Árboles Autoajustables - Splay Tree

Año 2026

Introducción

Un *árbol autoajustable* es una estructura de datos que se reestructura para obtener mayor eficiencia en sus operaciones. En la asignatura se han visto algunos árboles autoajustables, como lo son el AVL y la parva, ambos se modifican para mantener una determinada propiedad. El AVL recurre a las rotaciones para satisfacer una restricción en su altura que le asegura permanecer balanceado, logrando así mantener los costos máximos de cualquier operación con una cota del $O(\log n)$. Los intercambios que se realizan en la parva le permiten cumplir con un requisito de conformación, que establece que "*todo padre debe ser menor (mayor) que sus hijos*", logrando que el mínimo (máximo) del conjunto esté siempre en la raíz optimizando el acceso al mismo, que también es de $O(\log n)$. Estas estructuras se basan en la isoprobabilidad de consulta para asegurar su eficiencia, pero hay casos en los que esta distribución de probabilidades no forma parte del problema a resolver.

Consideremos el problema de mantener el registro con las historias clínicas de pacientes en un hospital. Los registros de los pacientes que están internados en el hospital son más activos que los registros de pacientes que no lo están, ya que los primeros son continuamente actualizados por doctores y enfermeros. Cuando los pacientes dejan el hospital, sus registros se vuelven mucho menos activos, siendo esporádica su consulta. Si en algún momento el paciente es readmitido en el hospital, su registro se vuelve activo nuevamente. Más aún, si la readmisión del paciente es una emergencia, su registro, a pesar de ser poco activo, debería estar disponible para ser consultado sucesivamente en forma rápida.

Si nos planteáramos usar un Árbol Binario de Búsqueda común, o un AVL, para almacenar la información de los pacientes, los registros de los pacientes recientemente ingresados al hospital serían dados de alta en las hojas y por lo tanto el acceso a ellos sería costoso; además, si es esperable que los mismos sean accedidos con mayor frecuencia, estos accesos muy caros degradarían el desempeño de la estructura. Lo que deseamos es que éstos registros, o los que son accedidos con mayor frecuencia, estén cercanos a la raíz para que el acceso a los mismos sea menos costoso. Esto podría lograrse en un ABB colocando aquellos elementos cuya probabilidad de ser consultados es mayor, en los niveles más cercanos a la raíz. Para eso se necesitaría conocer de antemano esa probabilidad, además la misma NO debería cambiar una vez establecida. Estas no serían características presentes en el problema de registros médicos planteado.

Por lo tanto, se necesita una estructura de datos que se adapte a problemas en los cuales los elementos de la relación *no* tienen la misma probabilidad de ser consultados, y esa probabilidad *no* es conocida, o incluso la misma *cambia* a través del tiempo; o problemas en los que se supone que si un elemento es consultado, es probable que tanto él, como elementos cercanos al mismo, sean consultados nuevamente o con mayor frecuencia. Sería muy conveniente que dicha estructura cambie su forma automáticamente, que se reestructure luego de cada acceso a un nodo para llevarlo cerca de la raíz, bajo la suposición de que los nodos recientemente accedidos serán consultados nuevamente, dejando a los que son menos accedidos más cercanos a las hojas, disminuyendo así los costos de acceso. Entonces el objetivo es idear una forma sencilla de reestructurar el árbol después de cada acceso, de manera que mueva el elemento accedido más cerca de la raíz.

Árboles Splay

Esta estructura logra el objetivo planteado, resolviendo de forma eficiente problemas similares al presentado más arriba. El ***Árbol Splay***[1] es una estructura propuesta por Daniel Sleator y Robert Tarjan en 1985; se trata de una versión *autoajustable* del Árbol Binario de Búsqueda (ABB), que se ***reestructura*** cada vez que se accede a un nodo, ya sea para una evocación, una inserción o una supresión. Esta reestructuración consiste en mover el nodo consultado hacia la raíz del árbol. Por esta razón, los árboles *Splay* aseguran que los nodos consultados recientemente pueden ser accedidos nuevamente en forma rápida. A diferencia de otras implementaciones eficientes de estas estructuras de datos, los árboles autoajustables no tienen que mantener ninguna condición de balance, por lo

[1] En la literatura se puede encontrar también como árbol: *desplegable, biselado, desdoblado, extendido*.

que no requieren el mantenimiento de información sobre su altura o su equilibrio, con lo que, en cierta medida, se ahorra espacio y se simplifica el código. No son estructuras "equilibradas" en ningún sentido, no deben verificar alguna condición de este tipo luego de cada operación, como es el caso de los AVL; su altura puede llegar a ser n.

Es una estructura eficiente para secuencias de operaciones no uniformes (aquellas que no son isoprobables) o cuando el patrón específico de la secuencia de operaciones no es conocido. Los árboles *Splay* son eficientes en un sentido ***amortizado***: no se considera cada operación sobre el árbol por separado, sino dentro de una secuencia, generalmente larga, de operaciones; desde este punto de vista, cualquier operación particular puede ser lenta, pero toda secuencia de operaciones deberá ser rápida.

El procedimiento usado para llevar a cabo la reestructuración del árbol, que también se llama *splay*, mueve un nodo específico a la raíz del árbol realizando una secuencia de rotaciones a lo largo de la ruta (original) desde el nodo hasta la raíz. Es claro que si este nodo es muy profundo, hay muchos otros nodos en el camino que también son relativamente profundos y que serán movidos más arriba en el árbol, durante la reestructuración, posibilitando que los futuros accesos a los mismos sean menos costosos. Otros nodos, en cambio serán enviados hacia las hojas.

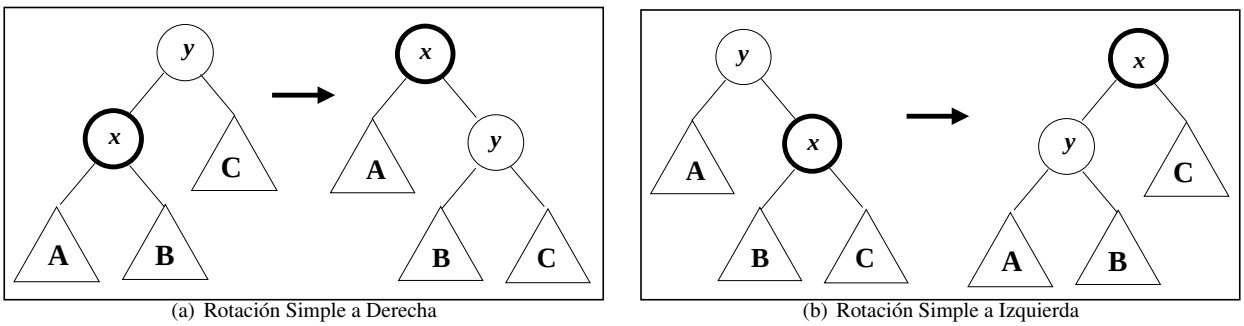
Luego de una reestructuración, un árbol *Splay* puede desbalancearse y provocar que el acceso a un nuevo nodo sea muy costoso. Sin embargo, si analizamos una secuencia larga de accesos, los accesos costosos (que son pocos) son promediados con los menos costosos (que generalmente son más) logrando una muy buena performance. Es decir, los árboles *Splay*, reducen el *tiempo amortizado*, lo que significa que reducen el tiempo promedio de una operación para la peor secuencia de operaciones.

Reestructuración

Supongamos que deseamos realizar una secuencia de operaciones de acceso sobre un ABB. Para que el tiempo de acceso total sea pequeño, los elementos accedidos más frecuentemente deben estar cerca de la raíz del árbol. Entonces, como se dijo, el objetivo es idear una forma sencilla de reestructurar el árbol después de cada acceso, que mueva el elemento accedido más cerca de la raíz, con la suposición de que es probable que pronto se acceda de nuevo a este elemento. La reestructuración se puede realizar de varias formas, pero su fin es llevar el nodo accedido hacia la raíz, y si el mismo está muy profundo en el árbol, es esperable que esta reestructuración tenga como efecto colateral el equilibrar, en algún sentido, el árbol.

Tanto Allen y Munro como Bitner propusieron dos heurísticas de reestructuración:

- *Rotación simple*: Después de acceder al elemento en un nodo x, rotar la arista que une x con su padre (a menos que x sea la raíz). Es decir que se ejecuta una rotación simple, como la usada en los AVL, entre x y su padre. (Ver Figura 1).
- *Mover a la raíz*: Después de acceder al elemento en un nodo x, rotar la arista que une x con su padre y repetir este paso hasta que x sea la raíz. Esto es, ejecutar rotaciones simples como las usadas por los árboles AVL, desde abajo hacia arriba, en el camino desde el nodo accedido hasta que éste llegue a la raíz.



(a) Rotación Simple a Derecha (b) Rotación Simple a Izquierda

Figura 1: *Rotación Simple*

Analicemos el proceso *mover a la raíz* sobre el árbol de la Figura 2. Supongamos que se accede al elemento almacenado en el nodo x. Esto significa que todos los nodos encontrados en el camino de acceso se rotan con su

padre. En la figura este camino se muestra con una linea punteada; entonces el primer paso sería rotar el nodo x con su padre, en este caso el nodo y, obteniendo el árbol mostrado en la Figura 3 a).

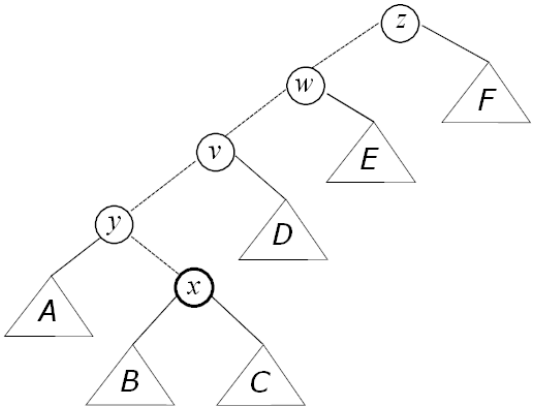
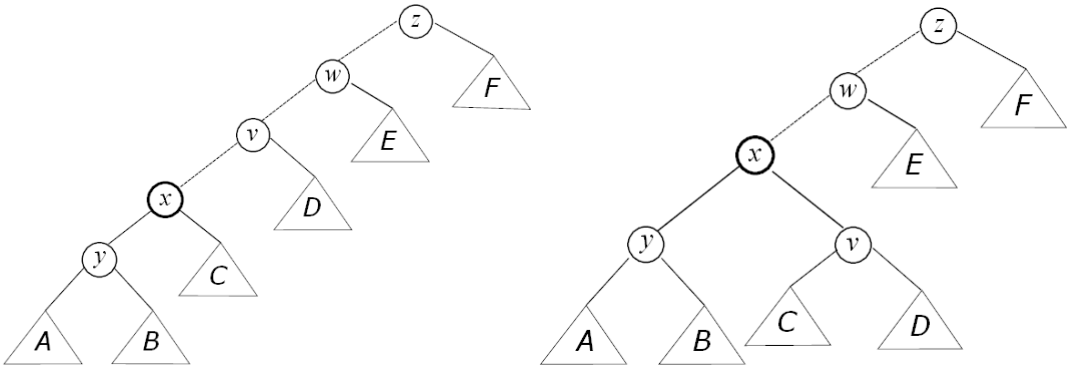


Figura 2: *Mover a la raíz*

El paso siguiente sería rotar el nodo x con su nuevo padre, que ahora es el nodo v, el árbol que resulta de esta rotación se muestra en la Figura 3 b). Luego de dos rotaciones más, entre x y w primero, Figura 4 a), y entre x y z después, el nodo x llega finalmente a la raíz, Figura 4 b).

De esta forma se logra el objetivo planteado pero desafortunadamente, ninguna de estas heurísticas es eficiente en un sentido amortizado: para cada una, hay secuencias de acceso arbitrariamente largas tales que el tiempo por acceso es $O(n)$. Allen y Munro demostraron que la heurística *Mover a la raíz* tiene un tiempo de acceso promedio asintótico que está dentro de un factor constante del costo mínimo, pero solo bajo el supuesto de que las probabilidades de acceso de los diversos elementos *son fijas* y los accesos son independientes.

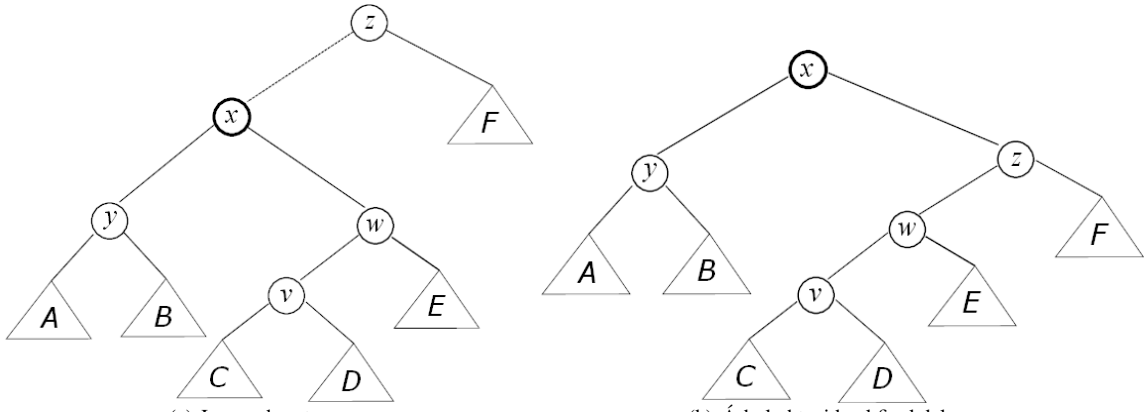


(a) Luego de rotar x e y (b) Luego de rotar x y v

Figura 3: *Mover x a la raíz*

Esto se debe a que, al rotar repetidamente un nodo, otros nodos pueden descender demasiados niveles en el árbol, y por lo tanto el árbol puede quedar muy desbalanceado. Si analizamos el árbol resultante del ejemplo anterior (Figura 4, (b)), puede observarse que si bien el nodo x llegó a ser la raíz del árbol y un próximo acceso al mismo será muy poco costoso, este proceso empujó a otro nodo (el v) tan profundo como estaba x. Un acceso a este nodo llevará a otro nodo a esa profundidad y un requerimiento sobre este tercer nodo provocará lo mismo sobre otro y así sucesivamente. Esta secuencia que requiere nodos profundos cada vez resulta muy costosa.

Otra estartegia para reestructurar el árbol es mediante el procedimiento llamado ***splay***. Este procedimiento también mueve el nodo accedido hacia la raíz utilizando rotaciones, pero es un poco más selectivo sobre cómo se efectúan las rotaciones.



(a) Luego de rotar x y w (b) Árbol obtenido al final del proceso

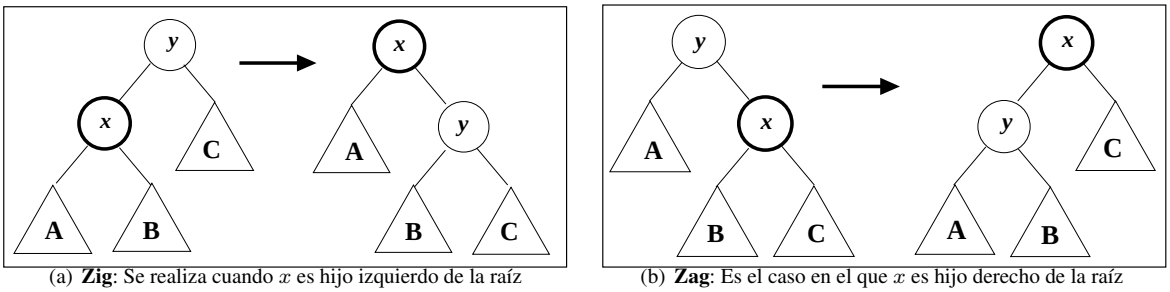
Figura 4: *Mover x a la raíz*

Procedimiento Splay

La heurística de reestructuración llamada ***splay*** propuesta por Sleator-Tarján, es similar a *mover a la raíz* en el sentido de que realiza rotaciones de abajo hacia arriba a lo largo de la ruta de acceso a un elemento, digamos x y mueve a dicho elemento hasta la raíz. Pero se diferencia de ésta en que hace las rotaciones *por parejas*, en un orden que depende de la estructura del camino de acceso. Los pasos utilizados para realizar el *splay* consisten de una o dos rotaciones, según corresponda, y pueden ser de tres tipos (además de sus simétricos).

Supongamos que se accede al nodo x, entonces, para realizar un *splay* a un árbol en el nodo x, repetimos los siguientes *pasos de splay* hasta que x sea la raíz del árbol. Llamaremos $p(x)$ al padre de x en el árbol y $g(x)$ a su 'abuelo":

Caso 1: Lo llamamos **Zig**, $p(x)$ **es la raíz del árbol** y x es su hijo izquierdo. En este caso se hace **una rotación simple entre** x **y su padre** (rotación simple a derecha). Este es un caso *terminal*, es la última rotación que se hace al final de un camino de acceso y deja a x en la raíz del árbol. Su simétrico es llamado **Zag** y consiste en una rotación simple a izquierda. (Ver Figura 5)

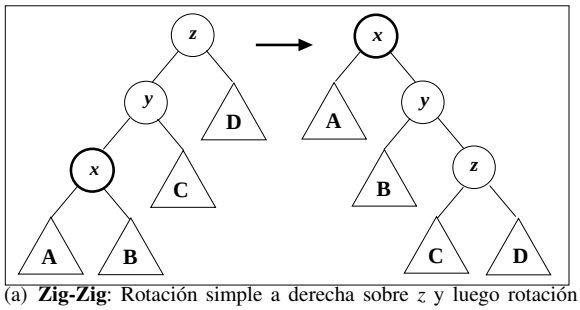


(a) **Zig**: Se realiza cuando x es hijo izquierdo de la raíz (b) **Zag**: Es el caso en el que x es hijo derecho de la raíz

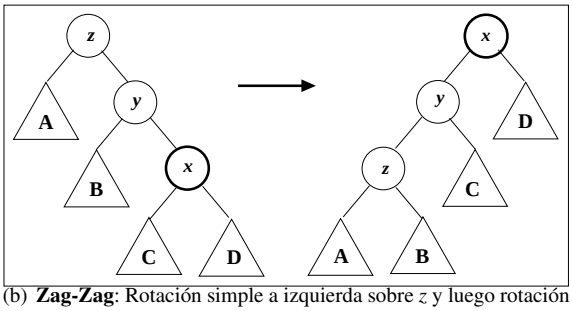
Figura 5: ***Zig y Zag**, rotaciones simples y terminales*

Caso 2: Lo llamamos **Zig-Zig**, si $p(x)$ **no es la raíz del árbol**, entonces x tiene un padre, $p(x)$, y un abuelo, $g(x)$; además, tanto x como $p(x)$ son hijos izquierdos. En este caso, **se rota la arista que une** $p(x)$ **con el abuelo** $g(x)$ y luego **se rota la arista que une** x **con** $p(x)$**,** son dos rotaciones simples a derecha (Figura 6(a)), primero sobre $g(x)$ y luego sobre $p(x)$. En el caso simétrico, llamado **Zag-Zag**, tanto x como $p(x)$ son hijos derechos y las rotaciones son a izquierda. (Figura 6(b))

Caso 3: Lo llamamos **Zig-Zag**, $p(x)$ **no es la raíz del árbol**, además $p(x)$ es hijo izquierdo de $g(x)$ y x es hijo derecho de $p(x)$. En este caso se realiza una de las rotaciones utilizadas en los AVL, la *rotación doble a derecha*: **se rota la arista que une** x **con** $p(x)$ (rotación simple a izquierda) y luego **se rota la arista que une** x **con su nuevo padre** $g(x)$ (rotación simple a derecha) (Figura 7(a)). El caso simétrico es llamado



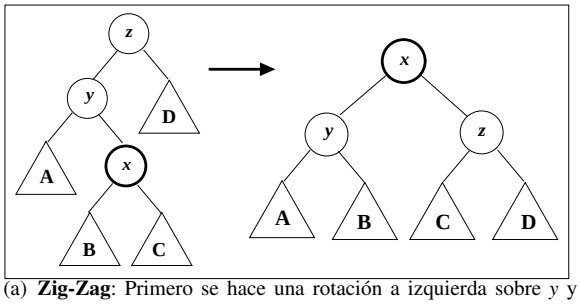
(a) **Zig-Zig**: Rotación simple a derecha sobre z y luego rotación simple a derecha sobre y



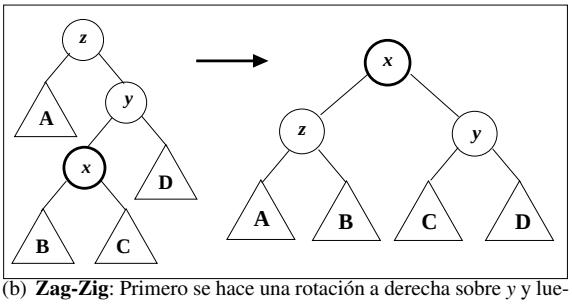
(b) **Zag-Zag**: Rotación simple a izquierda sobre z y luego rotación simple a izquierda sobre y.

Figura 6: ***Zig-Zig*** *y* ***Zag-Zag****, consisten en dos rotaciones simples a derecha o izquierda según corresponda*

Zag-Zig, aquí $p(x)$ es hijo derecho de $g(x)$ y x es hijo izquierdo de $p(x)$, entonces se realiza lo que para los AVL llamamos *rotación doble a izquierda*. (Figura 7(b))



(a) **Zig-Zag**: Primero se hace una rotación a izquierda sobre y y luego una rotación a derecha sobre z



(b) **Zag-Zig**: Primero se hace una rotación a derecha sobre y y luego una rotación a izquierda sobre z

Figura 7: ***Zig-Zag*** *y* ***Zag-Zig****, consisten en una rotación doble a derecha o izquierda según corresponda*

Analicemos el proceso de splay sobre un ejemplo que permita ver con más claridad como se realiza la reestructuración de un *Splay*. La Figura 8 muestra un ejemplo en el que se ha accedido al nodo rotulado con la letra c, lo que provoca un splay que lo llevará a la raíz del árbol. En la misma pueden observarse cada uno de los pasos que se realizan a lo largo del camino de acceso, para convertir al nodo c en la raíz del árbol.

Se realiza una secuencia pasos, de los ya explicados, determinada por el lugar que ocupa el nodo ***c*** dentro del árbol. En este caso se ejecutan un paso ***Zig-Zig***, un ***Zig-Zag*** y finalmente un ***Zig*** dejando a ***c*** en la raíz. En resumen, un splay realiza repetidamente uno de los pasos ***Zig-Zig***, ***Zig-Zag Zag-Zag***, ***Zag-Zig*** sobre el nodo accedido hasta que éste es la raíz o hijo de la raíz; en éste último caso, es necesario realizar un ***Zig*** o un ***Zag*** para llevar dicho nodo finalmente a la raíz.

Realizar splay en un nodo x de profundidad d toma $\theta(d)$ tiempo, es decir, un tiempo proporcional al tiempo necesario para acceder al elemento en el nodo x. Aunque es difícil verlo en ejemplos pequeños, el proceso de splay no solo mueve x a la raíz, sino que reduce aproximadamente a la ***mitad*** la profundidad de cada nodo a lo largo de la ruta de acceso, mientras que algunos de los nodos que se encontraban cerca de la raíz, serán bajados a lo más ***dos niveles***. Este efecto de reducción a la mitad de las profundidades de ciertos nodos, hace que la distribución sea eficiente y es una propiedad que no comparten otras heurísticas más simples, como *mover a la raíz*.

Volvamos a analizar el primer ejemplo que vimos, Figura 2, pero ahora realizando un proceso *splay* en lugar del proceso *mover a la raíz* que se realizó en ese caso. Supongamos que se accede al nodo x almacenado en el árbol *Splay* de la Figura 9; el camino de acceso al mismo se muestra en linea punteada. En la Figura 10 pueden verse los pasos que se siguen para llevar a x a la raíz, un paso Zig-Zag entre los nodos x, y y v (Figura 10(a)) y un paso Zig-Zig entre los nodos x, w y z (Figura 10(b)). Como puede observarse, después de todo el proceso de splay el árbol que se obtiene es más balanceado y de menor altura que el que se obtuvo al final del proceso mover a la raíz (Figura 4).

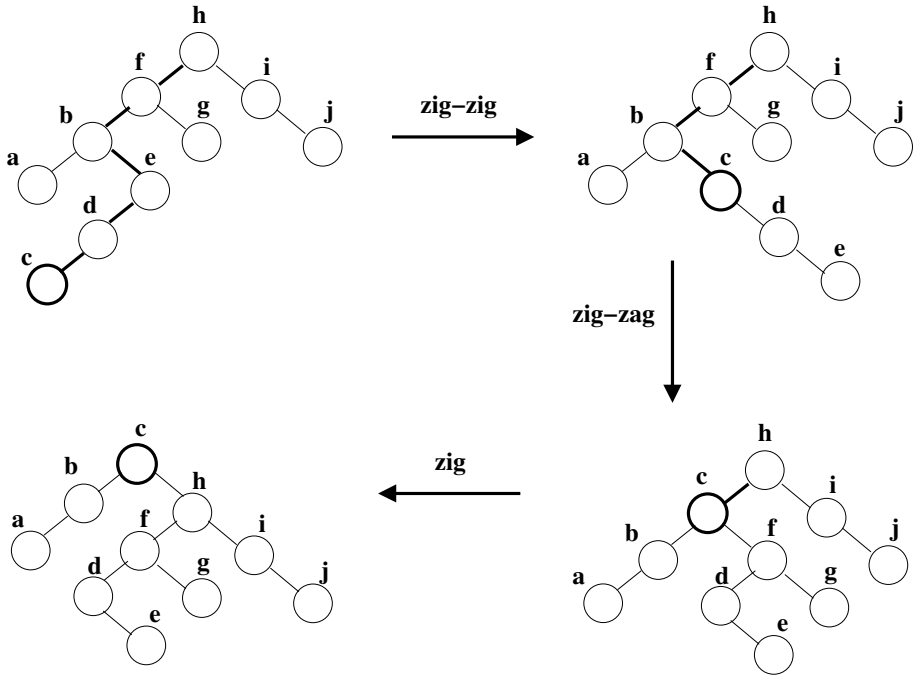


Figura 8: *Ejemplo de **splay** sobre el nodo **c***

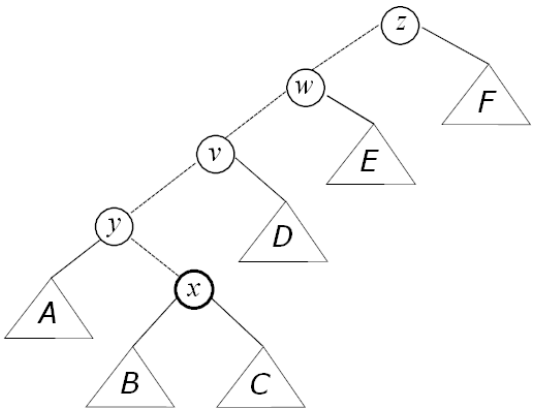


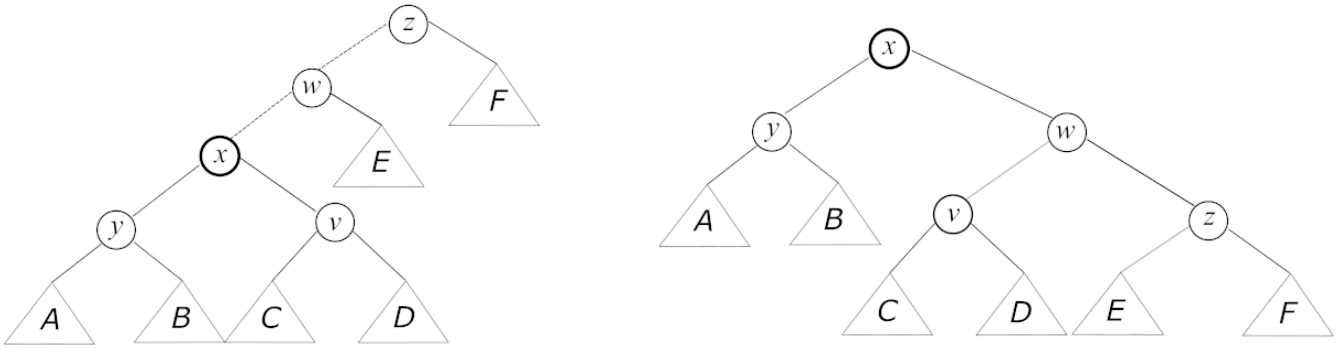
Figura 9: *Se accede al nodo x en un árbol Splay*

Operaciones sobre Splay Trees

Todas las operaciones básicas (localización, evocación, inserción y supresión) para árboles binarios de búsqueda pueden ser implementadas para los árboles *Splay* realizando, cuando corresponda, el proceso de splay.

Localización

Una localización sobre un árbol *Splay* comienza de la misma forma que la localización sobre un árbol binario de búsqueda común: compara el elemento buscado con la raíz, si el mismo está allí el proceso termina y si no es él se sigue a derecha o izquierda según corresponda; la diferencia aparece al finalizar la búsqueda: en los árboles autoajustables la localización *siempre* finaliza realizando un splay. Así, una localización exitosa que, encuentra el nodo que contiene al elemento buscado, termina el proceso con un splay sobre ese nodo y devuelve un árbol con el elemento buscado como su raíz; caso contrario, si la localización fracasa, se realiza un splay sobre el último nodo visitado y devuelve un árbol que tendrá como raíz dicho nodo.



(a) Luego del **Zig-Zag** entre x, y y v

(b) Luego del **Zig-Zig** entre x, w y z

Figura 10: *Proceso de splay luego de acceder al nodo x*

Un ejemplo de localización exitosa es el que se muestra en la Figura 8, en la cual el elemento buscado es el que se encuentra en el nodo al que hemos rotulado con la letra c. La localización comienza del mismo modo que en ABB, se compara el elemento buscado con la raíz del árbol y si son distintos, se mueve a derecha o izquierda según sea éste mayor o menor que el elemento que allí se encuentra (en este caso sigue por la izquierda de la raíz). El proceso continúa hasta encontrar el elemento buscado, o hasta llegar a un puntero a *nil*; en este punto en que el localizar finaliza en un ABB, en un *Splay* comienza el proceso de llevar al último elemento que visitó el localizar a la raíz; en este caso se realiza el proceso de splay de c.

La Figura 11 muestra un ejemplo de una localización que fracasa. En este caso el elemento buscado es el "35", el cual no pertenece al conjunto almacenado en la estructura. Al realizar la localización del mismo, ésta va bajando en el árbol (el camino seguido se muestra con linea más gruesa) y fracasa en el nodo con contenido "38"; la misma termina dejando un árbol con este nodo como raíz, mediante un splay (zag-zig, zig-zag, zig) cómo puede observase en la Figura 11.

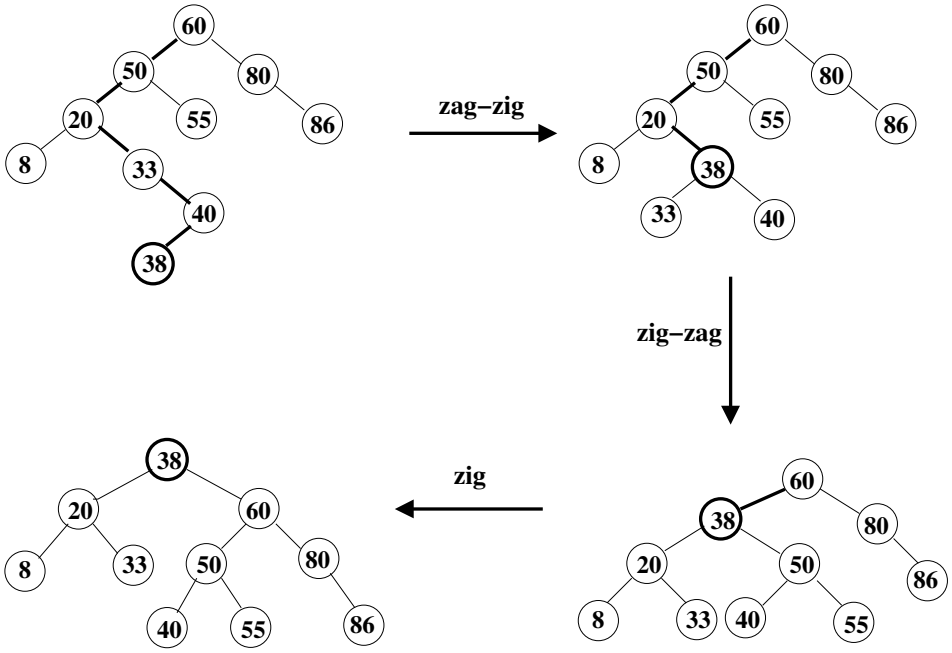


Figura 11: *Ejemplo de localización no exitosa. Localización del elemento 35*

Evocación

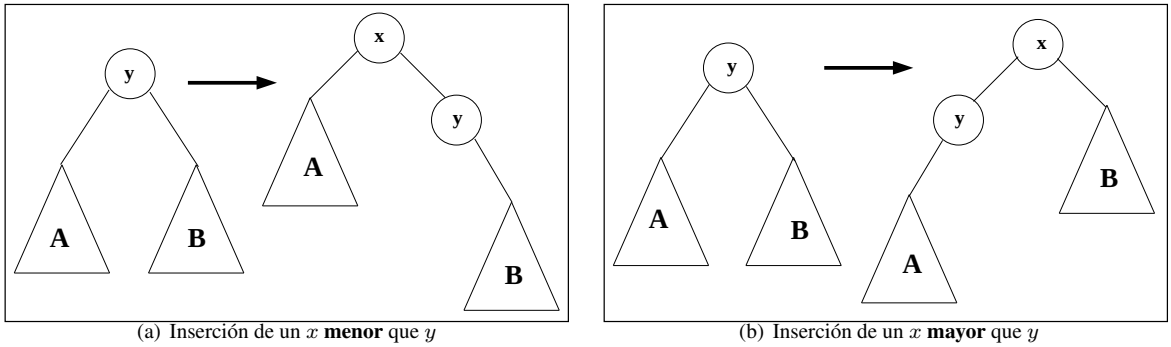
Como sabemos, una evocación consta, al igual que en el resto de las estructuras, de una localización seguida de la devolución de los valores asociados al elemento buscado, si la localización fue exitosa, o un aviso de fracaso si la misma no tuvo éxito. En un árbol *Splay*, si el elemento evocado pertenece a la estructura, la localización retorna un árbol con el elemento buscado en la raíz y el paso final para completar la operación de evocación es devolver los valores asociados al elemento en la raíz.

En el caso del ejemplo mostrado en la Figura 8, el localizar devolverá la dirección de la raíz del árbol y el éxito en *verdadero* indicando que el elemento fue encontrado, y finalmente la rutina de evocación accederá a la raíz para retornar la información asociada a c. En el caso mostrado en la Figura 11, el localizar devolverá la dirección de la raíz del árbol pero el éxito estará en *falso*, indicando que el elemento *no* fue encontrado, y la rutina de evocación indicará que el "35"no se encuentra en el conjunto almacenado.

Inserción

Del mismo modo que se definió en las estructuras ya analizadas, una inserción consiste en agregar a la estructura un elemento, suponiendo que éste todavía no está en ella. Por lo tanto, antes de hacer un alta en un árbol *Splay*, debemos comprobar que el elemento a insertar no pertenece a la estructura. Por esta razón, el primer paso ante una inserción de un nuevo elemento es localizarlo. Si la localización es exitosa, la inserción fracasa. Si la localización fracasa, es posible insertar el nuevo elemento (si hay espacio disponible), éste quedará como nueva raíz del árbol.

En un *Splay*, cuando la localización fracasa, devuelve un árbol cuya raíz es el último nodo visitado durante la búsqueda; con esto en mente, se deben realizar las modificaciones necesarias para agregar el nuevo elemento a la estructura. Las modificaciones correspondientes dependen de si el elemento almacenado en la raíz del árbol *es menor* o *es mayor* que el elemento que se quiere insertar.



(a) Inserción de un x **menor** que y (b) Inserción de un x **mayor** que y

Figura 12: *Modificación estructural del Splay durante una inserción*

Llamaremos x al elemento a insertar e y al elemento en el que fracasó la búsqueda y quedó como raíz del *Splay* luego del localizar. Así, los pasos a seguir en cada uno de los casos mencionados son los siguientes:

Caso 1: ***El elemento*** x ***es menor que el elemento*** y ***en el que fracasó la búsqueda.*** En esta situación se cumple que x es ***mayor*** que todos los elementos del subárbol izquierdo de y ya que la manera de autoajustar el *Splay* asegura que eso se satisfaga. Las operaciones a realizar en este punto son:

- Colocar a x como nueva raíz del *Splay*.
- La antigua raíz y pasa a ser hijo *derecho* de x.
- El subárbol izquierdo de y se convierte en subárbol izquierdo de x.
- El subárbol derecho de y *no* se modifica.

El proceso explicado en el Caso 1 se muestra en la Figura 12(a).

Caso 2: ***El elemento x es mayor que el elemento y en el que fracasó la búsqueda.*** En esta situación se cumple que x es ***menor*** que todos los elementos del subárbol derecho de y ya que el splay sobre la estructura asegura que eso se satisfaga. Las operaciones a realizar en este caso son:

- Colocar a x como nueva raíz del *Splay*.
- La antigua raíz y pasa a ser hijo *izquierdo* de x.
- El subárbol izquierdo de y *no* se modifica.
- El subárbol derecho de y se convierte en subárbol derecho de x.

El proceso explicado en el Caso 2 se muestra en la Figura 12(b).

La Figura 13 muestra un ejemplo en el que se quiere insertar el elemento "35" en la estructura. Se ha usado el mismo árbol de la Figura 11 para no repetir los pasos que involucra el fracaso de la localización del elemento "35", ya que en dicha figura puede verse el detalle de este proceso. La localización deja al "38" como raíz del *Splay* y, como "35" es menor que "38", se deben seguir los pasos del Caso 1.

En la figura puede observarse que *todos* los elementos del subárbol izquierdo de "38" son menores que "35", por lo cual, al poner al "35" como su raíz se mantiene la propiedad de "*hijos menores a la izquierda*"; luego el "38" pasa a ser hijo derecho de "35", y su subárbol derecho no se modifica en nada.

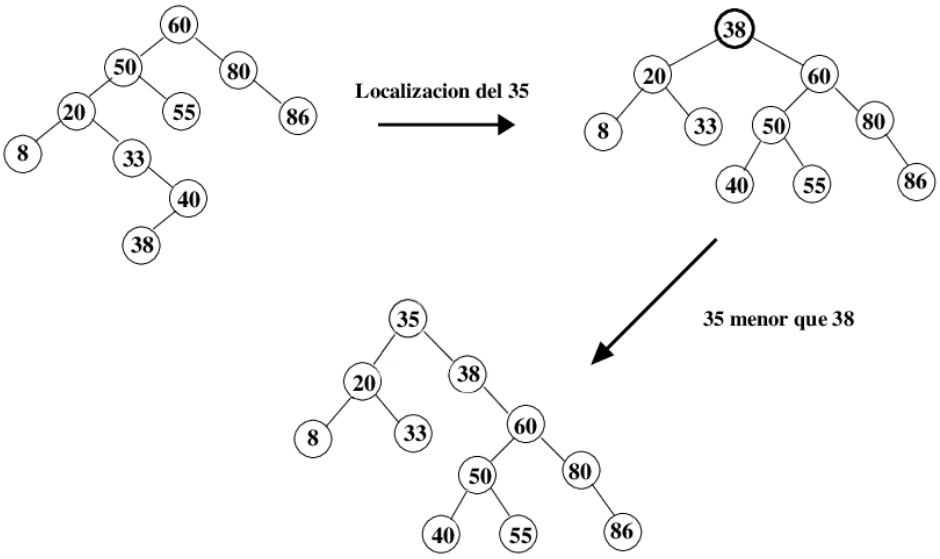


Figura 13: *Inserción del elemento 35*

La Figura 14 muestra el proceso de inserción del elemento "39" en la estructura. Nuevamente se utiliza el árbol de la Figura 11 para obviar los pasos del splay durante la localización del elemento "39".

La localización deja al "38" como raíz del *Splay*, pero en este caso el elemento de la raíz es menor que lo que se intenta insertar, entonces se deben seguir los pasos del Caso 2. Es claro ver en la figura que *todos* los elementos del subárbol derecho de "38" son *mayores* que "39", por lo cual, al poner al "39" como su raíz se mantiene la propiedad de "*hijos mayores a la derecha*"; entonces, el "38" pasa a ser hijo izquierdo de "39", y su subárbol izquierdo sigue siendo su hijo izquierdo.

Supresión

Suprimir un elemento de una estructura implica quitar ese elemento de la misma, asumiendo que éste está almacenado en ella. De nuevo, en un *Splay*, al igual que en cualquier estructura, es necesario realizar una localización antes de eliminar un elemento, para comprobar si el mismo pertenece o no a la relación almacenada en la

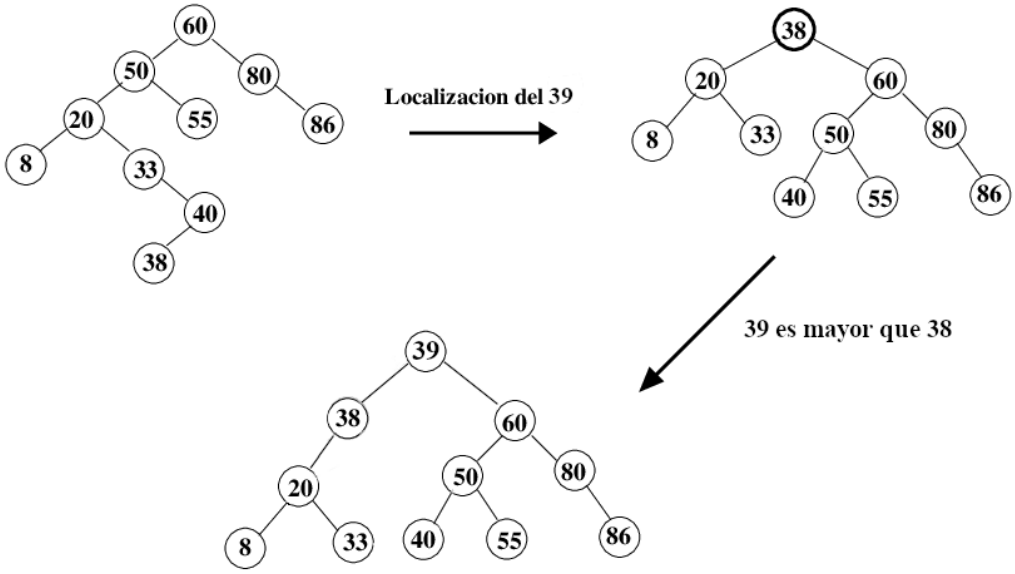


Figura 14: *Inserción del elemento* 39

misma. Si la localización es exitosa, la supresión se lleva a cabo; si la localización fracasa, no es posible hacerlo, ya que no se puede eliminar un elemento que no está presente.

Supongamos que queremos eliminar de un *Splay* el elemento x; el procedimiento necesario para llevar a cabo esta operación se muestra en la Figura 15 y puede resumirse en los siguientes pasos:

- Localizar el elemento x, si el mismo está en la estructura, que es el caso en que la eliminación será exitosa, la localización devuelve un árbol que tiene a x como su raíz.
- Teniendo a x en la raíz del árbol se puede eliminar fácilmente pero, como sabemos, si x tiene dos hijos se debe buscar algún candidato para ocupar ese lugar, teniendo en cuenta las restricciones de los ABB y considerando además que estamos en un árbol *Splay*.
- Para elegir un candidato se debe decidir la política de reemplazo a utilizar. Éste puede ser el elemento *más grande* del subárbol izquierdo de x (el subárbol A), es decir se elige la política "*el mayor de los menores*"; o el candidato puede ser aquel que es el *menor* en el subárbol derecho de x (el subárbol B), en este caso la política utilizada es "*el menor de los mayores*"). Supongamos que se elige la política *el mayor de los menores* para encontrar un candidato, éste se puede obtener realizando una búsqueda del elemento $+\infty$ en el subárbol A. Dado que ese elemento no pertenece a la relación almacenada la localización fracasará, además como el $+\infty$ es mayor que todos los elementos de A, esta localización termina al visitar el objeto más grande de A. Entonces, luego de realizar el splay correspondiente, el subárbol izquierdo se convierte en un nuevo árbol que tiene como raíz el *mayor* elemento de A.
- Por último, el subárbol derecho de x, es decir B, pasa a ser *hijo derecho* de la raíz del subárbol en que se convirtió el subárbol A luego del splay. Es claro que este nuevo árbol, llamémoslo A', no tiene hijos por la derecha, ya que su raíz, el nodo que aparece con valor z en la Figura 15, es mayor que todos sus elementos.

La Figura 16 muestra un ejemplo de eliminación. Usamos el mismo árbol del ejemplo de la Figura 11 para obviar los pasos que involucra la localización del elemento a eliminar, en este caso el 38, los cuales pueden seguirse en esa figura. En la Figura 16 se muestra el proceso que se debe realizar para llevar a cabo la eliminación: luego de encontrar el elemento a eliminar, el *Splay* se reestructura dejando al mismo en la raíz desde donde se elimina. Asumiendo que la política de reemplazo es "él mayor de los menores", se debe encontrar el elemento que será la nueva raíz, para eso se busca el $+\infty$ en el subárbol izquierdo del 38 y dado que éste no se encuentra en el conjunto, se obtiene un árbol con el elemento 33 como raíz. Por último se conecta el hijo derecho de 38 como hijo derecho del 33, finalizando el proceso.

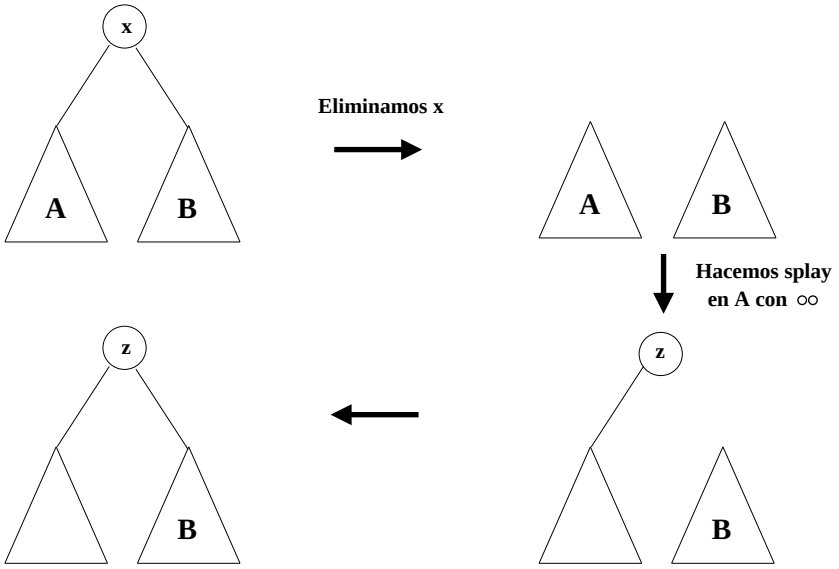


Figura 15: *Modificación estructural necesaria para la eliminación del elemento x*

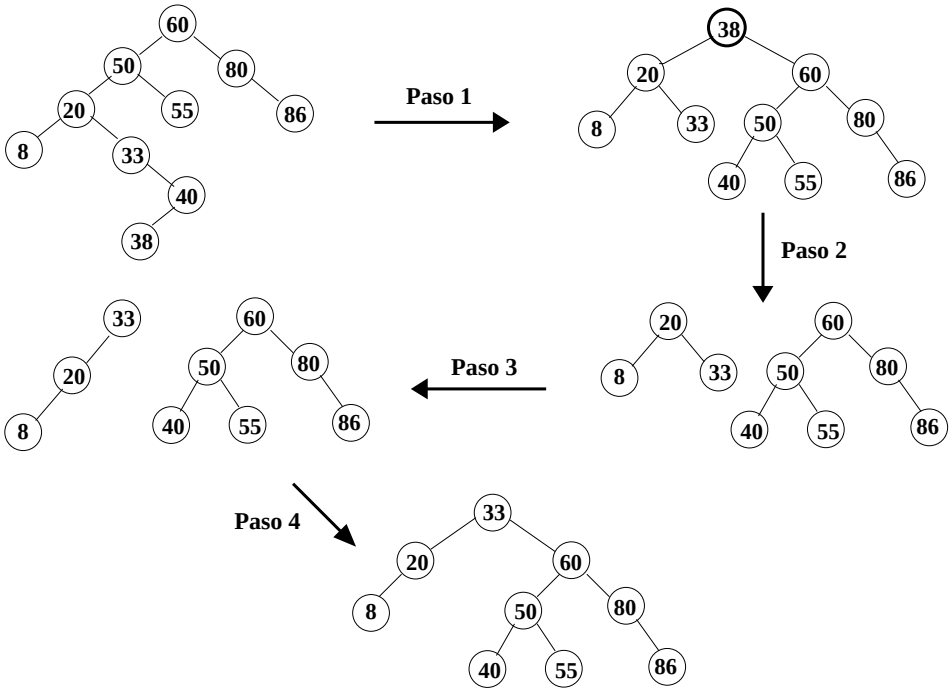


Figura 16: *Eliminación del elemento 38*

Análisis de Costos del *Splay*

En general, en las estructuras vistas hasta aquí, ya sea que se haga un análisis de costos a priori o un análisis de costos a posteriori, siempre se han considerado dos tipos de evaluaciones: el costo máximo y el costo medio de las diferentes operaciones consideradas. En todos los casos de análisis, se considera un evento o situación simple para determinar su costo (un alta, una evocación, etc.). Todas estas estructuras de datos están diseñadas para reducir el costo por operación en el peor de los casos. Sin embargo, en aplicaciones típicas, cuando se administra una estructura, no se realiza una sino una *secuencia* de operaciones, y lo que importa es el costo total que toma la secuencia completa, no los costos individuales de cada una de las operaciones de la secuencia.

Además es posible que luego de una operación la estructura se modifique (por ejemplo con una baja), lo que ocasiona que el costo del próximo evento dependa de la nueva estructura. Una operación, en una secuencia, puede ser muy costosa de realizar, pero puede dejar la estructura en un estado tal que las operaciones siguientes sean muy poco costosas; es decir, ese evento afecta el costo de los eventos posteriores. Esto es más notorio en una estructura como el *Splay* que se reestructura constantemente (luego de cada operación), por lo tanto, en éstos casos se debe pensar en otro tipo de análisis de costos de las operaciones sobre la misma. Introducimos aquí el concepto de un nuevo tipo de análisis: el ***análisis amortizado***.

Análisis Amortizado

Como se mencionó anteriormente, hay muchos casos en los cuales, en lugar de caracterizar el costo (mejor, peor, medio) de cada operación individual resulta más adecuado considerar *secuencias de operaciones* arbitrarias sobre la estructura de datos. Es posible que una operación tenga un peor costo elevado, pero que éste sea muy improbable cuando la operación se aplica dentro de secuencias de operaciones válidas. El costo amortizado evalúa la eficiencia de un conjunto de operaciones que se aplican en un mismo contexto.

Entonce, el análisis amortizado considera una secuencia de operaciones sobre una estructura de datos (en lugar de considerar operaciones o eventos simples, aislados) y determina una cota superior para el tiempo de esta secuencia; en particular, se considera el tiempo de ejecución para el peor caso de cualquier secuencia de m operaciones. En tales aplicaciones, un mejor objetivo es reducir los *tiempos amortizados* de las operaciones, donde por tiempo amortizado nos referimos al tiempo promedio de una operación en una secuencia de operaciones en el peor de los casos.

Esto contrasta con el análisis más común, en el cual se da una cota del peor caso para cualquier operación simple. Supongamos que se realiza el análisis del peor caso de manera normal: ejecutar N operaciones sobre una estructura de datos de n elementos lleva tiempo $O(Nf(n))$, donde $f(n)$ es el tiempo en el peor caso de una operación. Sin embargo, en muchos casos esa cota no es ajustada debido a que el peor caso puede NO ocurrir las N veces, o incluso ninguna de esas veces.

La forma de precisar el costo amortizado de una operación, es determinar el costo total de la secuencia de operaciones en la que aparezca y dividirlo por el número de éstas. Cuando cualquier secuencia de m operaciones, sobre una estructura de datos de n elementos, tiene tiempo total de $O(mf(n))$ en el peor caso, se dice que el tiempo amortizado de cada operación es $O(f(n))$. El análisis amortizado garantiza la eficiencia media de cada operación en el peor caso; no obstante, el análisis amortizado difiere del análisis del caso promedio en que la probabilidad *no* está involucrada.

Las cotas amortizadas son *más débiles* que las cotas correspondientes para el peor caso, porque no ofrecen garantía para ninguna operación simple. Como por lo regular esto no es importante, se está dispuesto a sacrificar la cota en una operación simple, si podemos retener la misma cota para la secuencia de operaciones y al mismo tiempo simplificar la estructura de datos. Las cotas amortizadas son *más fuertes* que la cota del caso medio. Por ejemplo, los árboles binarios de búsqueda tienen un tiempo medio $O(\log n)$ por operación, pero aun así es posible que una secuencia de m operaciones tome un tiempo $O(m\ n)$.

Costo Amortizado del *Splay*

Si en una estructura dada, una operación puede tener costo lineal ($O(n)$) y aún así tener costo amortizado sublineal, es necesario, que el elemento accedido en cada operación sea movido. Caso contrario, se podrían dar

secuencias de m operaciones que tomen tiempo $O(m\ n)$ accediendo m veces al nodo con costo lineal (el más costoso).

Si se consideran los árboles AVL, éstos permiten las operaciones estándar sobre árboles en un tiempo por operación de $O(\log n)$ para el peor caso. Sin embargo su implementación puede ser complicada y además hay que mantener y actualizar correctamente información acerca de su equilibrio en altura, incrementando con ello el espacio utilizado; todo esto sin considerar que los árboles equilibrados en altura no son los más eficientes si el patrón de acceso no es uniforme. La razón de que se usen los árboles AVL es que una secuencia de $O(n)$ operaciones sobre un árbol binario de búsqueda, no equilibrado, podría requerir un tiempo $O(n^2)$, ya que en un ABB se pueden dar secuencias completas de operaciones costosas, lo que lo hace ineficiente. En realidad, el tiempo de ejecución $O(n)$ del peor caso de una operación en los árboles binarios de búsqueda, no es el problema real, sino el hecho de que esto podría ocurrir repetidas veces, se pueden dar secuencias completas de malos accesos, entonces el tiempo de ejecución acumulado se vuelve evidentemente caro.

Entonces, dijimos que una forma de obtener eficiencia amortizada es utilizar una estructura de datos "autoajustable". Se permite que la estructura esté en un estado arbitrario, pero durante cada operación se aplica una regla de reestructuración simple destinada a mejorar la eficiencia de las operaciones futuras. Los árboles *Splay* ofrecen una buena alternativa, ya que se basan en el hecho de que un tiempo de $O(n)$ por operación en el peor caso para árboles binarios de búsqueda no es malo, en tanto que esto ocurra con *poca frecuencia*. Aunque cualquier operación todavía puede requerir un tiempo $O(n)$, este comportamiento extremo no ocurre en repetidas ocasiones, y se puede demostrar que cualquier secuencia de m operaciones tarda un tiempo (total) de $O(m \log n)$ para el peor caso; lo que es verdaderamente satisfactorio porque esto demuestra que *no hay secuencias de entrada malas*. Así, a la larga, esta estructura de datos se comporta como si cada operación tomara $O(\log n)$; esta es la llamada cota de tiempo amortizado.

Esto se debe a que, como ya se mencionó, un árbol *Splay* es un árbol binario de búsqueda que se reestructura luego de cada operación. En cada reestructuración no sólo mueve el nodo accedido a la ráız del árbol, haciendo que el mismo pueda ser eficientemente accedido en operaciones posteriores, sino que disminuye la profundidad de cada nodo en el camino de acceso. Esta propiedad hace a la operación de splay muy eficiente y es una propiedad que no logran otras formas de reestructuración. Luego de un splay, un árbol autoajustable puede desbalancearse y provocar que el acceso a otro nodo sea muy costoso. Sin embargo, al realizar cada operación, el árbol tiende a equilibrarse; por ésta razón, si analizamos una secuencia larga de accesos, los accesos costosos son promediados con los menos costosos logrando una muy buena performance.

Veamos algunos ejemplos de casos "malos" de accesos para ilustrar esto. La Figura 17 muestra un árbol en el cual las claves se encuentran en orden decreciente desde la raíz a las hojas $(l, k, j, i, ...)$ determinando el *Splay* con mayor altura posible con esa cantidad de nodos. Supongamos que se quiere acceder a los nodos en orden inverso $(a, b, c, ...)$; en las Figuras 17 y 18 puede verse parte de esta secuencia. Para comenzar, puede observarse que luego de acceder a los nodos a y b el árbol ya está más equilibrado y el tiempo de acceso al nodo c, al igual que los tiempos de algunos nodos más (h, d, e, f, entre otros), ya no resulta tan costoso como lo era en el árbol original. Después de acceder a los nodos c y d, Figura 18, el árbol vuelve a disminuir su altura.

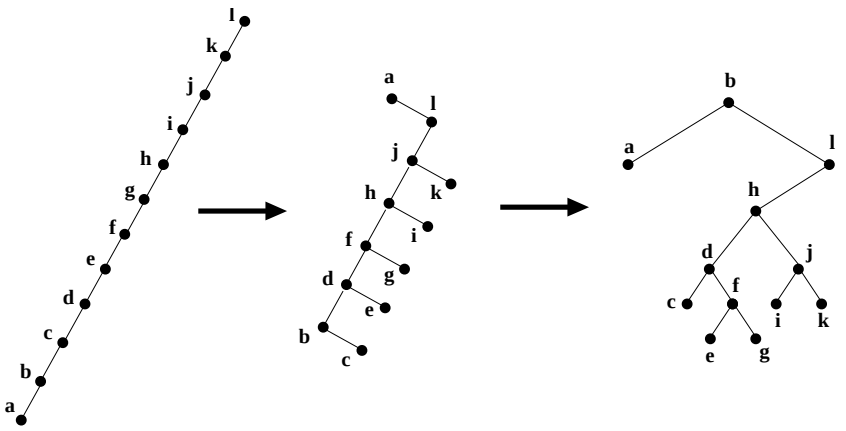


Figura 17: *Casos extremos de acceso a un Splay. Primero se localiza el nodo* **a** *y luego el nodo* **b**

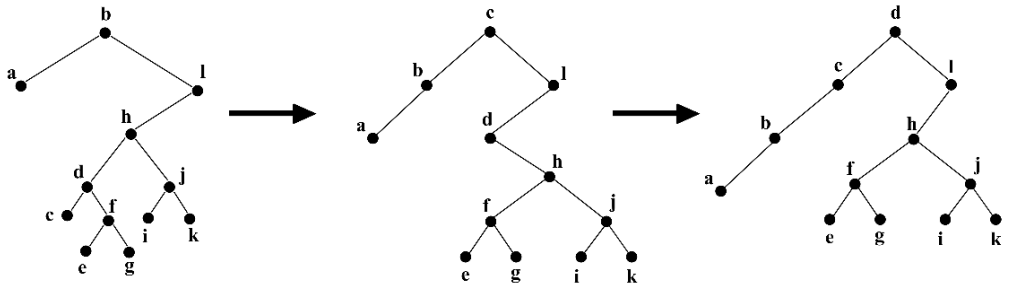


Figura 18: *Casos extremos de acceso a un Splay. Luego de acceder al nodo* **b**, *accede al* **c** *y al* **d**

En la Figura 19 se parte del mismo *Splay* mostrado anteriormente y se realiza la secuencia de accesos (d, g) la cual, si bien no localiza los nodos de mayor profundidad, accede a nodos al azar cuyo tiempo de acceso es alto. Igual que en el ejemplo anterior, puede verse como el *Splay* se va equilibrando luego de cada acceso.

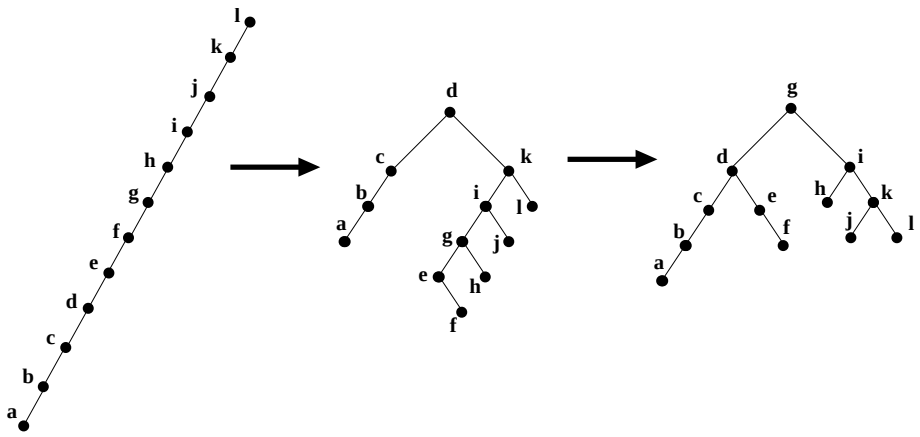


Figura 19: *Casos costosos de acceso a un Splay; la secuencia es* (d, g)

Finalmente, en la Figura 20 se presentan ejemplos de dos *Splays* muy desbalanceados sobre los cuales se realiza un acceso a su nodo más extremo, el que está a mayor profundidad, el nodo a, y al igual que en los casos anteriores puede observarse como el *Splay* reduce su altura drásticamernte y se va equilibrando luego de estas operaciones, haciendo menos costosos los accesos subsiguientes. Entonces, se puede asegurar que en un árbol *Splay* de

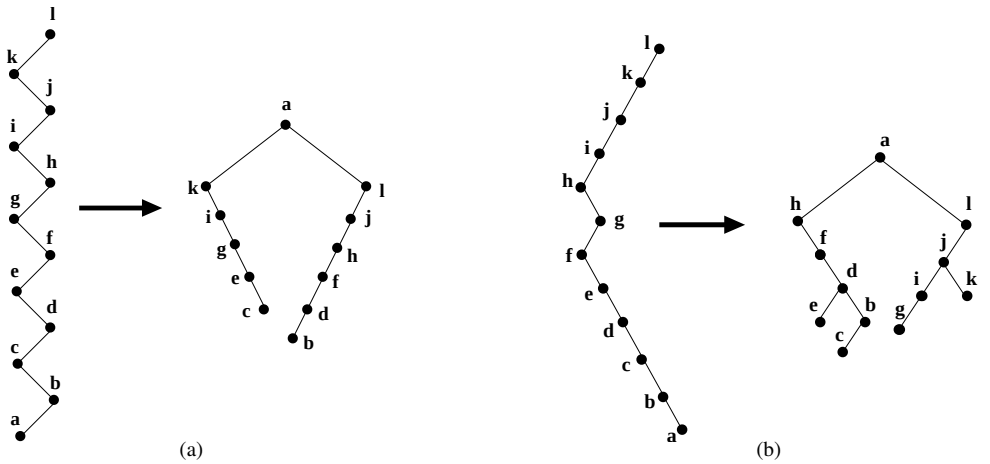


Figura 20: *Casos de acceso extremo a un Splay desbalanceado*

n nodos, todas las operaciones estándar del árbol de búsqueda tienen un límite de tiempo amortizado de $O(\log n)$ por operación. Por lo tanto, en un sentido amortizado e ignorando los factores constantes, los árboles *Splay* son tan eficientes como los AVL y los árboles óptimos estáticos [2], cuando el tiempo total de ejecución es la medida de interés. En particular, los árboles *Splay* son tan eficientes en cualquier secuencia de accesos suficientemente larga como cualquier forma de árbol de búsqueda binario actualizado dinámicamente, incluso uno adaptado a la secuencia de acceso exacta. La eficiencia de los árboles *Splay* no proviene de una restricción estructural explícita, como con los árboles balanceados, sino de la aplicación de una simple heurística de reestructuración (**splay**) cada vez que se accede al árbol.

Reconocimientos

El presente apunte se realizó tomando como referencia artículos y libros citados a continuación:

- *Self-Adjusting Binary Trees*, D. D. Sleator y R. E. Tarjan, Journal of ACM 32 (1985), 652-686.
- *Algorithm Desing*, R. E. Tarjan, Communications of the ACM 30, 3, 1987, 204-212.
- *Self-Adjusting Binary Trees*, D. D. Sleator y R. E. Tarjan, ACM3 4, 1983, 235-245, 0-89791-099-0.
- *Data Structures and Their Algorithms*, Lewis y Denenberg.
- *Estructuras de Datos y Algoritmos*, Weiss, ISBN 0-201-62571-7.
- *Data Structures and Program Design in C*, Kruse, Tondo y Leung.

[2] **https : //cse.hkust.edu.hk/mjg_ib/bibs/DPSu/DPSu.Files/jstor_514.pdf**