75.41 Algoritmos y Programación II
Tda Árbol

Dr. Mariano Méndez[1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

2 de junio de 2020

1. Introducción

Para cantidades grandes o muy grandes de datos, el tiempo de acceso a los datos lineal ($O(n)$) de las listas enlazadas es muy ineficiente [2]. Para ello es necesario trabajar con una estructura de datos capaz de reducir el tiempo de acceso a los mismos, para este fin se estudiará una estructura cuyo tiempo de acceso promedio a los datos es de $O(logN)$ en promedio.

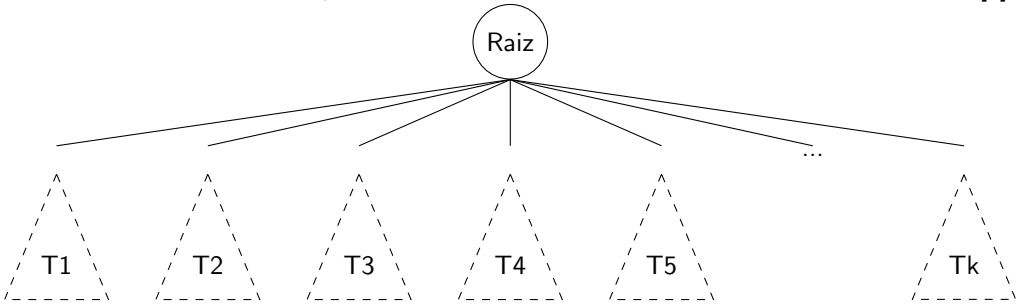
Esta es una abstracción llamada *Árbol*. Por supuesto que existen muchos tipos de variante de esta abstracción:

- Árbol N-arios o Generales.
- Arbol Binario
- Arbol Binario de Búsqueda
- Árbol AVL
- Árbol Splay
- Árbol B
- Árbol B+
- Árbol B*
- Árbol Rojo Negro

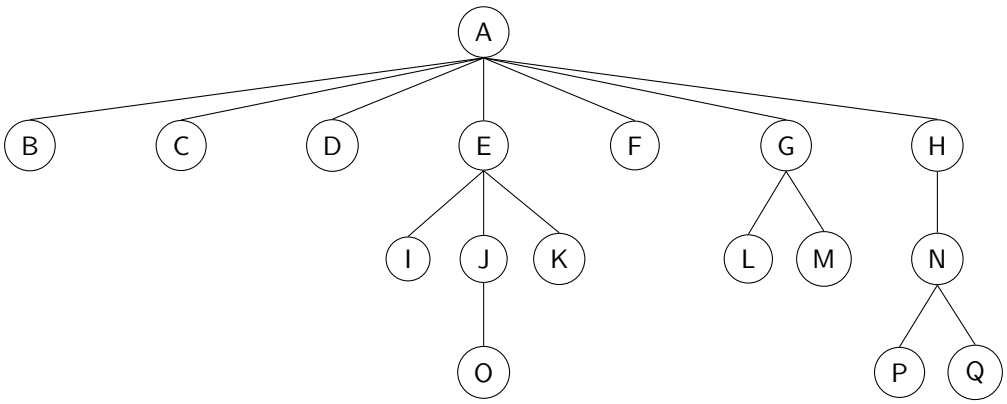
Una característica muy interesante del tipo de dato árbol es su naturaleza altamente recursiva ya sea en sus diversas definiciones y en sus implementaciones.

2. Concepto General y Definición Teórica

Un árbol puede ser definido de varias formas. Una forma natural es hacerlo en forma recursiva. Un *árbol es una* **colección** *de nodos*. Los **nodos** son los elementos o vértices del árbol. Esta colección puede estar vacía, de otra forma, un árbol consiste en un nodo r que se distingue del resto, llamado **raíz**, y cero o muchos sub-árboles no vacíos $T_1, T_2, ..., T_k$, cada uno de ellos posee su raíz conectada mediante un vértice al nodo raíz r [2]:

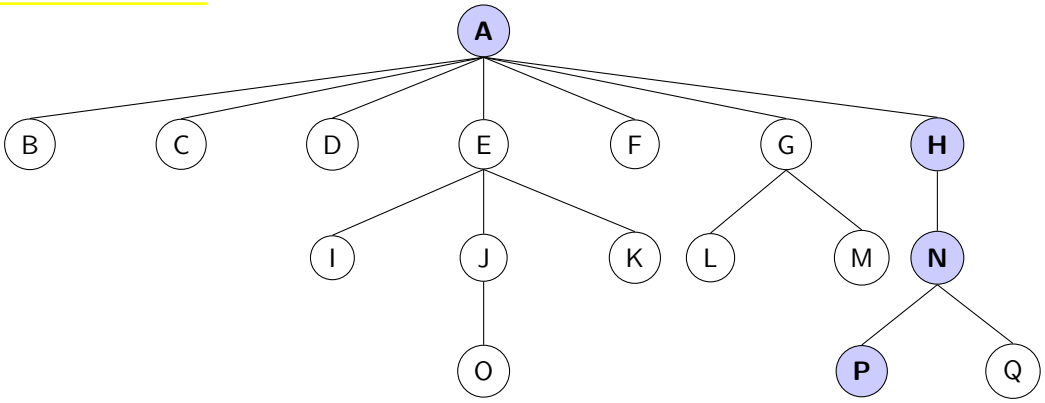


El **nodo raíz** de cada sub-árbol es denominado **nodo hijo** del nodo r (raíz), y r es el **nodo padre** de cada sub-árbol de r. Un ejemplo de un árbol puede verse en la figura:



A partir de la definición recursiva un árbol es una colección de N nodos , uno de los cuales es el nodo **raíz** y N-1 aristas. En la Figura anterior el nodo A es es nodo **raíz** y B,C y D se denominan **nodos hermanos (siblings)**, nodos hermanos son aquellos que tienen el mismo padre. El concepto de nodo abuelo y nodo nieto pueden definirse de igual forma [2].

Camino Un camino (path) desde n_1 a n_2 se define como una secuencia de nodos $n_1, n_2, n_3, n_4, ..., n_k$ tal que n_i es el padre de n_{i+1} para $1 <= i <= k$. La longitud de un camino es el numero de aristas en el camino, $k-1$. Existe un camino de longitud cero entre un nodo y sí mismo. En un árbol existe exactamente un camino entre el nodo raíz y cada nodo del árbol.



Profundidad Para cualquier nodo n_i la profundidad de n_i es la longitud del único camino entre el nodo raíz y el nodo n_i. Por ende la longitud del nodo raíz es 0. La altura (height) de n_i es la altura del camino más largo desde n_i a una hoja. Por ende la altura de un nodo hoja es 0.

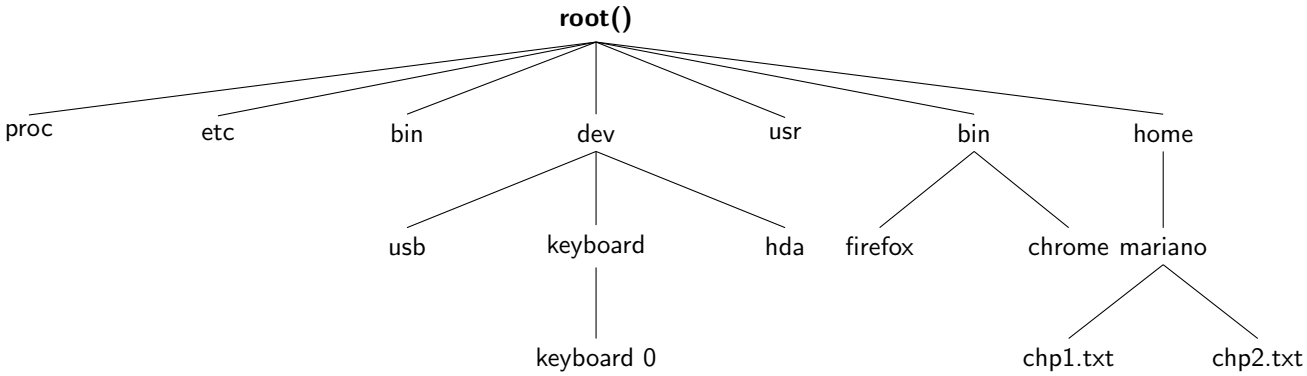
Si existe un camino entre n_1 y n_2, entonces se dice que n_1 es un **ancestro** de n_2 y que n_2 es un **descendiente** de n_1

Una implementacion en C de un árbol general puede ser la siguiente:

```
1
2 typedef struct nodo_arbol * arbol_t
3
4 struct nodo_arbol {
5         void *          elemento;
6         nodo_arbol *    primer_hijo;
7         nod_arbol *     proximo_hermano;
8 };
```

2.1. Ejemplo

Un ejemplo concreto del uso de la estructura anterior es el caso del sistema de archivos de Unix. un sistema de archivos tiene un directorio llamado raíz, este directorio no es hijo de ningún otro directorio. A su vez el directorio raíz puede contener otros directorios o archivos.

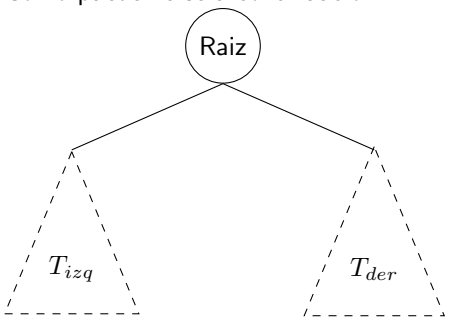


3. Árbol Binario

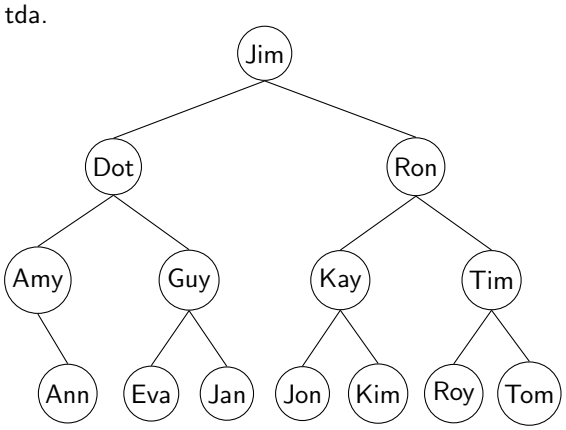
Los árboles binarios están íntimamente relacionados con las operaciones de búsqueda. Cuando se realiza una operación de búsqueda uno debe saber dónde seguir buscando, si a la derecha o a la izquierda de un de terminado valor. Los árboles binarios justamente abstraen ese comportamiento y permiten tener la noción de derecha e izquierda [1].

3.1. Definición

La definición formal de árbol binario es [2, 1]:
un árbol binario puede estar vacío, o consistir en un nodo llamado raíz conjuntamente con dos arboles binarios uno llamado derecha y otro llamado izquierda ambos respecto del nodo raíz.
Como puede verse a continuación:



Cabe destacar que la construcción de este tda es muy fácil de realizar en base a la definición misma de la estructura del tda.



3.2. Operaciones

las operaciones básicas de un árbol binario son:

- crear
- destruir
- vacio

- insertar
- eliminar
- buscar
- recorrer

3.2.1. Recorrido

Dentro de las operaciones más importantes que se realizan con los árboles binarios se encuentra el recorrido. No existe una única forma de recorrer un árbol binario, sino que tres formas.

Recorrer (to traverse) un árbol significa de alguna forma pasar por cada uno de los nodos. Si nombramos al nodo actual como N, al subarbol derecho como D y al subarbol izquierdo como I. Existen seis formas de recorrerlos:
$NID \quad IND \quad IDN \quad NDI \quad DNI \quad DIN$

De esas combinaciones existen tres que son estándares:

N I D	I N D	I D N
Preorden	Inorden	Postorden

Estas formas fueron elegidas en base a cómo se visita un nodo respecto a sus sub-árboles.

- **Preorden**: Primero se visita en nodo actual luego el sub-árbol izquierdo y luego el derecho.
- **Inorden**: Primero se visita el sub-árbol izquierdo, luego el nodo actual y por último el sub-árbol derecho.
- **Postorden**: Primero se visita el sub-árbol izquierdo, luego al sub-arbol derecho y por último al nodo actual.

3.3. Implementación

Si bien el tipo de dato abstracto árbol binario de por sí solo no es muy útil, el mismo se implementa en C de la siguiente forma:

```
typedef struct nodo_arbol* arbol_binario_t;

struct nodo_arbol{
        void * elemento;
        nodo_arbol * izquierda;
        nodo_arbol * derecha
};
```

4. Árbol Binario de Búsqueda

El tipo de dato abstracto árbol binario, por sí mismo carece de mucha utilidad, ya que en no tiene especificado ningún tipo de regla para poder insertar en él elementos.

Teniendo en cuenta que los árboles binarios están asociados al concepto de izquierda y derecha, se podrá diseñar un tipo de datos abstracto que permita realizar búsquedas cómo las de las búsquedas binarias y que además se pueda insertar y eliminar tan fácilmente como el una lista?

4.1. Definición y Operaciones

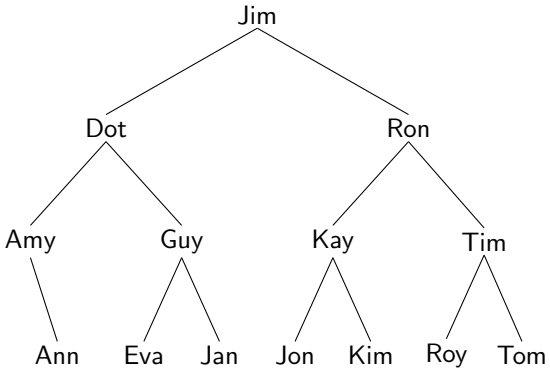
Los árboles de comparación permiten tener las ventajas de las listas enlazadas y además obtener la velocidad de la búsqueda binaria mediante el aprovechamiento de la estructura misma del árbol.

Básicamente un árbol de comparación permite saber que si uno se mueve al sub-árbol derecho sabe que se mueve a un sub-árbol con claves más grandes. Y si uno se mueve hacia el sub-árbol izquierdo el mismo almacenará valores de datos con claves más pequeñas [1, 2]. para ello a cada nodo del árbol se le asocia un valor único o clave.

Un **árbol binario de búsqueda** es un *árbol binario* que puede ser vacío o en cada nodo del mismo contener un **valor clave** que satisfaga las siguientes condiciones:

1. la clave en el nodo izquierdo del hijo (si existe) es menor que la clave en el nodo padre.
2. la clave en el nodo derecho del hijo (si existe) es mayor que la clave en el nodo padre
3. los arboles derecho e izquierdo son también arboles binarios de búsqueda

el siguiente árbol cumple con dicha especificación:



4.2. Operaciones

las operaciones básicas de un árbol binario son:

- crear
- destruir
- vacio
- insertar
- eliminar
- buscar
- recorrer

4.2.1. Búsqueda

La búsqueda de un elemento comienza en el nodo raíz y sigue estos pasos:

1. La clave buscada se compra con la clave del nodo raíz.
2. Si las claves son iguales, la búsqueda se detiene.
3. Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el sub-árbol derecho. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda en el sub-árbol izquierdo.

4.2.2. Insertar

1. Comparar la clave del elemento a insertar con la clave del nodo raíz, si es mayor avanzar hacia el sub-árbol derecho, si es menor hacia el izquierdo.
2. Repetir el paso anterior hasta encontrar un elemento con clave igual o llegar al final del sub-árbol donde debiera situarse el nuevo elemento.
3. Cuando se llega al final es porque no se ha encontrado, por tanto se deberá reservar memoria para una nueva estructura nodo, introducir en la parte reservada para los datos los valores del nuevo elemento y asignar nulo a los punteros izquierdo y derecho del mismo. A continuación se colocará el nuevo nodo como hijo izquierdo o derecho del anterior según sea el valor de su clave comparada con la de aquel.

4.2.3. Eliminar

La operación de eliminación de un nodo es también una extensión de la operación de búsqueda. El borrado o eliminación de un elemento requerirá, una vez buscado, considerar las siguientes posibilidades:

- Que no tenga hijos, sea hoja. Se suprime, asignando nulo al puntero de su antecesor que antes lo apuntaba a él.
- Que tenga un único hijo. El elemento anterior se enlaza con el hijo del que queremos borrar.
- Que tenga dos hijos. Se sustituye por el elemento más próximo en clave, inmediato superior o inmediato inferior. Para localizar estos elementos debe situarse en el hijo derecho del nodo a borrar y avanzar desde él siguiendo la rama izquierda de cada nodo hasta que a la izquierda ya no haya ningún nodo más, o bien, situarse en el hijo izquierdo y avanzar siguiendo siempre la rama derecha de cada nodo hasta llegar al final.

4.2.4. Recorrido

Estas formas fueron elegidas en base a cómo se visita un nodo respecto a sus sub-árboles.

- **Preorden**: Primero se visita en nodo actual luego el sub-árbol izquierdo y luego el derecho.
- **Inorden**: Primero se visita el sub-árbol izquierdo, luego el nodo actual y por último el sub-árbol derecho.
- **Postorden**: Primero se visita el sub-árbol izquierdo, luego al sub-árbol derecho y por último al nodo actual.

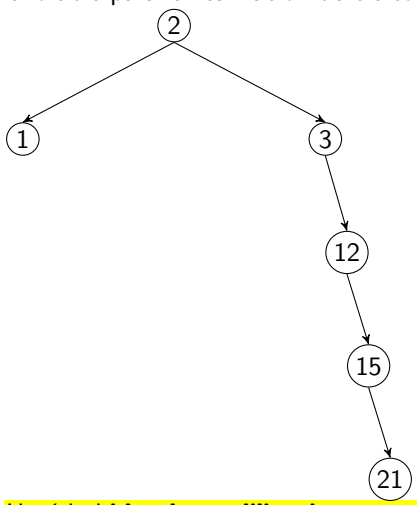
4.3. Implementación

```
typedef struct nodo_arbol* arbol_binario_t;

struct nodo_arbol{
        void * elemento;
        nodo_arbol * izquierda;
        nodo_arbol * derecha
};
```

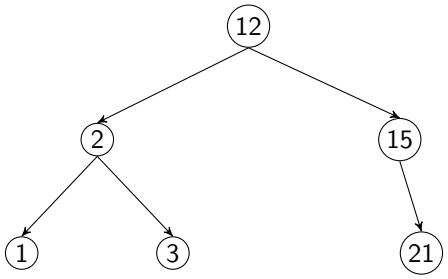
5. Árboles Binarios Equilibrados

Puede suceder que tras haber insertado y eliminado muchos elementos en un árbol binario de búsqueda, o también por la distribución de los datos mismos, tiendan a hacer que la estructura de un árbol binario de búsqueda quede desfavorable para la realización de ciertos algoritmos (búsqueda por ejemplo):



La búsqueda de un valor en el árbol puede no ser óptima si este se encuentra con esta estructura. Por ejemplo, si se desea buscar el item=1 la misma se realizaría en un tiempo que tiende a $O(log\ n)$ pero si se desea buscar el item=42, el tiempo de búsqueda tendería a $O(n)$. Ya que habría que moverse por el árbol siempre a derecha, debido a cómo está los datos dispuestos, hasta el último node para determinar que este no está en el árbol (pero de los casos posibles). Debe existir por ende una método de mantener el árbol con los elementos dispuestos de forma óptima para la búsqueda en él.

Un árbol **binario equilibrado** es aquel en el que la altura de los subárboles izquierdo y derecho de cualquier nodo nunca difiere en más de una unidad.



Para determinar si un árbol binario está equilibrado, se calcula su **factor de equilibrio**. El factor de equilibrio de un árbol binario es la **diferencia en altura entre los subárboles derecho e izquierdo**.

Si la altura del subárbol izquierdo es h_i y la altura del subárbol derecho como h_d , entonces el factor de equilibrio del árbol binario se determina por la siguiente fórmula:

$$Fe = h_d - h_i$$

Se debe recordar que la altura o height de un arbol binario n_i es el camino más largo desde n_i hasta un nodo hoja

5.1. Factor de Equilibrio

Cada nodo, además de la información que se pretende almacenar, debe tener los dos punteros a los árboles derecho e izquierdo, igual que los árboles binarios de búsqueda (ABB), y además el dato que controla el factor de equilibrio.

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

$$FE = altura\ subarbol\ derecho - altura\ subarbol\ izquierdo$$

5.2. AVL

Un árbol AVL o arbol de Adelson-Velsky and Landis es un árbol binario de búsqueda auto balanceado. En el cual cada hijo difiere en altura en un valor de -1,0,+1.

Por definición, para un árbol AVL, este valor debe ser -1, 0 ó 1. Si el factor de equilibrio de un nodo es:

- 0 → el nodo está equilibrado y sus subárboles tienen exactamente la misma altura.
- 1 → el nodo está equilibrado y su subárbol derecho es un nivel más alto.
- -1 → el nodo está equilibrado y su subárbol izquierdo es un nivel más alto.

Si el factor de equilibrio $|Fe| >= 2$ es necesario reequilibrar

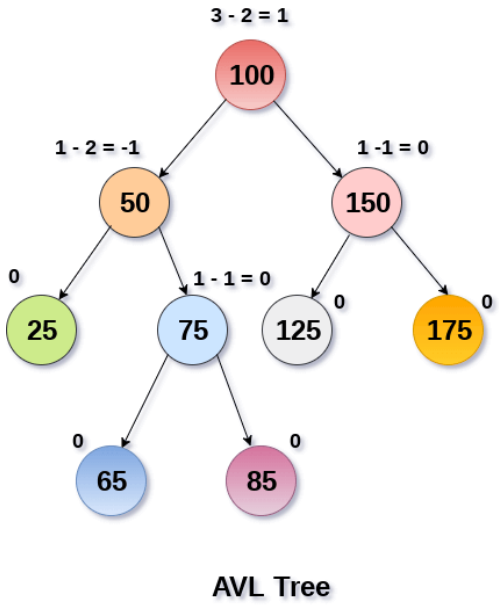


Figura 1

5.2.1. Implementación

Un nodo de un AVL debe poseer al menos los siguientes datos:

- Clave
- Factor de balanceo -1,0,1
- Un puntero a un AVL a derecha
- Un puntero a un AVL a izquierda

```
typedef struct avlnodo {
        int clave;
        int bal;                                    /* Factor de balance -1,0,1 */
        struct avlnodo *der, *izq;
} nodo, *pnodo;
```

5.2.2. Rotaciones

Cuando se inserta o borra un elemento en un AVL, las alturas de los nodos deben actualizarse y por ende utilizar operaciones de balanceo tienen que ser aplicadas. Estas operaciones se denominan **rotaciones**. Las rotaciones reorganizan la estructura del árbol después de cada inserción o borrado. Para ello se deben recalcular los **factores de equilibrio** de cada nodo, un costo asociado a tener en cuenta.

Existen dos tipos de rotaciones:

- Rotación Simple: Implica que el árbol reorganiza sus nodos hacia la izquierda o hacia la derecha
- Rotación Compuesta: Implica realizar dos rotaciones simples, izq-der o der-izq

Realizar una rotación a la izquierda y otra a la derecha en los nodos es una operación importante que se utiliza tanto en las operaciones de eliminación como de inserción.

Aquí hay una ilustración del proceso:

- Caso 1 **Rotación Simple a Derecha** (**Left left case**): Este caso ocurre cuando la altura del hijo izquierdo de un nodo es 2 mayor que la del hijo derecho, y el hijo izquierdo está equilibrado o pesa a la izquierda. Se puede arreglar usando una sola operación de rotación hacia la derecha en el nodo desequilibrado.

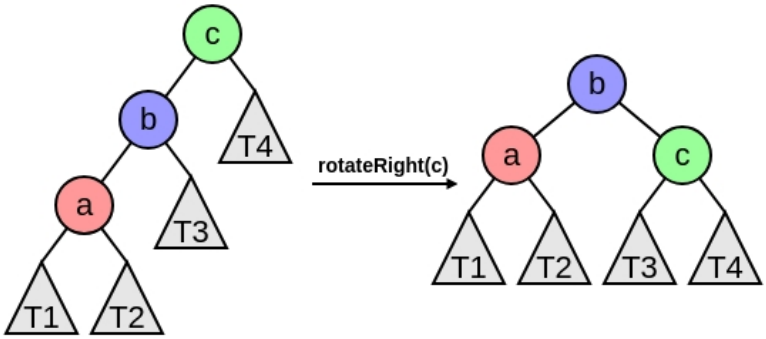
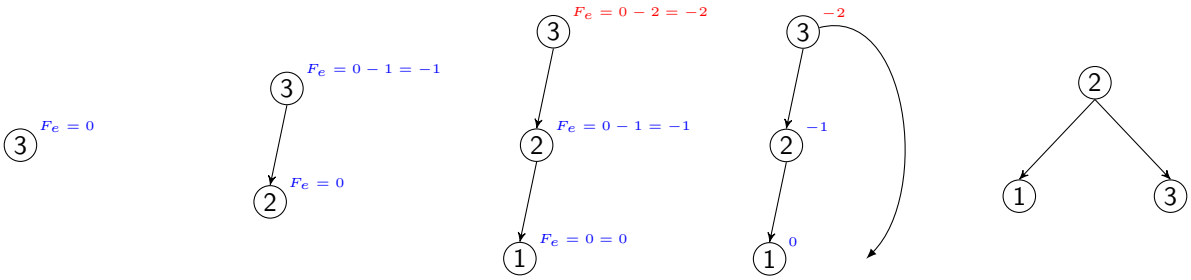
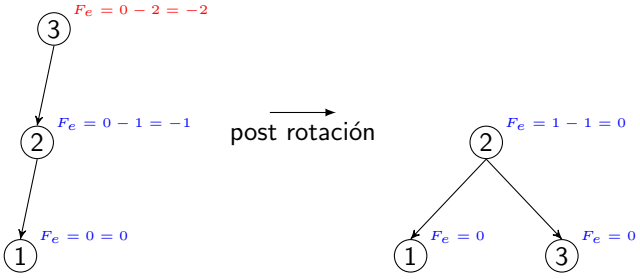


Figura 2

Supongase que se tienen que insertar 3 elementos en un arbol AVL vacio, los elementos son 3,2,1 en ese orden:



Si se recalcula el factor de equilibrio para cada nodo se obtiene, todos están dentro de los valores que mentienen la estructura del árbol balanceado $-1, 0, 1$:



- Caso 2 (**Right right case**): Este caso ocurre cuando la altura del hijo derecho de un nodo se vuelve 2 veces mayor que la del hijo izquierdo, y el hijo derecho es equilibrado o derecho-pesado. Este es el inverso del caso izquierdo izquierdo. Se puede arreglar usando una sola operación de rotación hacia la izquierda en el nodo desequilibrado.

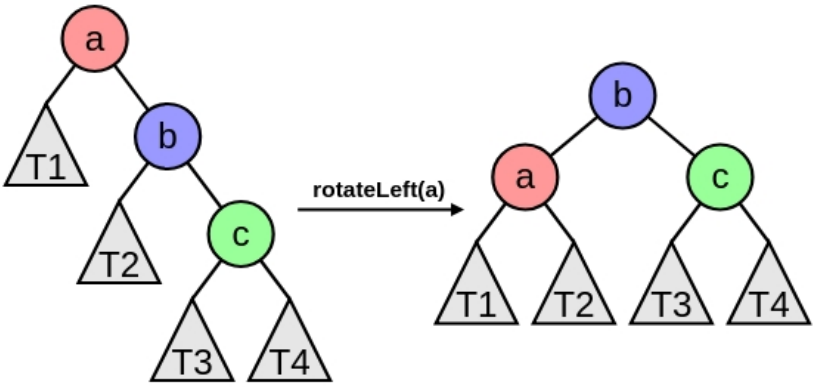
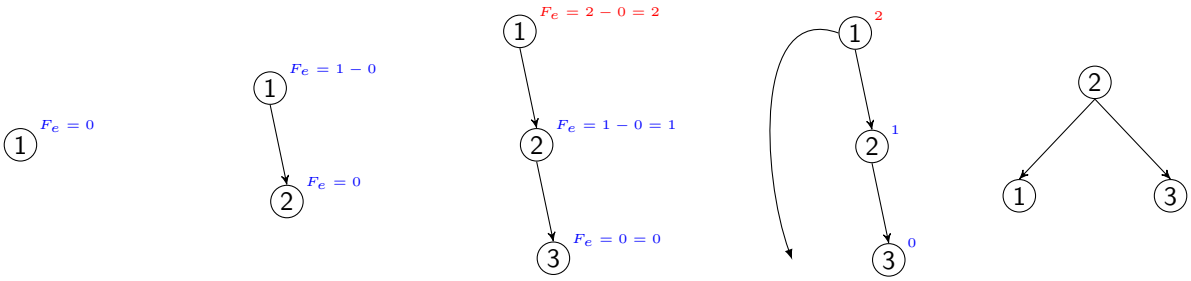
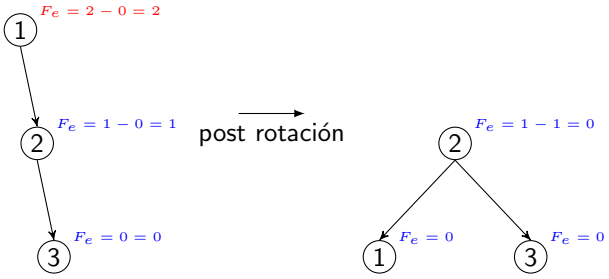


Figura 3

Este caso es muy parecido al anterior, ahora los elementos a insertar vienen en este orden 1,2,3:



Si se recalcula el factor de equilibrio para cada nodo se obtiene, todos están dentro de los valores que mantienen la estructura del árbol balanceado $-1, 0, 1$:



- Caso 3 **Rotación Izquierda - Derecha o ID** (**Left right case**) :Este caso ocurre cuando la altura del hijo izquierdo de un nodo es 2 mayor que la del hijo derecho, y el hijo izquierdo pesa a la derecha. Puede fijarse con un giro a la izquierda en el lado izquierdo, lo que da como resultado el caso izquierdo a la izquierda, que se fija con un giro a la derecha en el nodo desequilibrado.

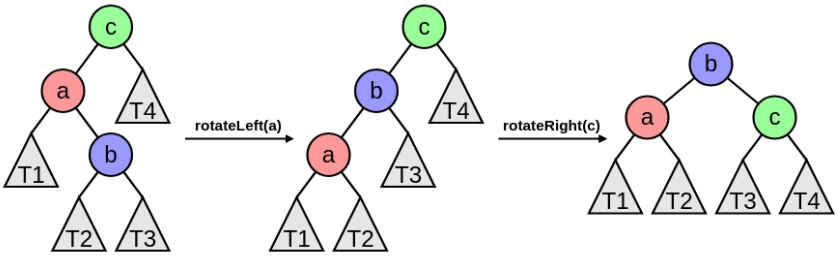
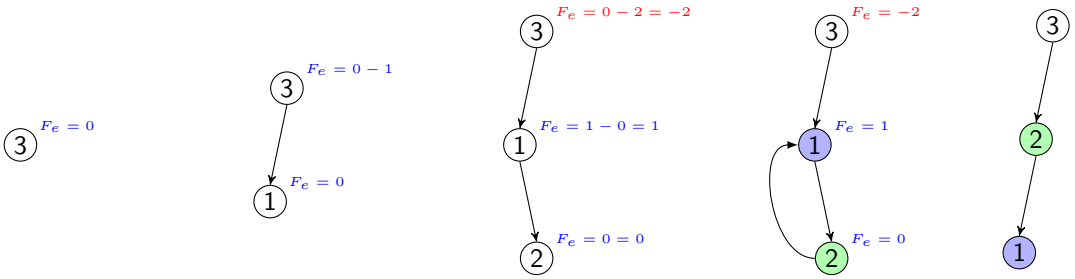
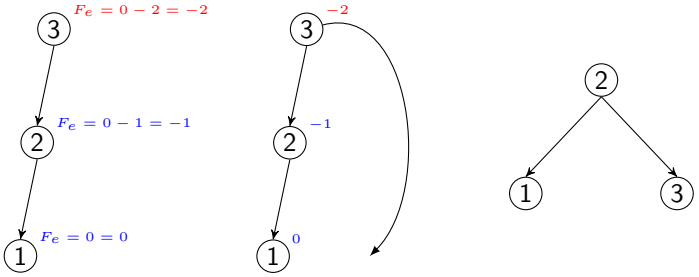


Figura 4

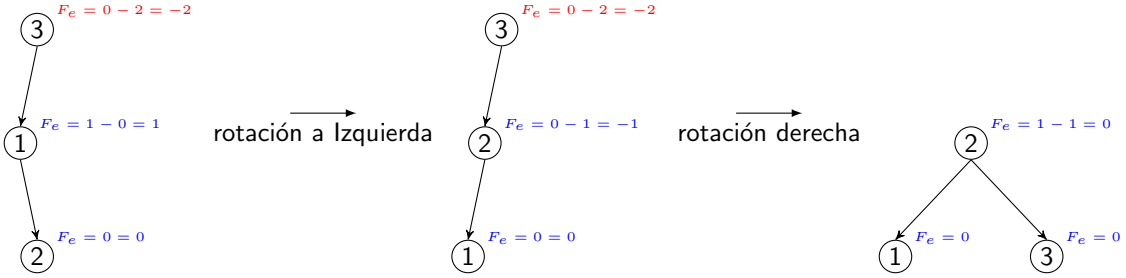
Ahora los elementos a insertar vienen en este orden 3,1,2:



esta rotación produce una situación de desbalanceo, nuevamente, que puede verse a simple vista pues se está en el caso 1, por ende se debe volver a aplicar una rotación **Simple a Derecha**:



Si se recalcula el factor de equilibrio para cada nodo se obtiene, todos están dentro de los valores que mantienen la estructura del árbol balanceado $-1, 0, 1$:



- Caso 4 Rotación Derecha - Izquierda o DI (Right left case): Este caso ocurre cuando la altura del hijo derecho de un nodo se vuelve 2 veces mayor que la del hijo izquierdo, y el hijo derecho pesa más en la izquierda. Este es el inverso del caso de la izquierda derecha. Se puede arreglar usando una rotación hacia la derecha en el lado derecho derecho, lo que da como resultado el caso derecho a la derecha que se fija mediante una rotación hacia la izquierda en el nodo no equilibrado.

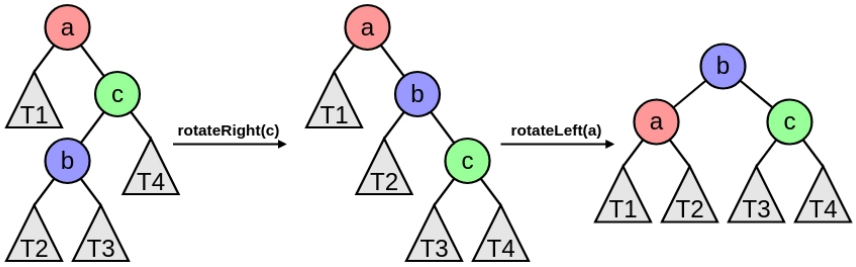
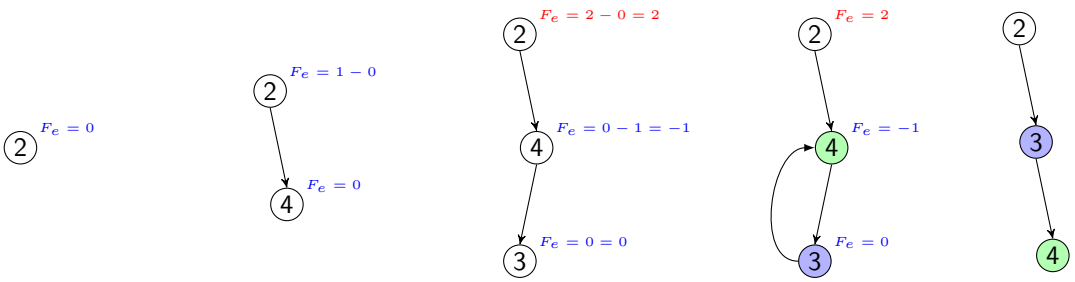
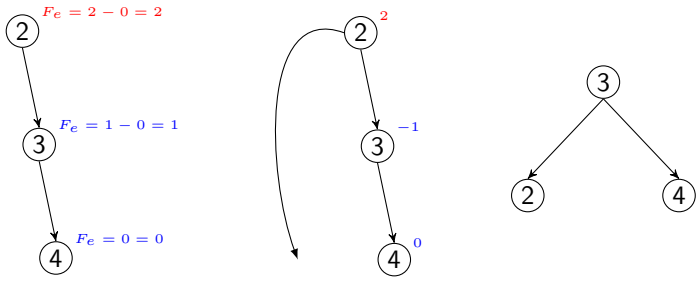


Figura 5

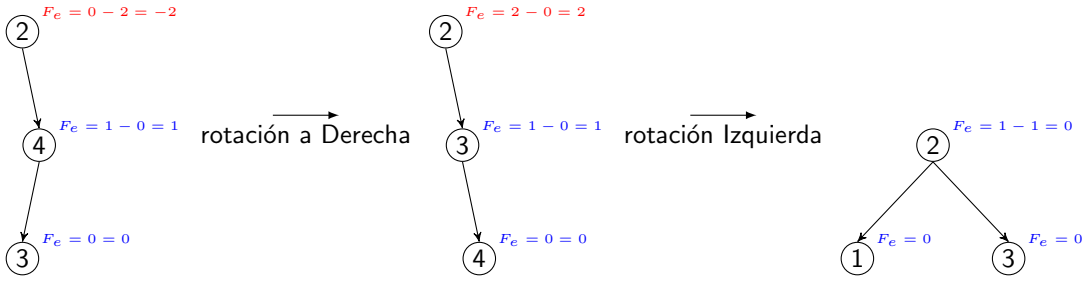
Ahora los elementos a insertar vienen en este orden 2,3,1:



esta rotación produce una situación de desbalanceo, que puede verse a simple vista pues se está en el caso 1, por ende se debe volver a aplicar una rotación Simple a Izquierda:



Si se recalcula el factor de equilibrio para cada nodo se obtiene, todos están dentro de los valores que mantienen la estructura del árbol balanceado $-1, 0, 1$:



5.2.3. Inserción

La inserción de un elemento en un árbol AVL utiliza el algoritmo usual de inserción de un nuevo elemento en un árbol binario modificado con la finalidad de conseguir que en ningún momento la altura de los subárboles izquierdo y derecho de un nodo difiera en más de una unidad. Para poder determinar ésto con facilidad, cada uno de los nodos de un AVL suele tener un campo donde almacenar su factor de equilibrio. El factor de equilibrio de un nodo es la diferencia entre las alturas de sus subárboles derecho e izquierdo y debe oscilar entre -1, 0 y 1, pues cualquier otro valor implicaría la necesaria reestructuración del árbol.

El proceso de inserción consistirá en:

- Comparar el elemento a insertar con el nodo raíz, si es mayor avanzar hacia el subárbol derecho, si es menor hacia el izquierdo y repetir la operación de comparación hasta encontrar un elemento igual o llegar al final del subárbol donde debiera estar el nuevo elemento. El camino recorrido desde la raíz hasta el momento en el que se terminan las comparaciones sin encontrar al elemento constituye el camino de búsqueda al que en reiteradas ocasiones se hará referencia.

- Cuando se llega al final es porque no se ha encontrado, entonces se crea un nuevo nodo donde se coloca el elemento y, tras asignarle un cero como factor de equilibrio, se añade como hijo izquierdo o derecho del nodo anterior según corresponda por la comparación de sus campos de clasificación. En este momento también se activará un interruptor sw para indicar que el subárbol ha crecido en altura.

- Regresar por el camino de búsqueda y si sw está activo calcular el nuevo factor de equilibrio del nodo que está siendo visitado. Deben distinguirse los siguientes casos:

  1. Las ramas izquierda y derecha del mencionado nodo tenían anteriormente la misma altura.
     Al insertar un elemento en la rama izquierda su altura se hará mayor que la de la derecha. Al insertar un elemento en la rama derecha ésta se hará más alta que la izquierda.
  2. La rama derecha era más alta que la izquierda.
     Un nuevo elemento en la rama izquierda consigue que las dos adquieran la misma altura. El subárbol ha dejado de crecer y sw se desactiva. Un nuevo elemento en la rama derecha rompe el equilibrio del árbol y hace que sea necesaria su reestructuración. La reestructuración anula el crecimiento en altura de la rama en la que se encuentra y, cuando se ejecuta, hay que conmutar el valor de la variable sw para que no se sigan recalculando factores de equilibrio.
  3. La rama izquierda era más alta que la derecha.
     Un nuevo elemento en la rama derecha consigue que las dos adquieran la misma altura. El subárbol ha dejado de crecer y sw se desactiva. Un nuevo elemento en la rama izquierda rompe el equilibrio del árbol y hace que sea necesaria su reestructuración. La restructuración anula el crecimiento en altura de la rama en la que se encuentra y, cuando se ejecuta, hay que conmutar el valor de la variable sw para que no se sigan recalculando factores de equilibrio.

5.2.4. Borrado

El borrado de un nodo en un árbol AVL consiste en la eliminación del mismo sin que el árbol deje de ser de búsqueda ni equilibrado. En principio el algoritmo de Eliminación en un AVL sigue casi los mismos pasos que el borrado en árboles de búsqueda binarios, la diferencia está en que, a las operaciones habituales, hay que añadir ahora las de cálculo de los factores de equilibrio y reestructuración del árbol (rotaciones de nodos simples, o dobles) cuando el equilibrio ha sido alterado. En un árbol AVL, la eliminación de un nodo implica la activación de una variable, sw , que, en este caso, indica ha disminuido la altura del subárbol considerado. Por tanto, una vez eliminado un nodo se activa sw y se regresa por el camino de búsqueda calculando, mientras sw esté activo, los nuevos factores de equilibrio (Fe) de los nodos visitados. Hay que tener en cuenta que, cuando se regresa por el camino de búsqueda con sw activo, el factor de equilibrio del nodo visitado disminuye en 1 si la eliminación se efectuó por la rama derecha y aumenta en 1 cuando se hizo por la izquierda. Si alguno de los nodos pierde la condición de equilibrio, ésta debe ser restaurada mediante rotaciones.

La variable sw en el borrado representa que decrece la rama que se esta considerando y por lo tanto sólo se desactiva cuando se verifique que la eliminación del nodo ha dejado de repercutir en la altura del subárbol. Así como en la inserción una vez efectuada una rotación sw siempre conmutaba y los restantes nodos mantenían su factor de equilibrio, en el borrado las rotaciones no siempre paran el proceso de actualización de los factores de equilibrio. Esto implica que puede producirse más de una rotación en el retroceso realizado por el camino de búsqueda hacia la raíz del árbol.

5.2.5. Arboles Rojo y Negro

Un árbol Rojo-Negro es un árbol binario de búsqueda auto-balanceado. Cada nodo de este árbol posee una información extra que es el color del nodo, este puede ser **rojo** o **negro**. La utilización de los colores es lo que hace que el árbol pueda quedar autobalanceado durante las inserciones o las eliminaciones de nodos.

El equilibrio se conserva pintando cada nodo del árbol con uno de los dos colores de manera que satisfaga ciertas propiedades, que en conjunto restringen el desequilibrio en que el árbol puede caer en el peor de los casos. Cuando se modifica el árbol, el nuevo árbol se reorganiza y se vuelve a pintar para restaurar las propiedades de coloración. Las propiedades están diseñadas de tal manera que esta reorganización y cambio de color se puede realizar de manera eficiente.

El equilibrio del árbol no es perfecto, pero es lo suficientemente bueno como para permitirle garantizar la búsqueda en el tiempo $O(logn)$, donde n es el número total de elementos en el árbol. Las operaciones de inserción y eliminación, junto con la reorganización del árbol y el cambio de color, también se realizan en tiempo $O(logn)$.

5.2.6. Definición

Un árbol Rojo-Negro deben satisfacer los requisitos de un Árbol Binario de Búsqueda y ademas :

- Todo nodo es o bien rojo o bien negro.
- La raíz es negra.
- Todas las hojas (NULL) son negras.
- Todo nodo rojo debe tener dos nodos hijos negros. No hay dos nodos rojos adyacentes.
- Cualquier camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.

Según esta definición:

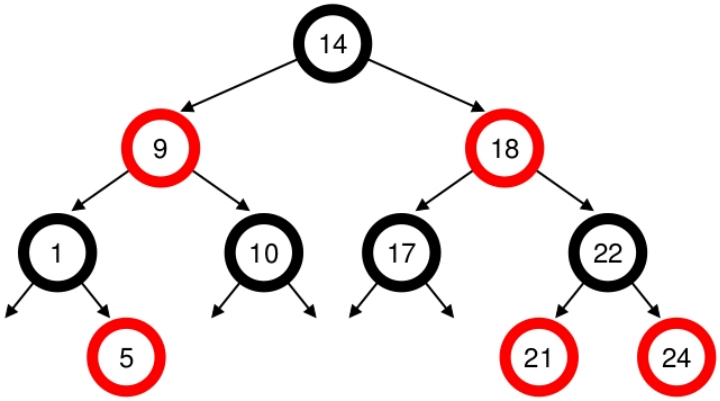


Figura 6: Ejemplo de Árbol Rojo-Negro

El número de nodos negros desde el nodo raíz a un nodo es denominada la profundidad negra del nodo. El número uniforme de nodos negros en todos los caminos desde la raíz hasta las hojas se denomina altura-negra. Estas dos propiedades hacen que:

> el camino más largo desde la raíz hasta una hoja no es más largo que dos veces el camino más corto desde la raíz a una hoja. El resultado es que dicho árbol está aproximadamente equilibrado.

Un árbol rojo-negro no n nodo internos tiene como altura como mucho $2log(n+1)$

5.2.7. Configuraciones Erróneas

Existen ciertas configuraciones que nunca pueden darse de un árbol Rojo Negro, estas son:

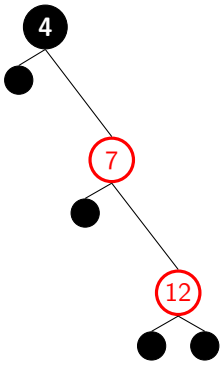


Figura 7: a

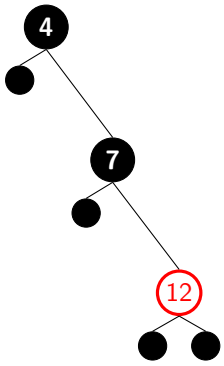


Figura 8: b

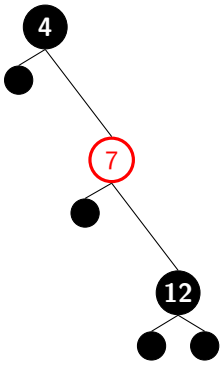


Figura 9: c

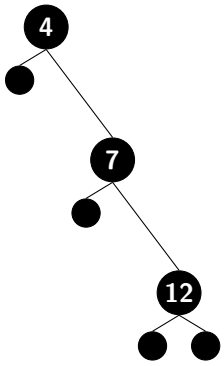


Figura 10: d

5.2.8. Implementación en C

El nodo del árbol rojo negro debe mantener la siguiente información:

- Color: ROJO o NEGRO
- Clave
- Un puntero a un árbol RN a izquierda
- Un puntero a un árbol RN a derecha
- Un puntero al nodo padre

```
typedef struct nn {                                    /* RB-Tree */
        int color;                                                  /* Solo puede ser Rojo o Neg
        int clave;                                                  /* clave entera */
        struct nn *left, *right, *padre;          /* tres punteros */
} nodo, *pnodo;
```

5.2.9. Rotaciones

Para conservar las propiedades que debe cumplir todo árbol rojo-negro, en ciertos casos de la inserción y la eliminación será necesario reestructurar el árbol, si bien no debe perderse la ordenación relativa de los nodos. Para ello, se llevan a cabo una o varias rotaciones, que no son más que reestructuraciones en las relaciones padre-hijo-tío-nieto.

5.2.10. Inserción

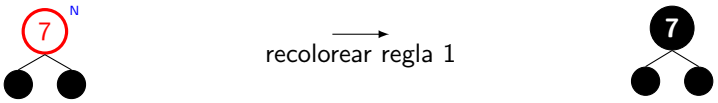
La inserción comienza **añadiendo el nodo como se haría en un árbol binario de búsqueda convencional y pintándolo de roj**o. Lo que sucede después depende del color de otros nodos cercanos. El término tío nodo será usado para referenciar al hermano del padre de un nodo, como en los árboles familiares humanos. Conviene notar que:

La propiedad 3 (Todas las hojas, incluyendo las nulas, son negras) siempre se cumple. La propiedad 4 (Ambos hijos de cada nodo rojo son negros) está amenazada solo por añadir un nodo rojo, por repintar un nodo negro de color rojo o por una rotación. La propiedad 5 (Todos los caminos desde un nodo dado hasta sus nodos hojas contiene el mismo número de nodos negros) está amenazada solo por repintar un nodo negro de color rojo o por una rotación. Al contrario de lo que sucede en otros árboles como puede ser el Árbol AVL, en cada inserción se realiza un máximo de una rotación, ya sea simple o doble. Por otra parte, se asegura un tiempo de recoloración máximo de $O(\log_2 n)$ por cada inserción.

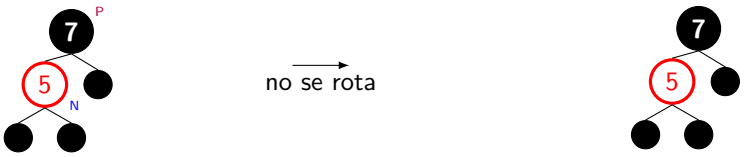
- N será utilizada por el nodo que está siendo insertado
- P para los padres del nodo N
- A para los abuelos del nodo N
- T para los tíos del nodo N

Notamos que los roles y etiquetas de los nodos están intercambiados entre algunos casos, pero en cada caso, toda etiqueta continúa representando el mismo nodo que representaba al comienzo del caso. Cualquier color mostrado en el diagrama está o bien supuesto en el caso o implicado por dichas suposiciones.

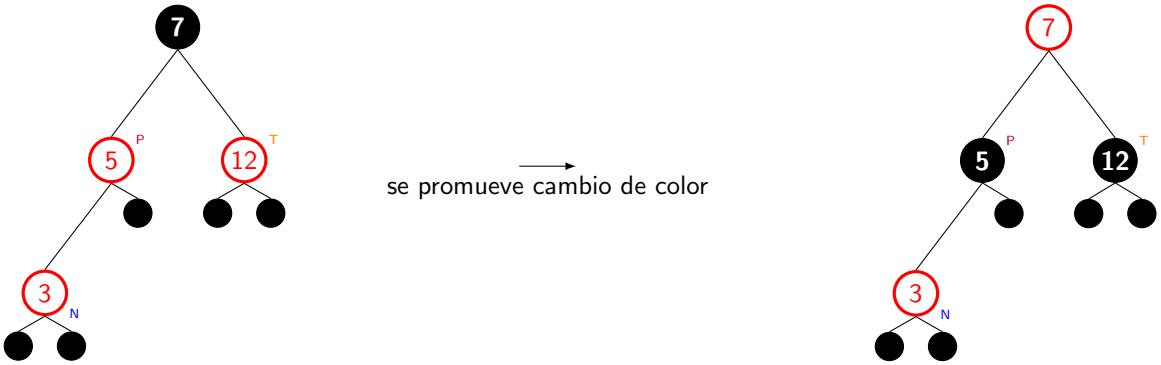
- **Caso 1**: El nuevo nodo N es la raíz del árbol. En este caso, es repintado en color negro para satisfacer la propiedad 2 (la raíz es negra). Como esto añade un nodo negro a cada camino, la propiedad 5 (todos los caminos desde un nodo dado a sus hojas contiene el mismo número de nodos negros) se mantiene:



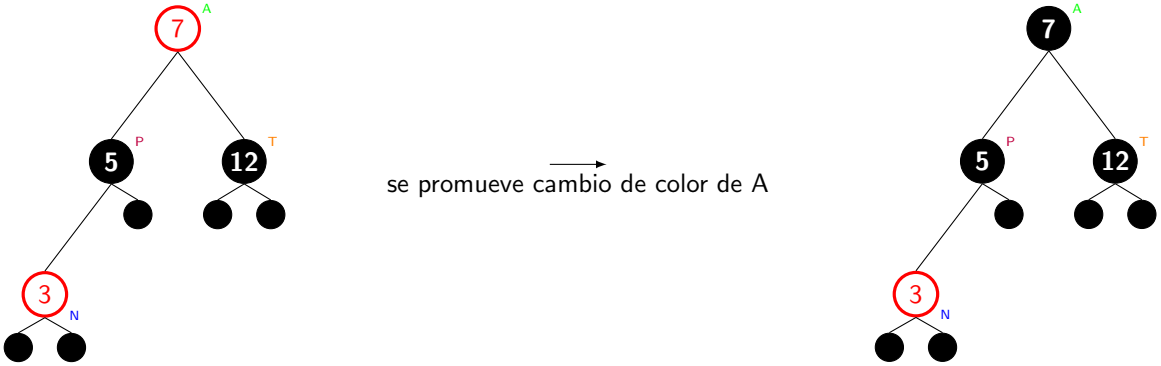
- **Caso 2**: El padre del nuevo nodo (esto es, el nodo P) es negro, así que la propiedad 4 (ambos hijos de cada nodo rojo son negros) se mantiene. En este caso, el árbol es aun válido. La propiedad 5 (todos los caminos desde cualquier nodo dado a sus hojas contiene igual número de nodos negros) se mantiene, porque el nuevo nodo N tiene dos hojas negras como hijos, pero como N es rojo, los caminos a través de cada uno de sus hijos tienen el mismo número de nodos negros que el camino hasta la hoja que reemplazó, que era negra, y así esta propiedad se mantiene satisfecha.



- **Caso 3**: Si el padre P y el tío T son rojos, entonces ambos nodos pueden ser repintados de negro y el abuelo A se convierte en rojo para mantener la propiedad 5 (todos los caminos desde cualquier nodo dado hasta sus hojas contiene el mismo número de nodos negros). Ahora, el nuevo nodo rojo N tiene un padre negro. Como cualquier camino a través del padre o el tío debe pasar a través del abuelo, el número de nodos negros en esos caminos no ha cambiado.

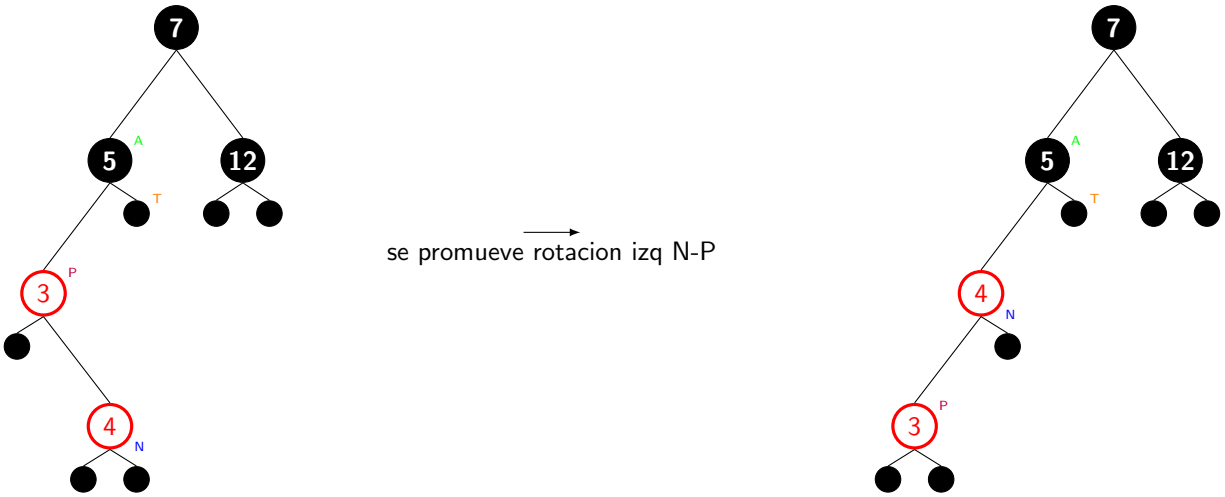


Sin embargo, el abuelo A ahora viola la propiedad 2 **(la raíz es negra**) o la 4 **(ambos hijos de cada nodo rojo son negros**), en el caso de la 4 porque A podría tener un padre rojo. Para solucionar este problema, el procedimiento completo se realizará de forma recursiva hacia arriba hasta alcanzar el caso 1.



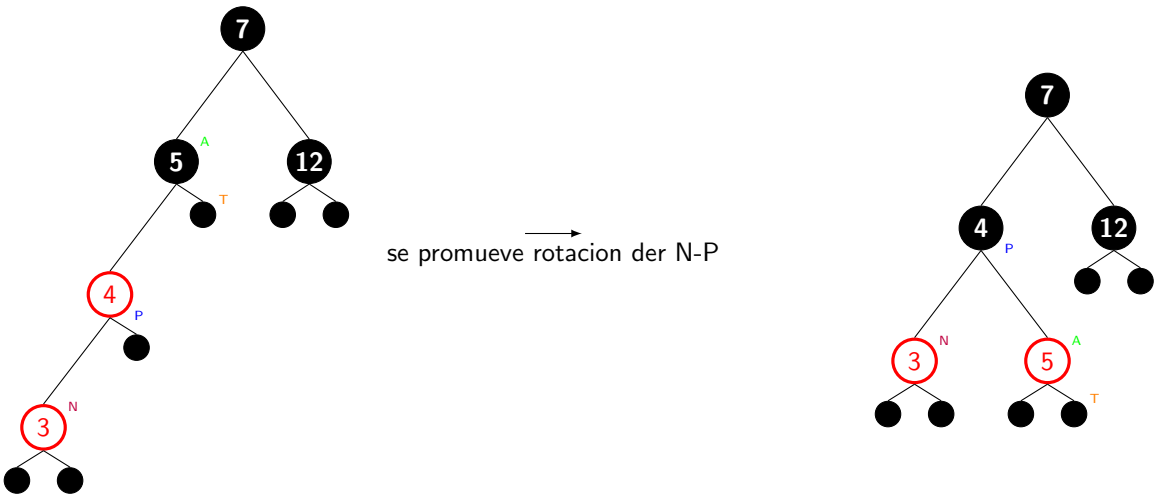
Nota: pasa lo mismo si N estahijo a derecha de P

- **Caso 4**: El nodo padre P es rojo pero el tío T es negro; también, el nuevo nodo N es el hijo derecho de P, y P es el hijo izquierdo de su padre A. En este caso, **una rotación a la izquierda** que cambia los roles del nuevo nodo N y su padre P puede ser realizada; entonces, el primer nodo padre P se ve implicado al usar el caso 5 de inserción (re etiquetando N y P) debido a que la propiedad 4 (ambos hijos de cada nodo rojo son negros) se mantiene aún incumplida.



La rotación causa que algunos caminos pasen a través del nuevo nodo donde no lo hacían antes, pero ambos nodos son rojos, así que la propiedad 5 (todos los caminos desde cualquier nodo dado a sus hojas contiene el mismo número de nodos negros) no es violada por la rotación, después de completado este caso, se puede notar que aún se incumple la propiedad número 4 (ambos hijos de cada nodo rojo son de color negro), esto se resuelve pasando al caso 5.

- **Caso 5**: El padre P es rojo pero el tío T es negro, el nuevo nodo N es el hijo izquierdo de P, y P es el hijo izquierdo de su padre A. En este caso, **se realiza una rotación a la derecha sobre el padre P**:



el resultado es un árbol donde el padre P es ahora el padre del nuevo nodo N y del inicial abuelo A. Este nodo A ha de ser negro, así como su hijo P rojo. Se intercambian los colores de ambos y el resultado satisface la propiedad 4 (ambos hijos de un nodo rojo son negros). La propiedad 5 (todos los caminos desde un nodo dado hasta sus hojas contienen el mismo número de nodos negros) también se mantiene satisfecha, ya que todos los caminos que iban a través de esos tres nodos entraban por A antes, y ahora entran por P. En cada caso, este es el único nodo negro de los tres.

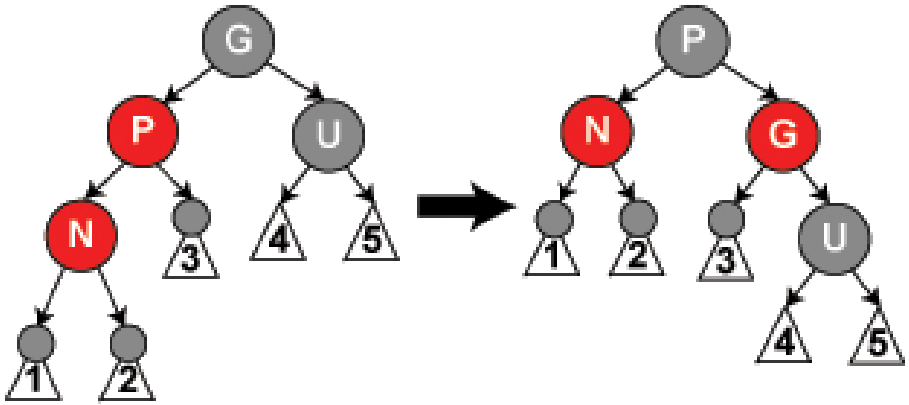


Figura 11: Ejemplo de Árbol Rojo-Negro

5.2.11. Arboles Splay o Biselados

6. Familia de Árboles B

Los árboles B de búsqueda nacen a partir de la necesidad de tener un número muy grande de elementos en los cuales hay que realizar dicha operación, con el agravante que este número tan grande de elementos no entra completamente en memoria. El ejemplo clásico que se encuentra en muchos libros es el de los clientes de un banco, que sin ninguna duda podrían rondar el millón de individuos.

Los creadores del árbol B, Rudolf Bayer y Ed McCreight, no han explicado el significado de la letra B de su nombre. Se cree que la B es de balanceado, dado que todos los nodos hoja se mantienen al mismo nivel en el árbol. La B también puede referirse a Bayer, o a Boeing, porque sus creadores trabajaban en los Boeing Scientific Research Labs por ese entonces.

6.0.1. Definición

Los árboles B son árboles de orden M, M>2, equilibrados, de **Búsqueda** y **Mínima Altura**, propuestos por Bayer y McCreight que han de cumplir las siguientes características:

- El nodo raíz tiene entre 2 y M ramas descendientes.
- Todos los nodos (excepto la raíz) tienen entre (M + 1) división entera entre 2 y M ramas descendientes.
- Todos los nodos (excepto la raiz) tienen entre (M - 1) división entera entre 2 y (M - 1) claves.

- El número de claves en cada nodo es siempre una unidad menor que el número de sus ramas.
- Todas las ramas que parten de un determinado nodo tienen exactamente la misma altura.
- En los nodos las claves se encuentran clasificadas y además, a su vez, clasifican las claves almacenadas en los nodos descendientes. Es costumbre denominar a los nodos de un árbol B, páginas. La estructura de una página de un árbol B de orden 5 puede representarse como muestra la Figura

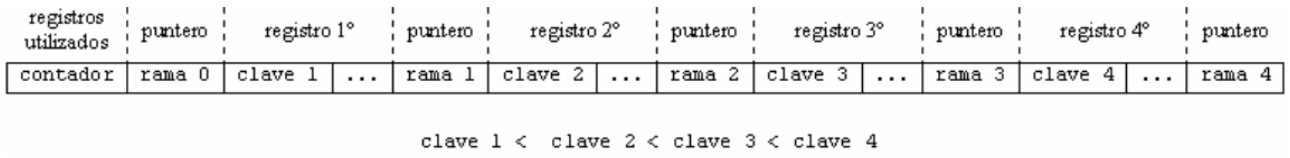


Figura 12: Ejemplo de Nodo de Árbol B

Si la rama 0 no está vacia, las claves situadas en el nodo apuntado por la rama 0 serán menores que la clave 1 del nodo actual y las claves situadas en nodo apuntado por rama 1 mayores que ella. Esta relación se repite para las restantes claves de los registros utilizados del nodo actual respecto a las almacenadas en sus correspondientes ramas izquierdas.

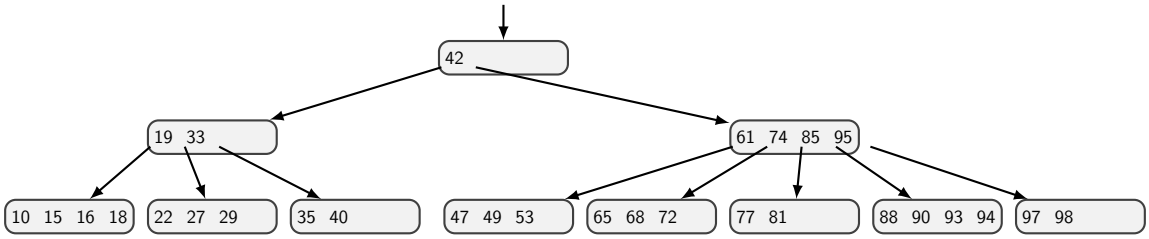
Como puede verse cada nodo posee la siguiente información:

- múltiples cantidad de registros (según su orden),
- punteros a otros nodos,
- el contador de registros utilizados.

6.1. Búsqueda

La búsqueda en un árbol B se realiza de la siguiente forma, sea la clave a buscar K_s:

1. Se comienza por el nodo (en este tipo de árboles se denomina página) raíz. Si este es NULL, el árbol esta vacío y se termina la búsqueda,
2. Si no es NULL se recorren las claves K_i de esa página, si se halla una clave que cumpla $K_i = K_s$, se encontró la clave.
3. Si no se encontró la clave en esa página, entonces se recorre la pagina correspondiente según:
   a) Si $K_s < K_1$ se continua por la $Rama[0]$
   b) si $K_{p-1} < K_s < Key_p$ por la $Rama[p-1]$, p puede ser cualquier numero entre 2 y el contador de registros.
   c) Si $K_s > cont$ se continua por la $Rama[Cont]$
4. Las operaciones descritas se repiten con la nueva página hasta que la clave se encuentre o el puntero a la página sea NULL y, por tanto, se pueda dar por descartado el encontrarla.



6.2. Inserción

Un árbol B es una estructura que crece de abajo hacia arriba, es decir desde las hojas hacia la raíz. El algoritmo de Inserción en un árbol B, sigue los siguientes pasos:

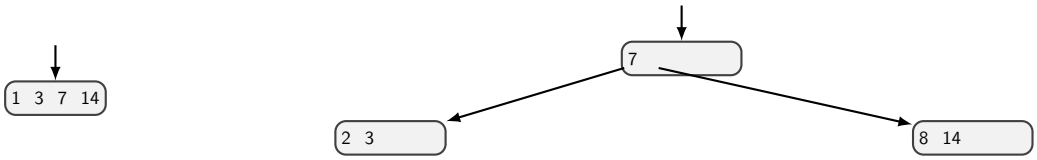
1. Buscar el el árbol B la hoja nodo donde el nuevo valor clave debería ser insertado.

2. si el nodo hoja (o pagina) no está completa, esto es, contiene menos de $m-1$ claves, entonces insertar el nuevo valor clave manteniendo el orden de los elementos en el nodo hoja.
3. Si el nodo hoja esta lleno, esto es, el nodo hoja contiene $m-1$ claves, entonces:
   a) Insertar el nuevo valor de clave en el conjunto existente de claves,
   b) dividir el nodo en dos en su mediana (representa el valor de la variable de posición central en un conjunto de datos ordenados) notando que los nodos partidos están completos a la mitad, y
   c) empujar el elemento que corresponde a la mediana hacia el nodo padre que está arriba. Si el nodo padre también está lleno, entonces separar el nodo padre endos nuevos nodos y seguir los pasos anteriores.

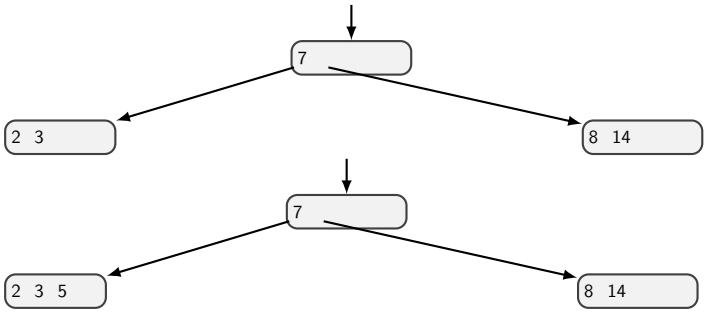
Para ver las reglas anteriores se creara un arbol B de orden 5 insertando los siguientes elementos: 3,14,7,1,8,5,1,17,13,6,23,12,20,2
y 19

La inserción de los **elementos 3,14,7,1** corresponden al Caso 2 del algoritmo de inserción:

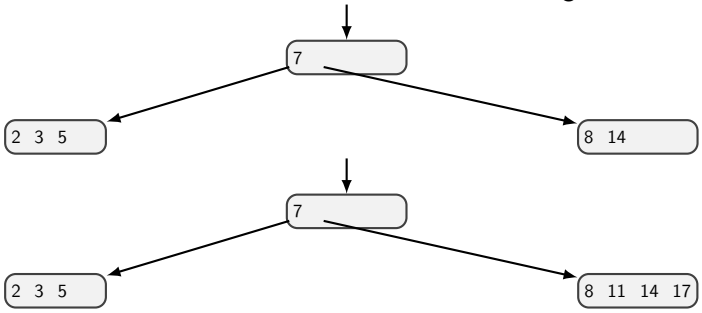
Al agregar el **elemento 8**, se plantea el caso 3 de la inserción, con el nodo completo, por lo cual se deben crear dos nuevos nodos y promover al padre al nodo central:



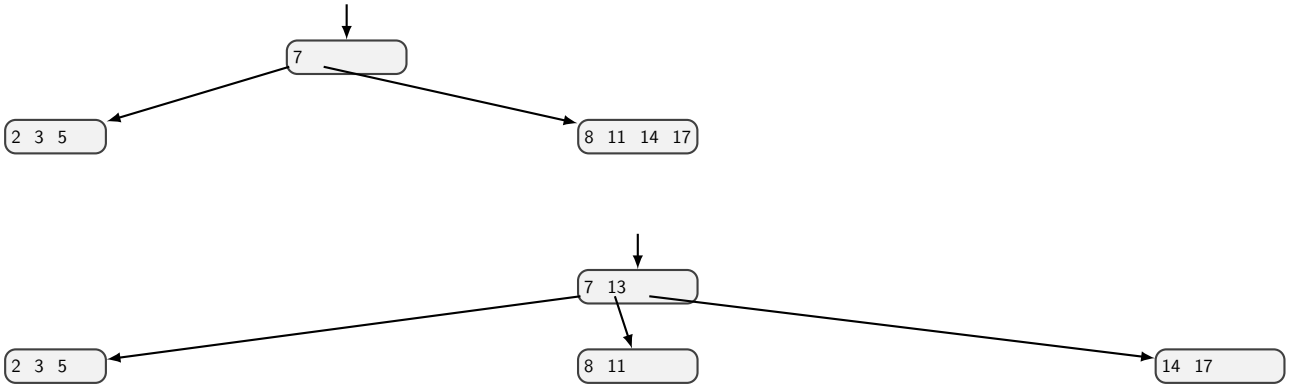
A continuación se insertan los elementos 5 según el Caso 2.



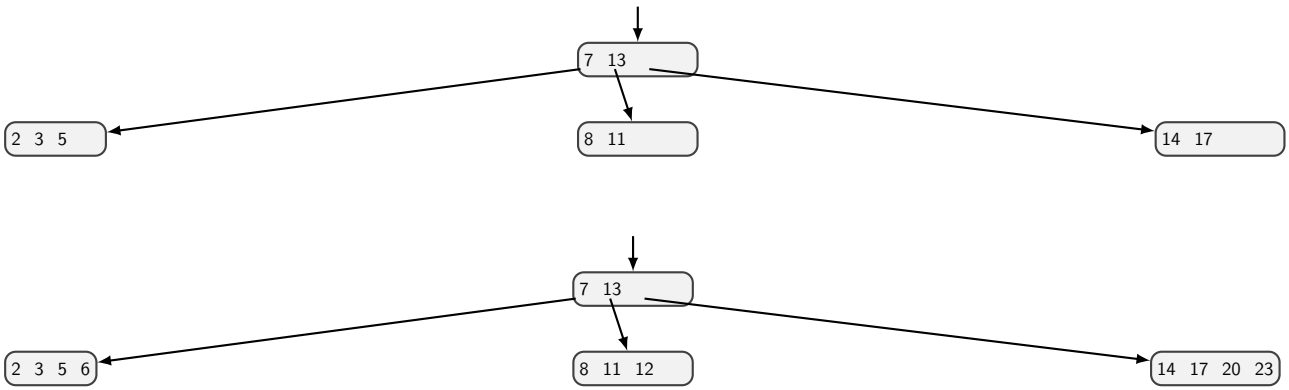
A continuación se insertan los elementos 11,17, según el Caso 2.



A continuación se inserta el elemento 13, a través de la aplicación del caso 3:



A continuación se insertaran los **elementos 6,23,12,10** todos aplicando el Caso 2:



¿Cómo quedaría al insertar la clave 26 ?

6.3. Eliminación

Al igual que la inserción la eliminación de elementos se realiza desde los nodos hojas. Existen dos casos de borrado. En el primer caso se debe eliminar en un nodo hoja. En el segundo de los casos, se debe eliminar en un nodo interno.

6.3.1. Caso 1: Eliminación de un Nodo Hoja

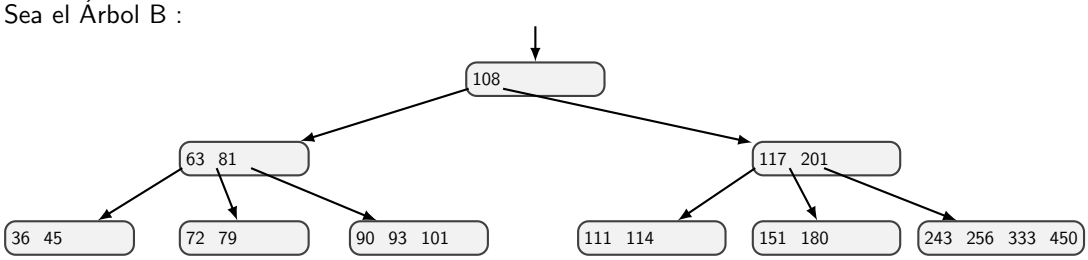
1. Localizar el nodo hoja en el que se debe eliminar.
2. Si el nodo hoja contiene más claves del mínimo número de claves (más de $m/2$ elementos), entonces se elimina el valor.
3. Sino, si el nodo hoja no contiene $m/2$ elementos, entonces completar el nodo tomando un elemento del hermano de izquierda o de derecha
   a) Si el hermano de izquierda tiene más elementos que el mínimo numero de valores, subir su clave más grande y bajar el elemento interviniente desde el nodo padre al nodo hoja donde la clave fue eliminada.
   b) Sino, si el hermano derecho tiene más que el minimo numero del valores de claves, subir su clave más pequeña a su nodo padre y bajar la el elemento interviniente desde el padre al nodo en que fue eliminado el valor de la clave.
4. Sino, si ambos nodos hermanos, derecho e izquierdo,contienen el mínimo número de elementos , entonces crear un nuevo nodo combinando los dos nodos y el elemento interviniente del nodo padre (asegurando que el numero de elementos no supere el máximo numero de elementos que se puede tener que es m). Si bajando el elemento interviente del nodo padre, lo deja con menos elementos que el minimo por nodo, entonces propagar el proceso hacia arriba ,así se reduce la altura del arbol.

6.3.2. Caso 2: Eliminación de un Nodo Interno

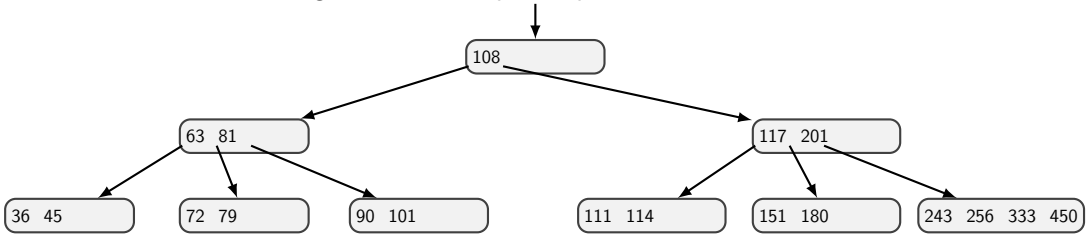
Para borrar un nodo interno, promover el sucesor o predecesor de la clave a ser eliminada para que ocupe la posición de la clave eliminada. Este sucesor o predecesor debe estas siempre en el nodo hoja.

6.4. Ejemplo

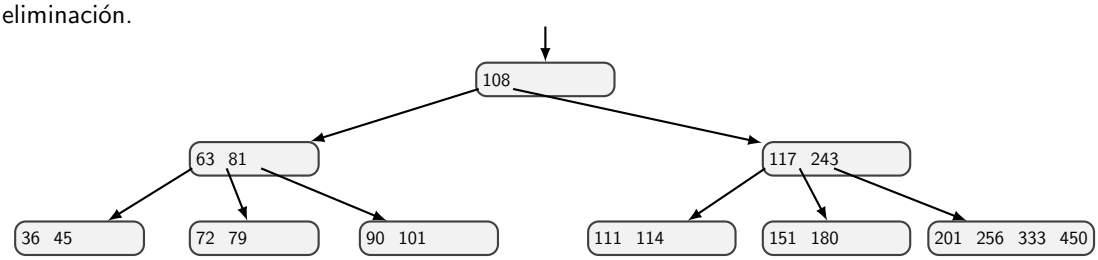
Sea el Árbol B :



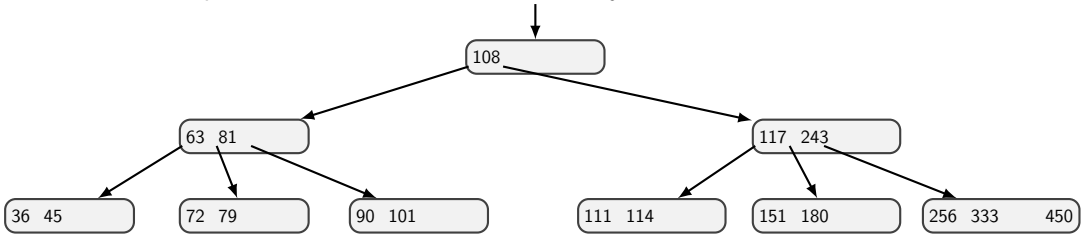
Eliminar el 93 Eliminar los siguientes valores paso a paso, 93



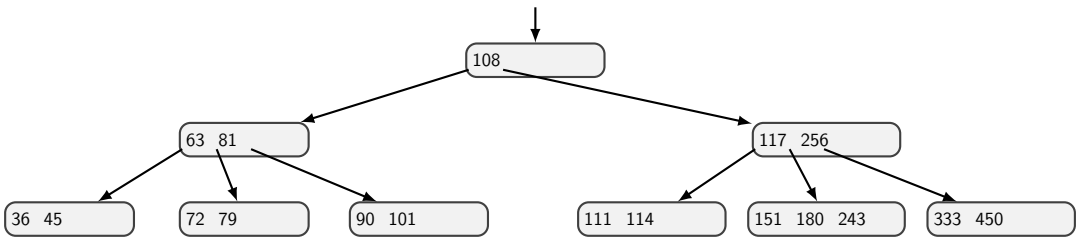
Eliminar el 201 Se elimina el 201, para eso pasa intercambiamos el 201 y el 243 (podría haber sido la opción con el otro nodo hoja) con lo cual, ya que esta en un nodo no terminal, ahora lo eliminamos del nodo hoja lo que facilita la eliminación.

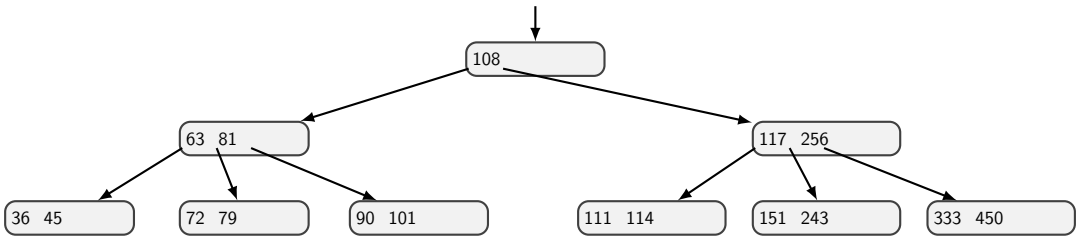


Eliminar el 201 se procede a la eliminación del nodo hoja donde se encuentra el 201:

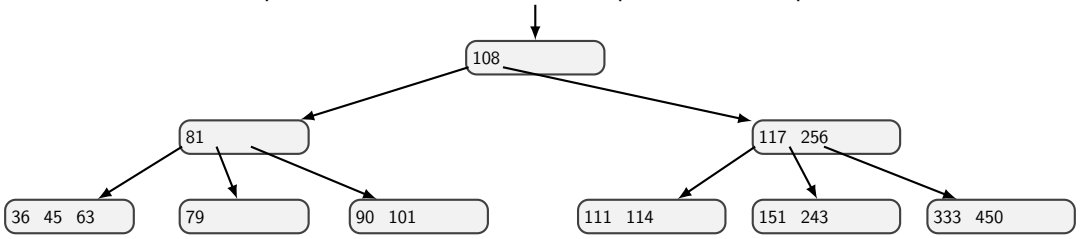


Eliminar el 180 Se elimina el 180, siempre se comienza por una hoja, en este caso está bien. Pero si se elimina sin más la clave , la hoja queda desbalanceada. por ende, se debe bajar el el 243 a la hoja en el que se encuentra el 180 y promover al 256 al nodo padre.

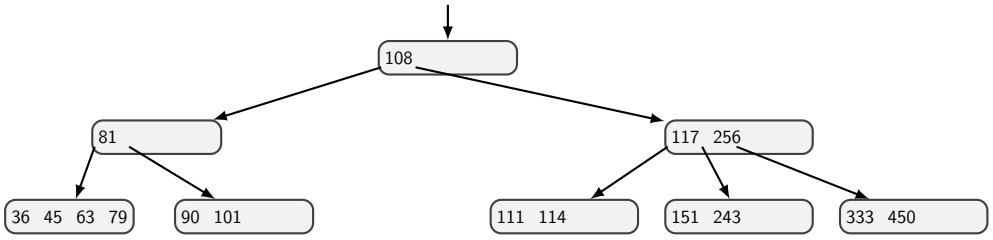




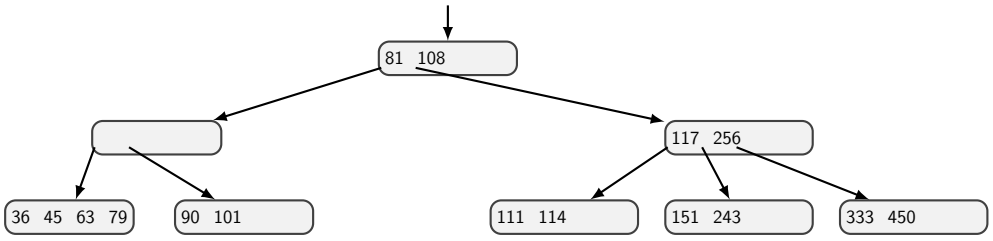
Eliminar el 72 Ahora se pasa a eliminar el elemento 72, para ello debo aplicar el caso 4:



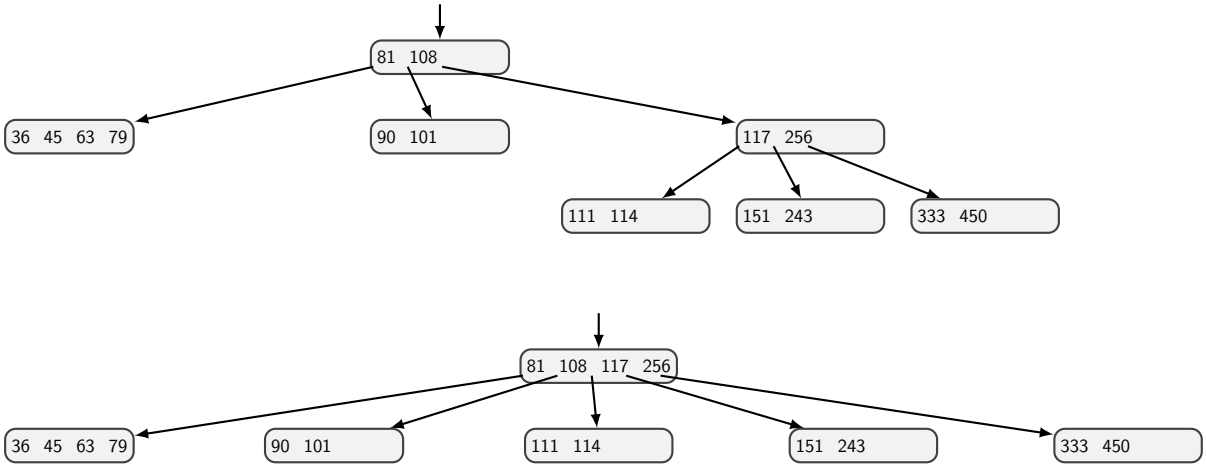
Al quedar el árbol des-balanceado se debe pasar el 79 al nodo hoja anterior, ademas se debe eliminar ese nodo hoja:



Pero ahora el nodo que contiene al elemento 81 esta des-balanceado respecto de $m/2$, por lo cual hay que promoverlo hacia el nodo raíz:



Al haber promovido el 81 al raiz ahora hay que reorganizar el nodo raiz pues de tener 2 punteros ahora tiene que tener 3 punteros, como mínimo. Pero si se promueve el nodo hoja de los elementos que contiene a las claves 117 y 256 el nodo raiz queda completo, con lo cual esta es la solucion requerida:



Referencias

[1] Robert Kruse, CL Tondo, et al. *Data structures and program design in C*. Pearson Education India, 2007.

[2] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 199.