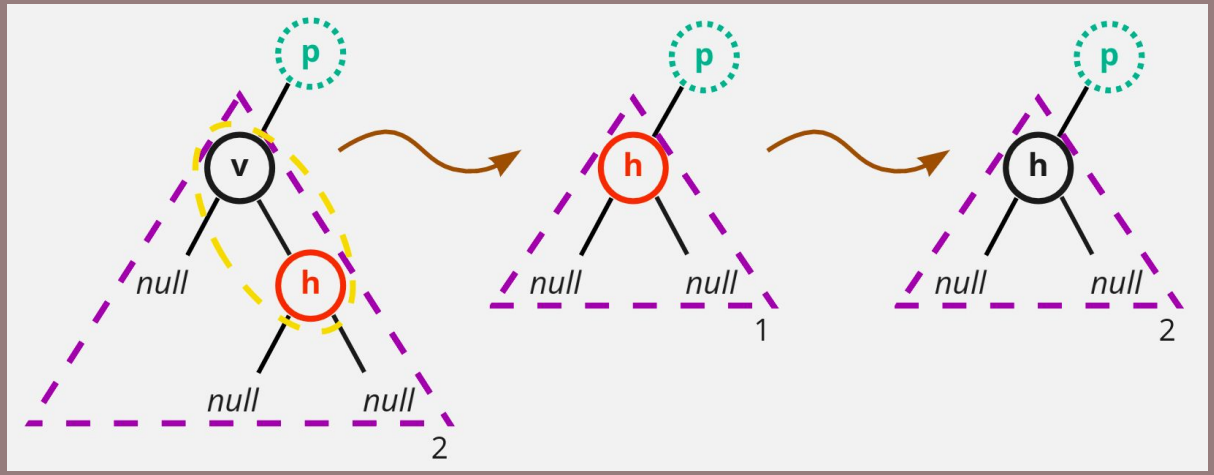
Por la propiedad 5, podemos deducir que **h** es en realidad *null*. Luego, al eliminar **v**, se siguen cumpliendo todas las propiedades de un árbol rojinegro pues la cantidad de nodos negros en cualquier camino del subárbol sigue siendo 1.

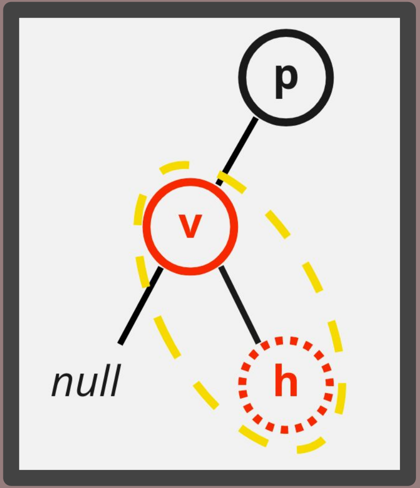


Caso 2. v es **negro** y h es **rojo**:

Por la propiedad 3 y 5, podemos deducir que **h** es en realidad una hoja. Luego, a eliminar **v**, se reduce la cantidad de nodos negros en cualquier camino de 2 a 1, además de tal vez crear un conflicto con **p** si es rojo. Para cubrir esa pérdida de un nodo negro y evitar conflictos, se recolorea **h** y de esta forma se mantienen todas las propiedades en un árbol rojinegro.







Caso 3. v es rojo y h es rojo:

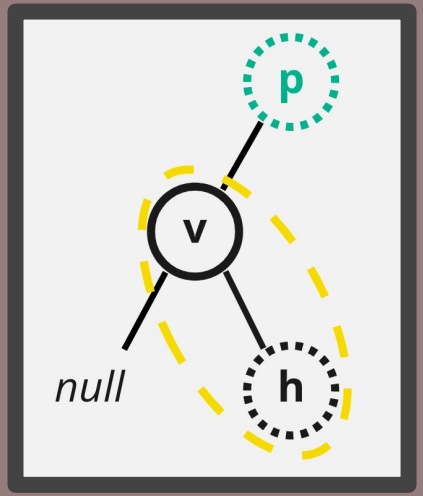
Este caso [illegible] da para tanto.

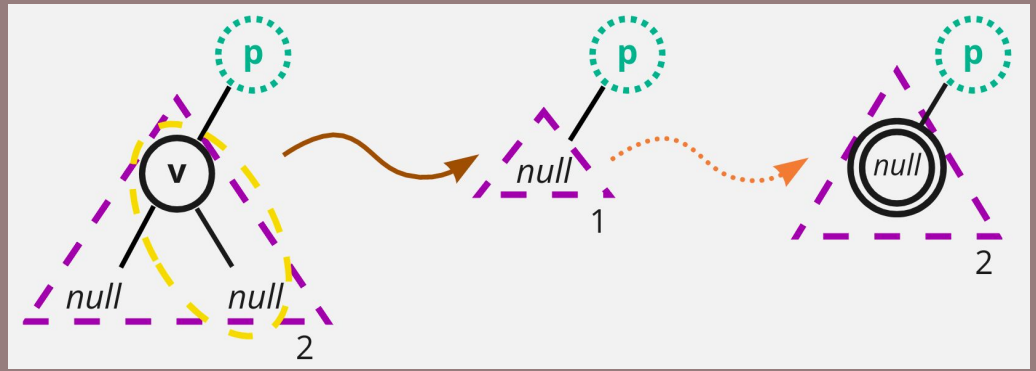
ED
ESTRUCTURAS DE DATOS

Caso 4. **v** es **negro** y **h** es **negro**:

Por la propiedad 5, podemos deducir que **h** es en realidad *null*. Luego, al eliminar **v**, se reduce la cantidad de nodos negros en cualquier camino de 2 a 1.

No se puede resolver la violación de la propiedad 5 solo recoloreando, pues no hay manera de de agregarle más ***negrura*** a un vértice con tal de compensar la pérdida de nodos negros.





Eliminación en Árboles Rojinegros

Algoritmo para eliminar un elemento **e** de un árbol rojinegro:

1 Procedemos a eliminar **e** del árbol como un BST cualquiera, pero antes de la ascención final revisamos:
   a) El color de **v**, el nodo a eliminar.
   b) El color de **h**, el hijo que ascenderá.

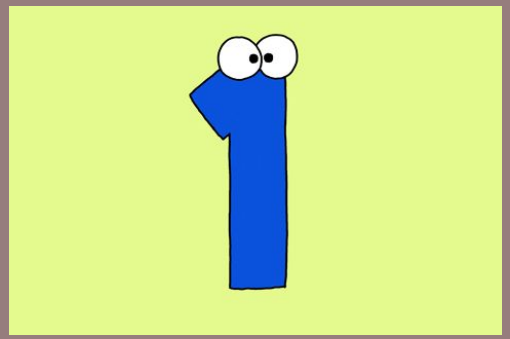
2 Hacemos la revisión de casos:
   a) Si **v** es rojo, terminamos.
   b) Si **h** es rojo, lo coloreamos de **negro** y terminamos.
   c) Si **v** y **h** son **negros** (**v** es una hoja **negra**):
      1. Envolvemos a **h** en un nodo **negro**.
      2. Rebalanceamos sobre **h**.
      3. Desenvolvemos a **h**.

Gracias a los casos que revisamos podemos obtener este algoritmo de eliminación.

Para la resolución de la violación de la propiedad 5 cuando ambos vértices son negros se aplicará lo que llamaremos *rebalanceo*.

A diferencia de la inserción, este rebalanceo solo recibe vértices negros que, solo teóricamente, consideraremos doblemente negros.

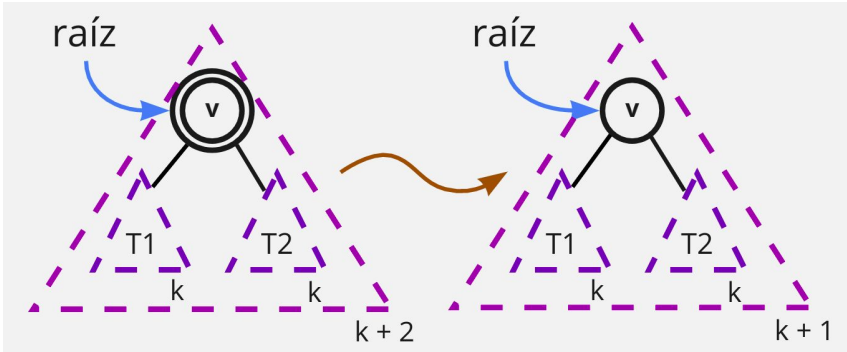
Rebalanceo de un nodo negro v.



Caso 1. El nodo v no tiene padre:
1 Terminamos.

Explicación:

Si v no tiene padre, quiere decir que es la raíz del árbol.
Una violación en la propiedad 5 ocurre cuando la cantidad de nodos negros en cualquier camino varía respecto a la cantidad que hay en el subárbol de su hermano, pero al ser la raíz no tiene hermano, por lo que no hay conflicto.
La cantidad de nodos negros en cualquier camino solo se reduce.



Rebalanceo de un nodo negro v.

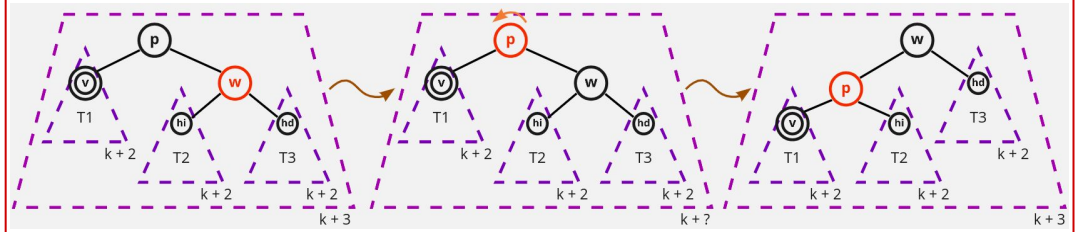
Sea **p** el padre de **v**. Como **v** está compensando la pérdida de un nodo negro, forzosamente debe tener un hermano **w** distinto de *null*.

Caso 2. El nodo **w** es **rojo**:

1. Se recolorea **p** de **rojo** y **w** de **negro**.
2. Se rota **p** en la dirección de **v**. Si es hijo izquierdo se rota a la izquierda y si es derecho a la derecha.
3. Se actualizan referencias, pues uno de los sobrinos de **v** ahora es su hermano.
4. Se sigue con la verificación de los casos 4, 5 y 6.

Explicación:

Este caso no resuelve el conflicto, lo que hace al recolorear y rotar es no crear nuevos conflictos y preparar el escenario para casos posteriores.

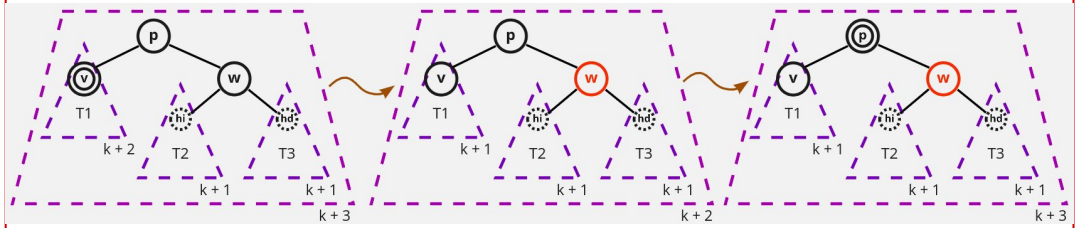


Rebalanceo de un nodo negro v.

El hermano **w** es **negro**.
Sean **hi** y **hd** los hijos izquierdo y derecho de **w** respectivamente.

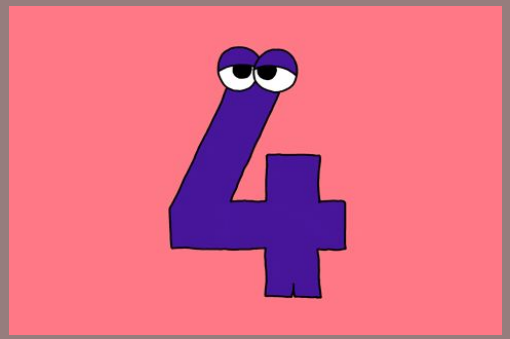
Caso 3. Los nodos **p**, **hi** y **hd** son **negros**:
1 Se colorea **w** de **rojo**.
2 Se aplica recursión. Ahora se rebalancea el nodo **p**.

Explicación:



Al recolorear **w**, ya no es necesario que **v** compense la pérdida de un vértice negro, por lo que deja de ser doblemente negro.
Sin embargo, el árbol perdió un nodo negro, por lo que **p** ahora se considera doblemente negro y para resolver nuevos conflictos se aplica recursión sobre él.

Rebalanceo de un nodo negro v.



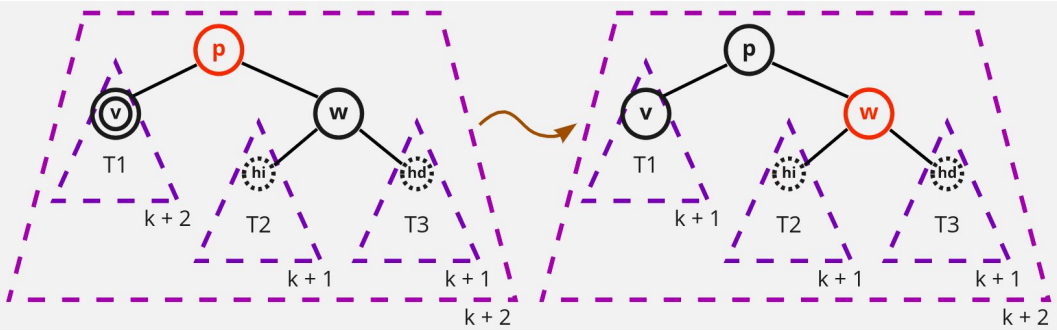
Caso 4. El nodo p es rojo y los nodos hi y hd son negros:

1 Se recolorea p de negro y w de rojo.

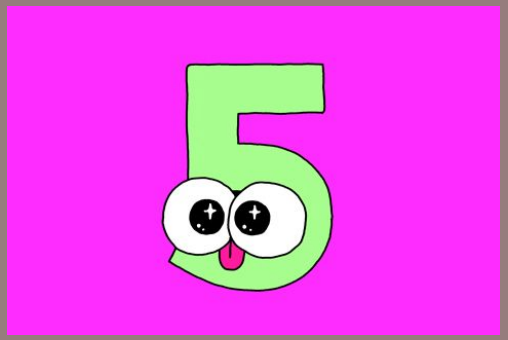
2 Terminamos.

Explicación:

Al igual que el caso anterior, al recolorear w ya no es necesario que v compense la pérdida de un vértice negro, por lo que deja de ser doblemente negro.
Para compensar la pérdida de un nodo negro en el árbol se colorea p de negro, lo cual no ocasiona nuevos conflictos, por lo que el árbol ya se encuentra balanceado.



Rebalanceo de un nodo negro **v**.



A partir de aquí el color de **p** es indistinto.

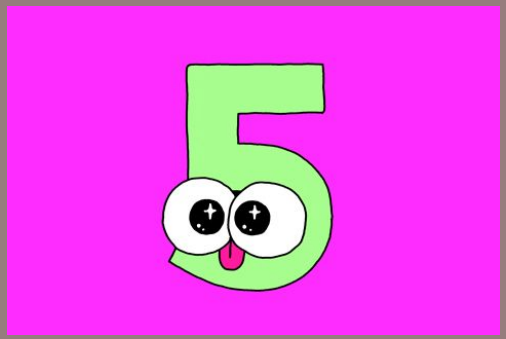
<u>Caso 5.</u> El único sobrino rojo de **v** es el que está en su misma dirección. Si **v** es hijo izquierdo, entonces **hi** es **rojo** y **hd** es **negro**, y si es derecho, entonces **hi** es **negro** y **hd** es **rojo**:

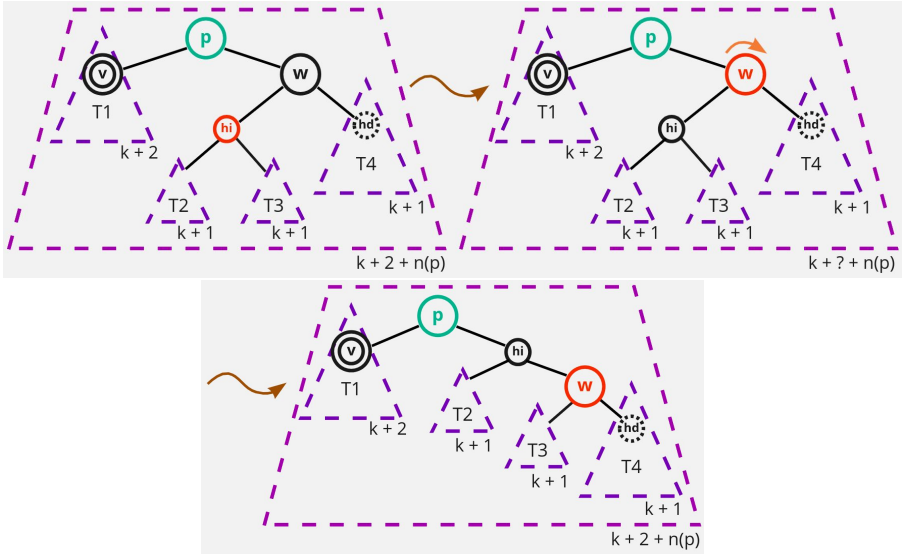
1 Se recolorea **w** de **rojo** y el sobrino rojo de **v** de **negro**.
2 Se rota **w** en la dirección contraria a **v**. Si **w** es hijo izquierdo se rota a la derecha y si es derecho a la izquierda.
3 Se actualizan referencias, pues uno de los sobrinos de **v** ahora es su hermano.
4 Sigue el caso 6.

Explicación:

Estás en lo siguiente.

Rebalanceo de un nodo negro v.



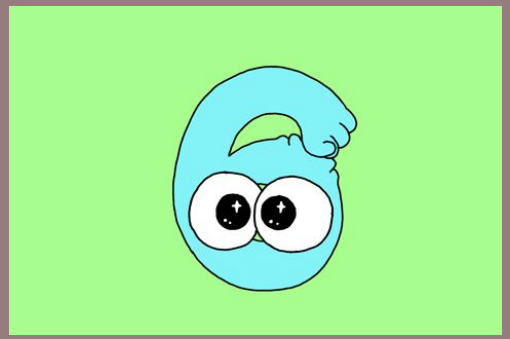


Explicación del caso 5:

Este caso no resuelve el conflicto, lo que hace al recolorear y rotar es no crear nuevos conflictos y preparar el escenario para el caso 6.

La función n(ρ) devuelve 1 si ρ es negro, 0 si es rojo. El número de nodos negros en cualquier camino prevalece al final de este caso.

Rebalanceo de un nodo negro v.



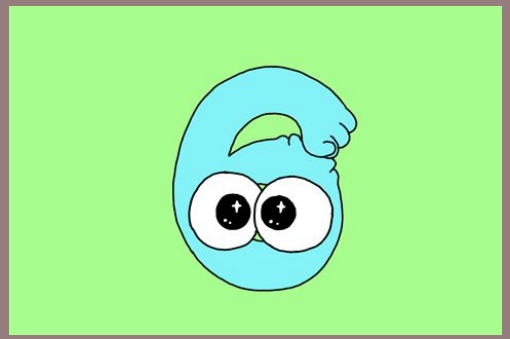
Caso 6. El sobrino en la dirección opuesta de **v** es **rojo**. Si **v** es hijo izquierdo, entonces **hd** es **rojo**, y si es derecho, entonces **hi** es **rojo**. (El color del sobrino que no está en la dirección opuesta nos es indiferente):

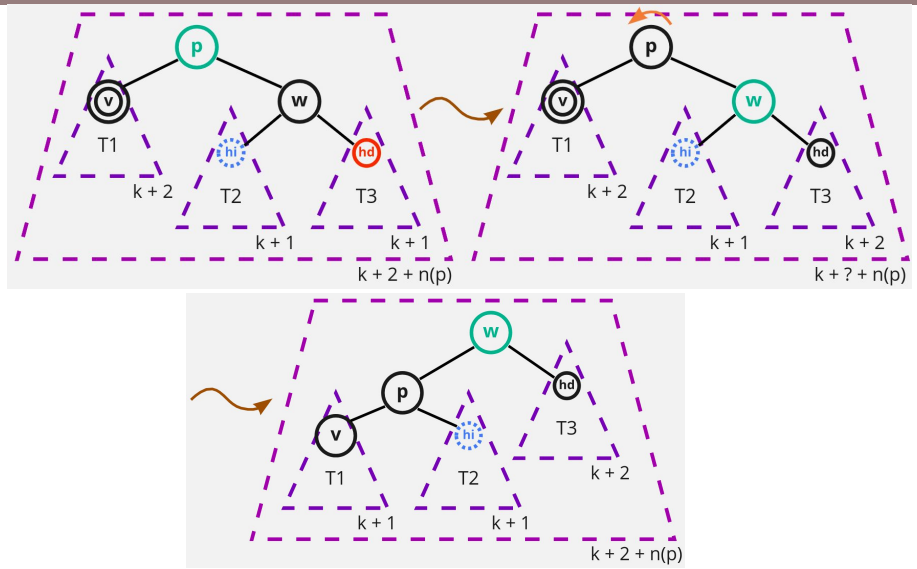
1 Tanto **ρ** como el sobrino en la dirección opuesta de **v** de colorean de **negro**, y **w** se colorea con el color original de **ρ**.
2 Se rota **ρ** en la dirección de **v**. Si **v** es hijo izquierdo se rota a la izquierda y si es derecho a la derecha.
3 Terminamos.

Explicación:

Está en la siguiente diapositiva.

Rebalanceo de un nodo negro v.



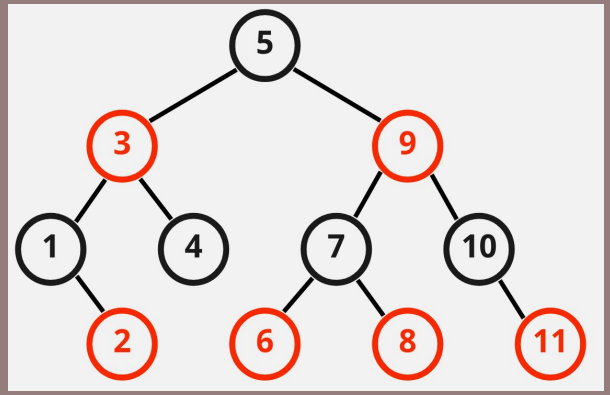


Explicación:

Al recolorear y rotar de esta manera, en el árbol resultante se tiene la misma cantidad de nodos negros en cualquier dirección que en el árbol original sin necesitar que v sea doblemente negro. El color de la raíz del árbol resultante es del mismo color que la raíz del árbol original, por lo que no se generan nuevas violaciones de las demás propiedades. Logramos balancearlo.

Ejercicio

Elimina del siguiente árbol rojinegro los elementos 4, 3, 5, 9, 8, 7, 1, 2 y 11 en ese orden:



Para construir este árbol en el simulador, inserta los elementos en este orden: 5, 3, 9, 1, 4, 7, 10, 2, 6, 8, 11

Complejidad en Árboles Rojinegros

Aunque hay varias rotaciones y recoloreos al insertar o eliminar, solo toma un tiempo constante realizar esas acciones. Además insertar y eliminar solo poseen un caso recursivo que se propaga hacia arriba, es decir, en el peor caso llegan a la raíz, por lo que toman tiempo O(log n).

	Tiempo
Búsqueda	O(log n)
Inserción	O(log n)
Eliminación	O(log n)

Hay varios árboles rojinegros en el kernel de Linux

ED
ESTRUCTURAS DE DATOS

Altura

Recordemos cómo habíamos definido la altura. Sea v un nodo de un árbol binario, la altura del nodo se denota **h(v)** donde:

- Si v es *null*, entonces **h(v) = -1**
- Sean v_i y v_d los hijos de v, entonces **h(v) = 1 + max(h(v_i), h(v_d))**

Al programarlo, el cálculo de las alturas va de las hojas a la raíz.

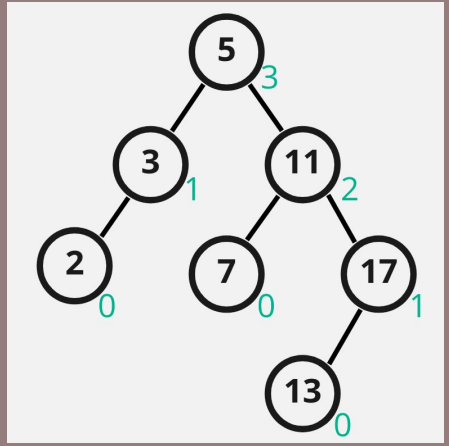
ED
ESTRUCTURAS DE DATOS

Árboles AVL - 1962 Adelson-Velsky y Evgenii Landis

Un árbol binario de búsqueda **T** es AVL si cumple la siguiente propiedad:

- Para todo nodo no vacío **v** ϵ **T**, siendo **hi** y **hd** sus hijos, se cumple que **|** h(**hi**) - h(**hd**) **|** < 2.

En los árboles AVL es conveniente que cada nodo conozca su altura.



ED
ESTRUCTURAS DE DATOS

Simulador de Árboles AVL

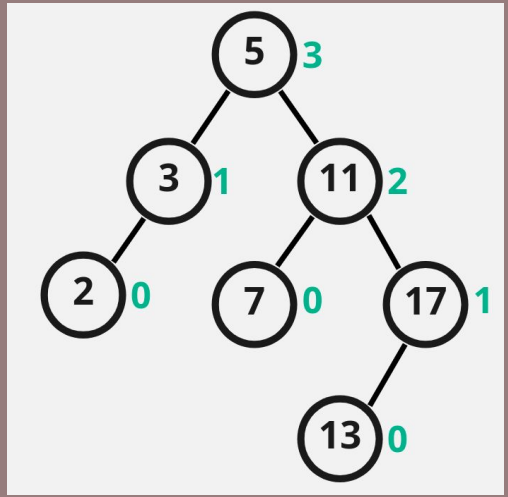
Pedro Ulises Cervantes González - Facultad de Ciencias

Búsqueda en Árboles AVL

Como un árbol AVL es un BST, el algoritmo de búsqueda sigue siendo el mismo que en un BST cualquiera.

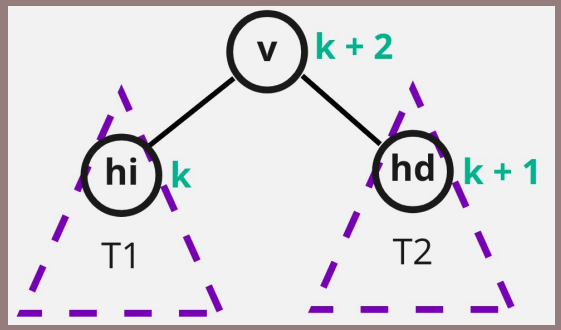
La condición de un AVL obliga que si la longitud de la trayectoria mínima de un vértice cualquiera a alguna de sus hojas es **k**, entonces la longitud de trayectoria máxima será a lo más **k + 1**.

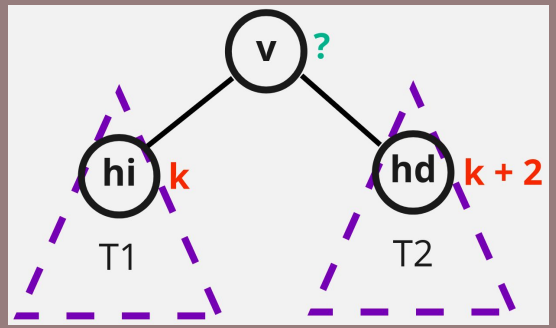
Así que podemos considerar a los árboles AVL balanceados, por lo que en la búsqueda $T(n) \in O(\log n)$.

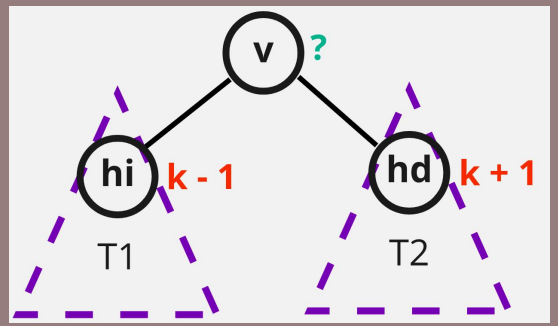


ED
ESTRUCTURAS DE DATOS

¿Qué puede desbalancear a un Árbol AVL?







$$| h(\text{hi}) - h(\text{hd}) | = 2$$

Teniendo un nodo balanceado en el que la altura de sus hijos sea distinta, si aumenta la altura del lado del hijo de mayor altura o si disminuye la altura del lado del hijo de menor altura, obtenemos un desbalance.

ED
ESTRUCTURAS DE DATOS

En un desbalance sucede que para cierto nodo **v**, siendo sus hijos **hi** y **hd**, entonces | h(**hi**) - h(**hd**) | = 2.

$$
\begin{aligned}
&| h(\textbf{hi}) - h(\textbf{hd}) | = 2 \\
\Rightarrow\ &(h(\textbf{hi}) - h(\textbf{hd}) = 2) \vee (h(\textbf{hi}) - h(\textbf{hd}) = -2) \\
\Rightarrow\ &(h(\textbf{hi}) = h(\textbf{hd}) + 2) \vee (h(\textbf{hd}) = h(\textbf{hi}) + 2)
\end{aligned}
$$

Por lo que un desbalance siempre ocurre cuando la altura de uno de los hijos de un nodo difiere en 2 unidades de la altura del otro hijo.

ED
ESTRUCTURAS DE DATOS

Inserción en Árboles AVL

Al agregar un nuevo nodo en un árbol AVL puede ocurrir que la altura de algunos nodos aumente provocando un desbalance , así que hay que actualizar las alturas y rebalancear después de insertar un nodo.

Algoritmo para agregar un elemento **e** a un árbol AVL:

1 Insertamos **e** al árbol como un BST cualquiera.
2 Sea **v** el nodo donde se guarda **e**, actualizamos alturas y rebalanceamos desde **v** hasta la raíz.

ED
ESTRUCTURAS DE DATOS

Eliminación en Árboles AVL

Al remover un nodo de un árbol AVL puede ocurrir que la altura de algunos nodos disminuya provocando un desbalance , así que hay que actualizar las alturas y rebalancear después de eliminar un nodo.

Algoritmo para eliminar un elemento **e** de un árbol AVL:

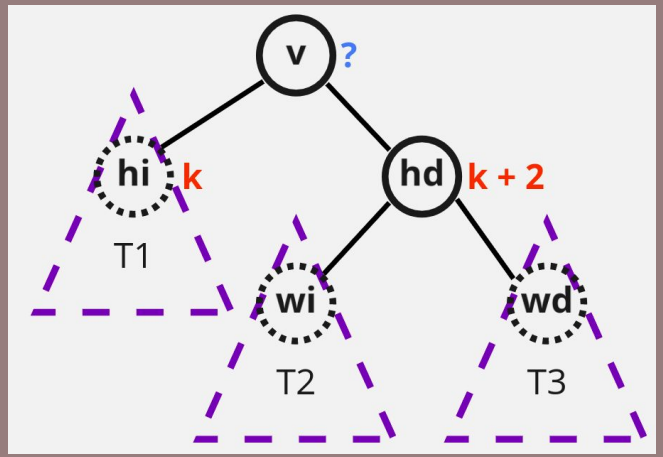
1 Eliminamos **e** del árbol como un BST cualquiera.
2 Sea **v** el nodo eliminado, actualizamos alturas y rebalanceamos desde **el padre de v** hasta la raíz.

ED
ESTRUCTURAS DE DATOS

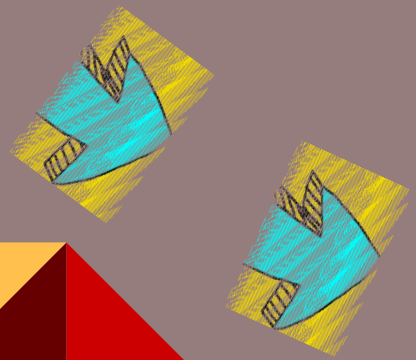
Desbalance a la derecha

Esto ocurre cuando h(**hd**) = h(**hi**) + 2.

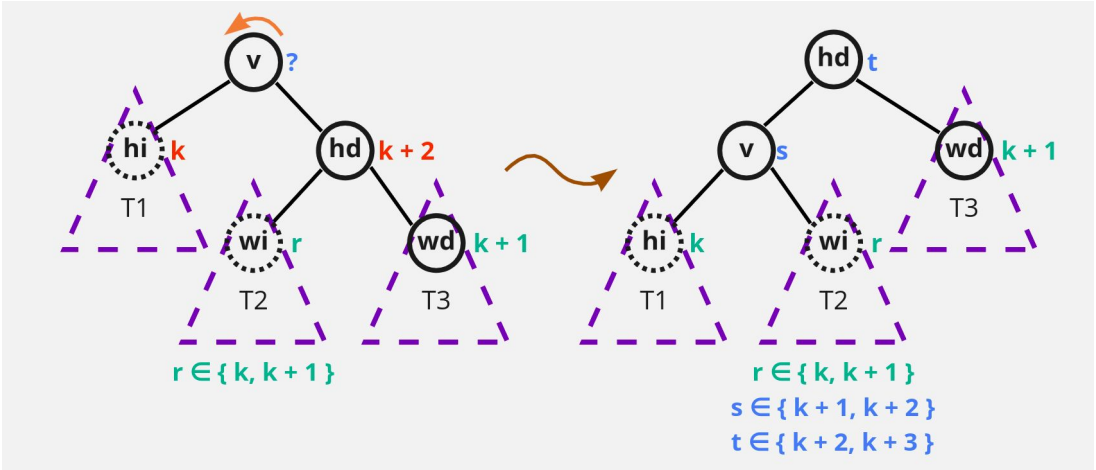
Consideremos que la altura de **hi** es k y la de **hd** es k + 2. Los hijos izquierdo y derecho de **hd** son **wi** y **wd** respectivamente.



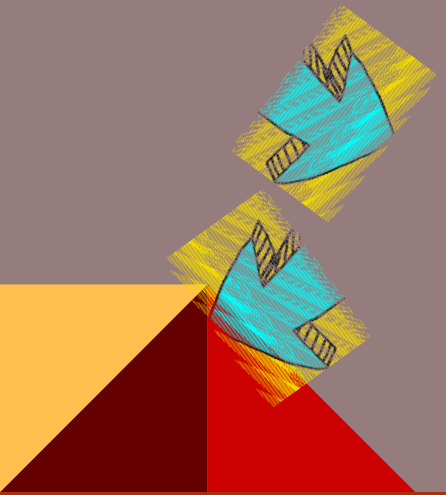
Desbalance a la derecha - Caso 1 (línea recta)



Caso 1. La altura de **wd** es k + 1
(dicho de otra manera, h(**wd**) = h(**hd**) - 1):
1 Se rota **v** a la izquierda.
2 El recálculo de alturas sigue en **hd**.

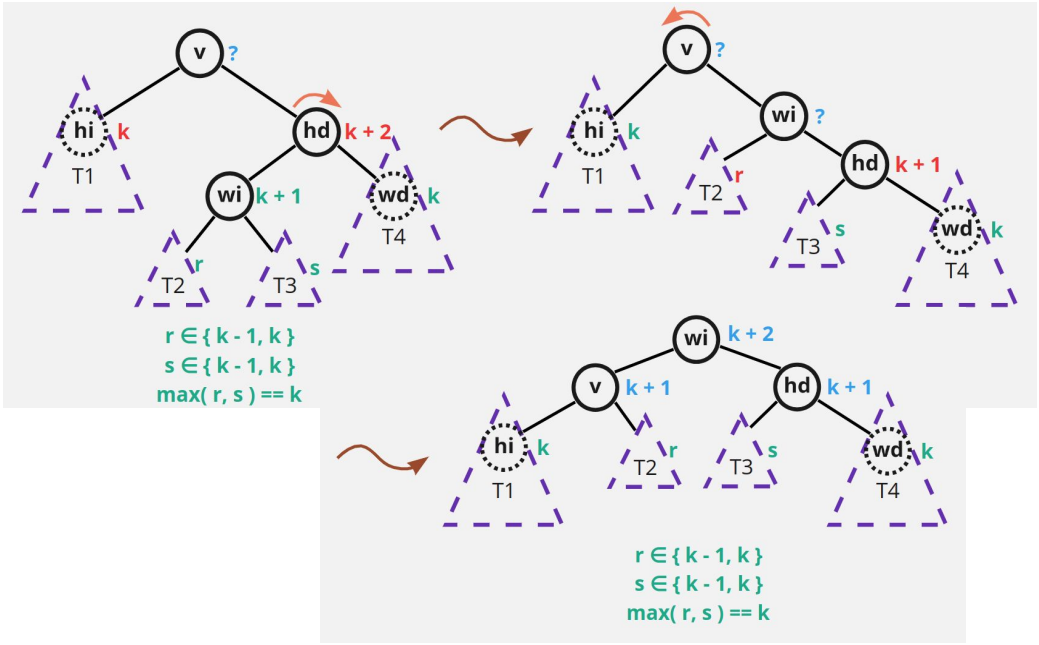


Desbalance a la derecha - Caso 2 (zig zag)



Caso 2. La altura de **wd** es k
(dicho de otra manera, h(**wd**) = h(**hd**) - 2):

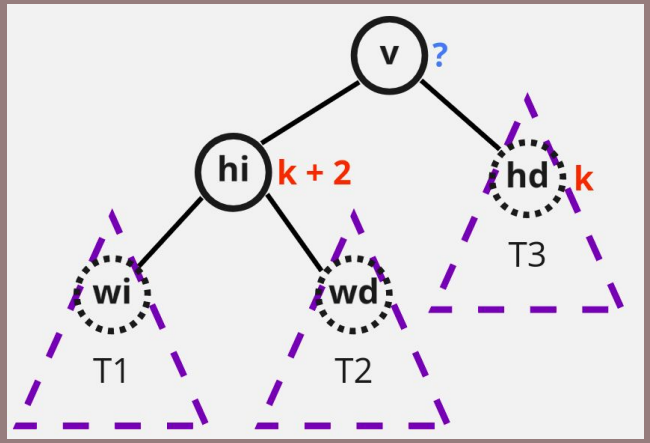
1 Se rota **hd** a la derecha.
2 Se rota **v** a la izquierda.
3 El recálculo de alturas sigue en **wi**.



Desbalance a la izquierda

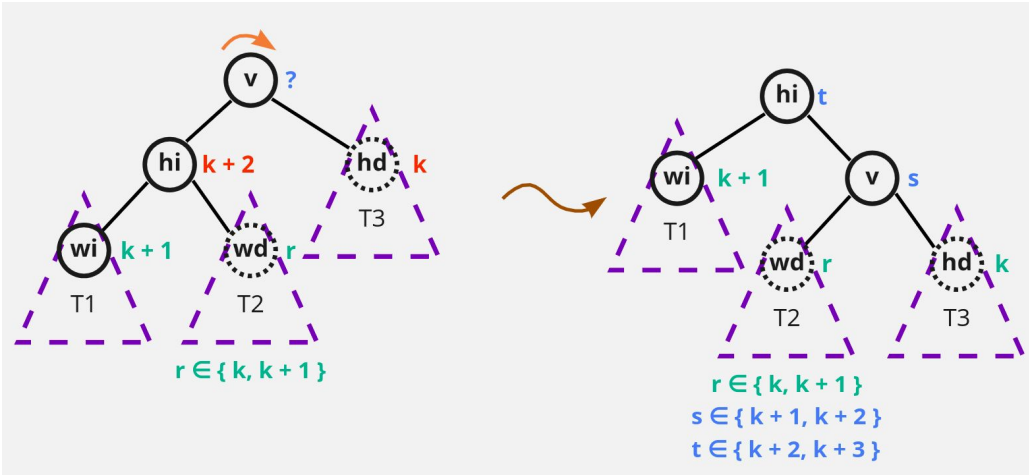
Esto ocurre cuando h(**hi**) = h(**hd**) + 2.

Consideremos que la altura de **hi** es k + 2 y la de **hd** es k. Los hijos izquierdo y derecho de **hi** son **wi** y **wd** respectivamente.

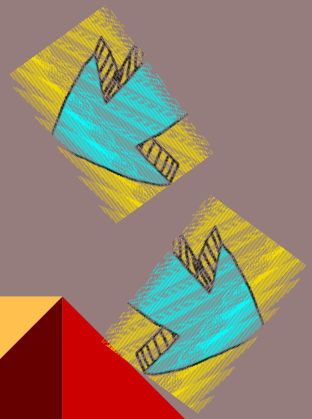


Desbalance a la izquierda - Caso 1 (línea recta)

<u>Caso 1.</u> La altura de **wi** es k + 1
(dicho de otra manera, h(**wi**) = h(**hi**) - 1):
1 Se rota **v** a la derecha.
2 El recálculo de alturas sigue en **hi**.

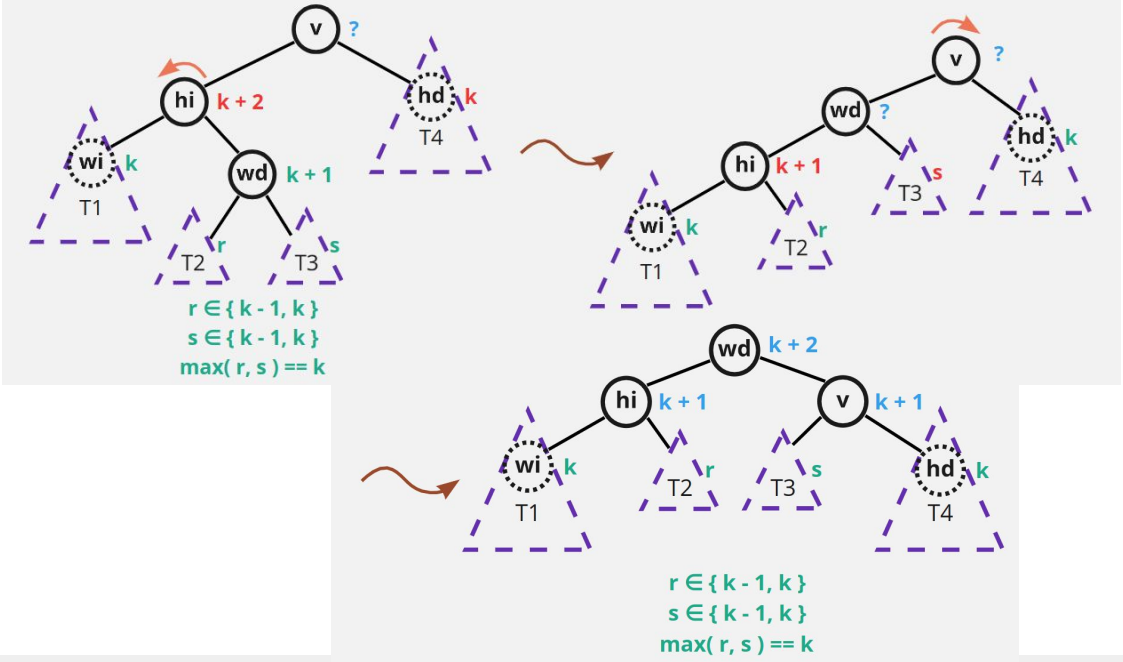


Desbalance a la izquierda - Caso 2 (zig zag)



Caso 2. La altura de **wi** es k
(dicho de otra manera, h(**wi**) = h(**hi**) - 2):

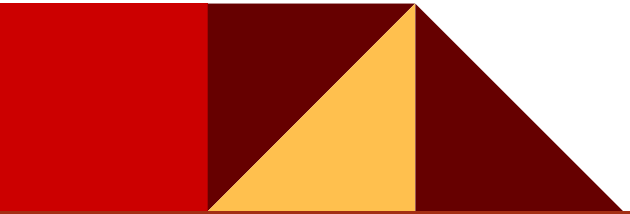
1 Se rota **hi** a la izquierda.
2 Se rota **v** a la derecha.
3 El recálculo de alturas sigue en **wd**.



Ejercicio

Inserta en un árbol AVL en orden los elementos 10, 20, 30, 5, 7, 15 y 40.

Posteriormente elimina los elementos 7, 30 y 40.



Árboles Rojinegros VS Árboles AVL

Recordemos que los árboles AVL son del año 1962 y los árboles rojinegros surgen de los B-simétricos del año 1972, por lo que podemos decir que los rojinegros son una mejor propuesta para implementar árboles 2-balanceados, sin embargo, la complejidad en tiempo es la misma para ambos.

	Rojinegros	AVL
Búsqueda	O(log n)	O(log n)
Inserción	O(log n)	O(log n)
Eliminación	O(log n)	O(log n)

Los árboles AVL son más restrictivos, por lo que se tienen que hacer más rotaciones que en los rojinegros, por eso es que suelen usarse más los rojinegros.

ED
ESTRUCTURAS DE DATOS