Métodos de Ordenamiento

Estos se clasifican en dos categorías: algoritmos de ordenamiento externo y algoritmos de ordenamiento interno. Los algoritmos externos se aplican a conjuntos de datos que son muy grandes para ser manejados en memoria; los algoritmos internos se manejan completamente en memoria.

Ordenamiento Quicksort.

Es un ordenamiento de intercambio con partición (u ordenamiento rápido).

Considere x como un vector y n el número de elementos del vector que han de ser ordenados.

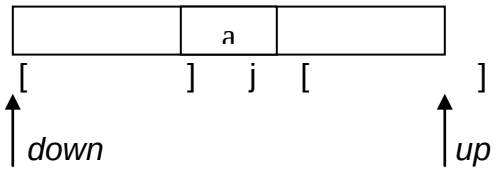
Escoja un elemento a de alguna posición específica dentro del vector (ejemplo, a puede ser escogido como el primer elemento tal que a = x[1]. Asuma que los elementos de x están organizados en tal forma que a queda colocado en la posición j y se cumplen las siguientes condiciones:

1. Los elementos en posiciones 1 hasta j - 1 son menores o iguales a "a".
2. Los elementos en posiciones j + 1 hasta n son mayores o iguales a "a".

Observe que si estas dos condiciones se cumplen para algún valor de a y j en particular, entonces a queda en la posición j cuando el vector quede completamente ordenado. (Usted debe probar este hecho como ejercicio). Si se repite el proceso con los subvectores x[1] hasta x[j - 1] y x[j + 1] hasta x[n] y con cualquiera de los subvectores creados mediante el proceso de iteraciones sucesivas, el resultado final será un archivo ordenado.

Para este algoritmo manejaremos dos punteros: *up* y *down* e iremos ordenando por partes, luego de aplicar el algoritmo tendremos un elemento posicionado en el vector de tal forma que todos los elementos a su izquierda son menores que él y todos los elementos a la derecha son mayores.

1 . . . j-1 < a < j+1 . . . n



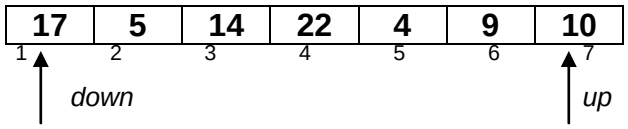
Debo cuestionarme:
si *down* < *up*
up = *up* -1
sino
Intercambio

El algoritmo siempre comienza comparando down y up, en el caso que down< up se decrementará *up.*
Si intercambio valores (down y up) empiezo a mover el otro puntero.
Termina cuando *up=down*

Ilustraremos este método de ordenamiento rápido con un ejemplo. Si el vector inicial es:

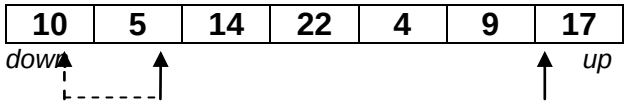
17	5	14	22	4	9	10

Se definen los punteros y se comienza a operar:

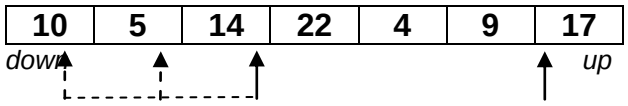


En teoría deberíamos comenzar decrementando up, pero debido que en el ejemplo al comparar ambos punteros el down es menor que el up, se intercambian los valores y luego se continúa incrementando el puntero down.

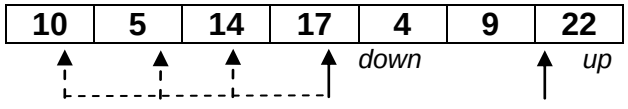
Pregunto si 17<10 ? ➔ Intercambio

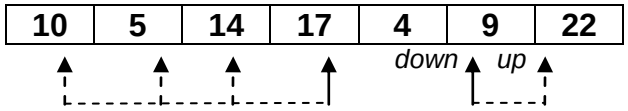


Pregunto si 5<17 ? ➔ No intercambio, muevo el puntero *down*

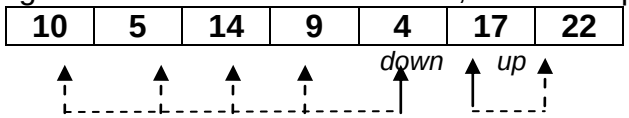


Pregunto si 14<17 ? ➔ No intercambio, muevo el puntero *down*
Pregunto si 22<17 ? ➔ Intercambio, cambio el puntero

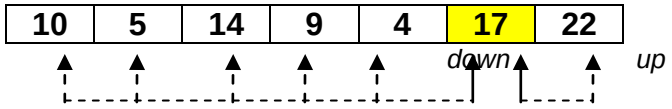




Pregunto si 17<9 ? ➔ Intercambio, cambio el puntero



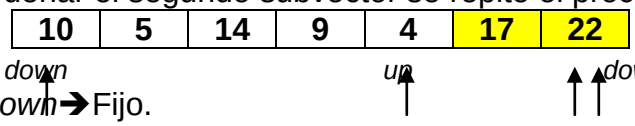
si el primer elemento (17) se coloca en la posición que debe ser, el vector resultante es:



Si *down=up* ➔ el elemento queda fijo todos los elementos de la izquierda son menores y los de la derecha mayores.

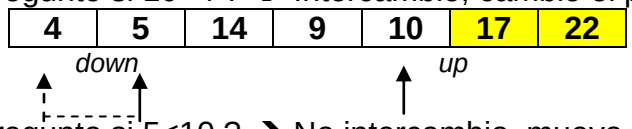
Puesto que 17 está en su posición final, el problema original se ha descompuesto en el problema de ordenar dos subvectores.

Ahora se llama al algoritmo con el subvector izquierdo y el derecho. En este caso no se necesita hacer nada para ordenar el segundo de estos subvectores; un archivo de un solo elemento está ordenado de hecho. Para ordenar el segundo subvector se repite el proceso.

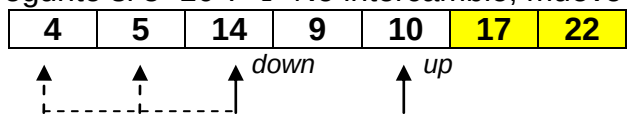


Para el 22 *up=down*➔Fijo.

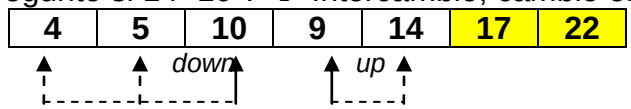
Pregunto si 10<4 ? ➔ Intercambio, cambio el puntero.



Pregunto si 5<10 ? ➔ No intercambio, muevo el puntero *down*

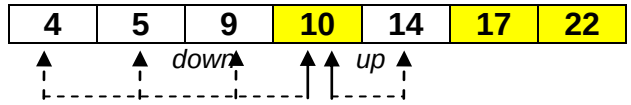


Pregunto si 14<10 ? ➔ Intercambio, cambio el puntero.

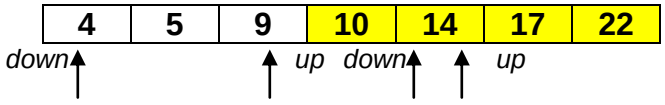


Pregunto si 10<9 ? ➔ No intercambio, muevo el puntero *down*

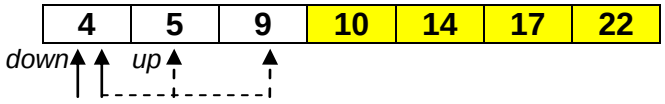
En este caso el subvector es dividido una vez más. El vector completo se puede visualizar como:



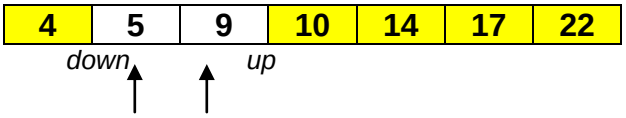
Para el 10 *up=down*➔Fijo.



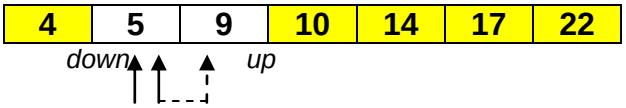
Para el 14 *up=down*➔Fijo.



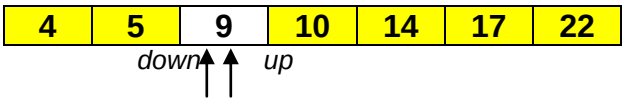
Pregunto si 4<9 ? ➔ No intercambio, muevo el puntero *up.*
Pregunto si 4<5 ? ➔ No intercambio, muevo el puntero *up.*
Para el 4 *up=down*➔Fijo.



Pregunto si 5<9 ? ➔ No intercambio, muevo el puntero *up.*

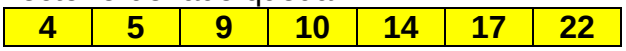


Para el 5 *up*=*down*➔Fijo.



Para el 9 *up*=*down*➔Fijo.

El vector ordenado queda:



En este momento usted debe haber observado que el método de ordenamiento rápido (quicksort) se puede definir en una forma más conveniente como un procedimiento recursivo. Podemos resumir el algoritmo quick(down,up) para ordenar todos los elementos de un vector entre x[down] y x[up] (donde down es el límite inferior, up el límite superior) como sigue:

```
if down < up
then begin    {si down>=up, el vector es ordenado     }
              { y se retorna al programa que          }
              { ha hecho la llamada                   }
     rearrange(down,up,j);
              { reorganiza los elementos del          }
              { subvector tal que uno de los          }
              { elementos (posiblemente x[down]              }
              { está ahora en x[j] (j es un           }
              { parámetro de salida) y:          }
              { 1. x[i]<=x[j] para down<=i<j          }
              { 2. x[i]<=x[j] para j<i<=up       }
              { x[j] está ahora en su            }
              { posición final                   }
     quick(lc,j-l);
              { ordena el subvector entre             }
              { x[down] y x[j-1] }
     quick(j+l,up)
              { ordena el subvector entre             }
              { x[j+l] y x[up]                   }
end {termina then begin}
```

El único problema que queda es el de obtener un mecanismo que implemente a rearrange, el cual permite que un elemento específico encuentre la posición que debe ser con respecto a los otros en el subvector. Observe que la forma mediante la cual este rearrange puede ser implementado es irrelevante al método de ordenamiento. Todo lo que se requiere desde el punto de vista del ordenamiento es que todos los elementos sean repartidos apropiadamente. En el ejemplo anterior, los elementos de cada uno de los dos subarchivos permanecen en el mismo orden relativo a como ellos estaban en el archivo original.

¿Qué tan eficiente es el ordenamiento rápido? Asuma que el tamaño del archivo n es una potencia de 2, tal que $n = 2^m$, o sea que m = log2 n. Asuma también que la posición apropiada para x[down] resulta ser siempre el punto medio del subvector. En este caso resultaría aproximadamente n comparaciones (realmente n - 1) en el primer paso, después del cual el archivo es separado en dos subarchivos cada uno de tamaño n/2 aproximadamente.

Para cada uno de estos dos archivos se presentarían aproximadamente n/2 comparaciones y resultaría un total de cuatro archivos cada uno de tamaño n/4. Cada uno de estos archivos requiere n/4 comparaciones, generando un total de n/8 subarchivos. Después de dividir los subarchivos n veces, entonces se tendrían n archivos de tamaño 1. Es decir, el número total de comparaciones para todo el archivo es aproximadamente

$$n + 2(n/2) + 4(n/4) + 8(n/8) + ...+ n(n/n)$$

o

$$n + n + n + n + ...+ n \text{ (m términos)}$$

comparaciones. Existen m términos debido a que el archivo es subdividido m veces. Es decir que el número total de comparaciones es O(nm) o O(n log n) (recuerde que m = log2 n). Es decir, que si estas propiedades describen el archivo, el método de ordenamiento rápido es O(n log n), el cual es relativamente eficiente.

Este análisis asume que el vector original y todos los subvectores resultantes no están ordenados, tal que x[down] siempre encuentra su posición apropiada en el punto medio del subvector. Asuma que estas condiciones no aplican y que el vector original está ordenado (o casi ordenado). Si por ejemplo, x[down] está en su posición correcta, el archivo original es subdividido en dos subarchivos de tamaño 0 y n - 1. Si se continúa con este proceso, se ordenarían un total de n - 1 subarchivos, el primero de tamaño n, el segundo de tamaño n - 1, el tercero de tamaño n - 2, y así sucesivamente. Asumiendo que se requieren k comparaciones para un archivo de tamaño k, el número total de comparaciones para ordenar todo el archivo sería

$$n + (n -1) + (n -2) +... + 2$$

lo cual es equivalente a O(n2). Igualmente, si el archivo original está ordenado en orden descendente, la posición final de x [down] es up y el archivo de nuevo se divide en dos subarchivos los cuales están muy desbalanceados (tamaños n - 1 y 0). Es decir, que el método de ordenamiento rápido tiene aparentemente la propiedad absurda de que trabaja mejor para archivos que están completamente desordenados y es peor para archivos que están casi completamente ordenados. Esta situación es precisamente la opuesta del ordenamiento de burbuja, el cual trabaja mejor para archivos que están casi ordenados y muy mal para archivos desordenados.

El análisis para el caso donde el tamaño del archivo no es una potencia entera de 2 es algo similar pero un poco más complejo; sin embargo, los resultados permanecen igual. Se puede demostrar que en promedio (para todos los archivos de tamaño n) el método de quicksort hace O(n log n) comparaciones.

El espacio requerido para el ordenamiento rápido depende del número de llamadas recursivas embebidas o del tamaño de la pila. Obviamente, la pila no puede ser mayor que el número de elementos en el archivo original. Que tan pequeño comparado con n llega a crecer la pila, depende del número de subarchivos generados y de su tamaño. El tamaño de la pila puede ser regulado si se almacenan en la pila siempre el mayor de los dos subvectores, y se aplica la rutina al más pequeño de ellos. Esto garantiza que todos los subvectores menores son subdivididos antes que los subvectores mayores, dando el efecto neto de tener el menor número de elementos en la pila en cualquier momento. La razón para esto es que un subvector pequeño puede ser dividido, un menor número de veces que un subvector grande. Por supuesto, que a lo último el subvector grande será procesado y subdividido, pero esto ocurrirá después de que los subvectores pequeños han sido ordenados.

Búsqueda en Arbol Binario.

En los ejemplos anteriores realizamos búsquedas en un archivo organizado como un arreglo o vector. En este caso consideremos la idea de organizar el archivo como un árbol.

En este método todos los descendientes izquierdos de un nodo son menores y los derechos son mayores. El recorrido simétrico del árbol genera la lista ordenada.

La ventaja de utilizar este método es que un árbol permite la búsqueda, inserción y eliminación en forma eficiente; en un vector requeriría que la mitad de los elementos sean movidos; por el otro lado la inserción o eliminación en árbol binario de búsqueda, requiere que se ajusten unos pocos punteros.

El nodo que utilizaremos tendrán la siguiente estructura:

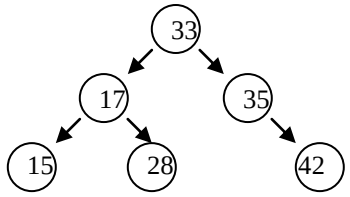
ID	
L	R

Ejemplo:

33	17	28	35	42	15

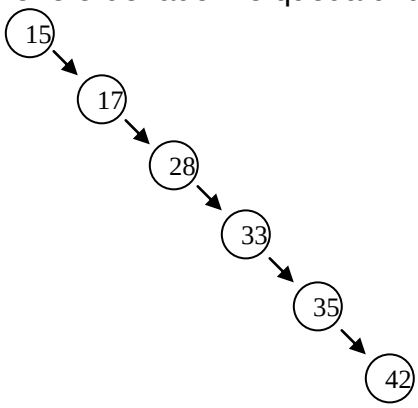
Inserción en ABB

Inserta si la búsqueda no tiene éxito. Si es mayor a la derecha, si es menor a la izquierda.



Si realizo un barrido simétrico obtengo el cjto. ordenado: 15 17 28 33 35 42

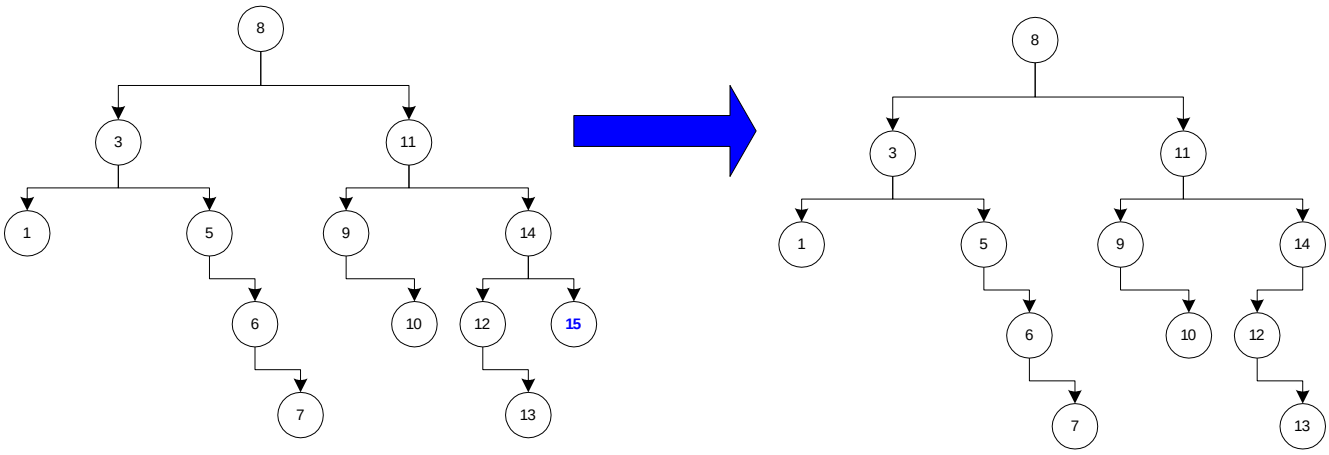
Si el conjunto viene ordenado me queda una lista, queda en forma lineal:



Eliminación en ABB

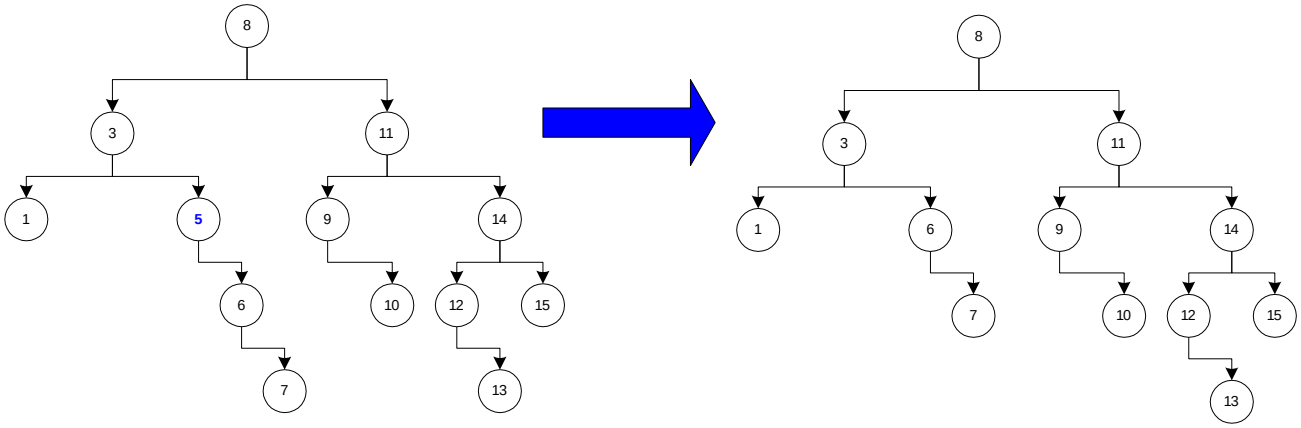
- Si no tiene hijos puede eliminarse sin ajuste.

Eliminación del nodo con la clave 15



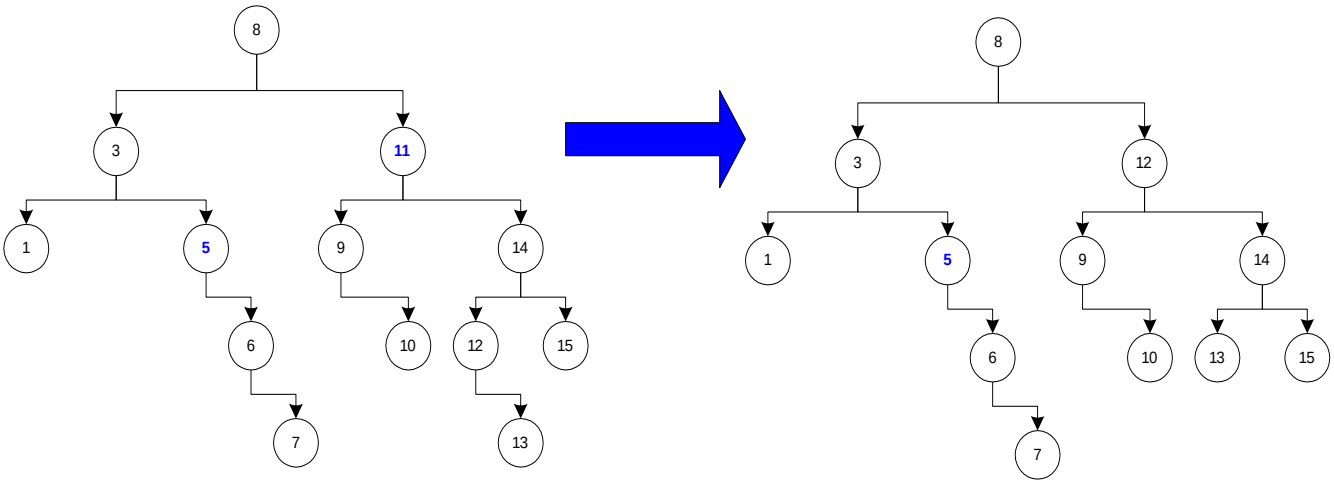
- Si tiene solo un subárbol, su hijo toma su lugar.

Eliminación del nodo con la clave 5



- Si tiene dos subárboles, su predecesor debe tomar su lugar, es decir el menor de los hijos del subárbol derecho puede tomar el lugar del padre.

Eliminación del nodo con la clave 11



Eficiencia

Método de ordenamiento Quicksort.

$n*\log_2 n$ ➔ $O(n \log n)$ mejor caso y promedio.

Mejor opción el pivote al centro y n/2 a cada lado y así sucesivamente.

Si viene ordenado ➔ $O(n^2)$ peor caso.

Método de Búsqueda mediante Arbol Binario.

$n \log_2 n$ ➔ $O(n \log n)$ mejor caso.

Si viene ordenado ➔ $O(n^2)$ peor caso.

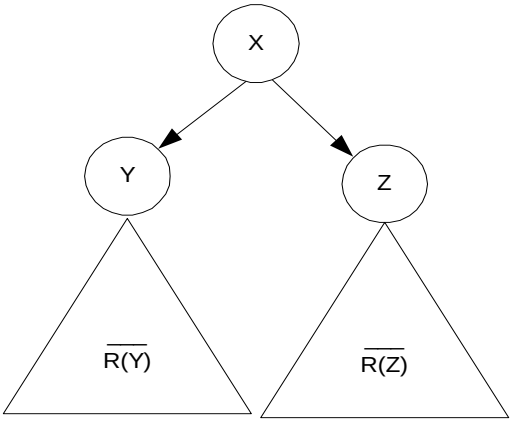
$n\sqrt{n}$ ➔ $O(n\sqrt{n})$ promedio.

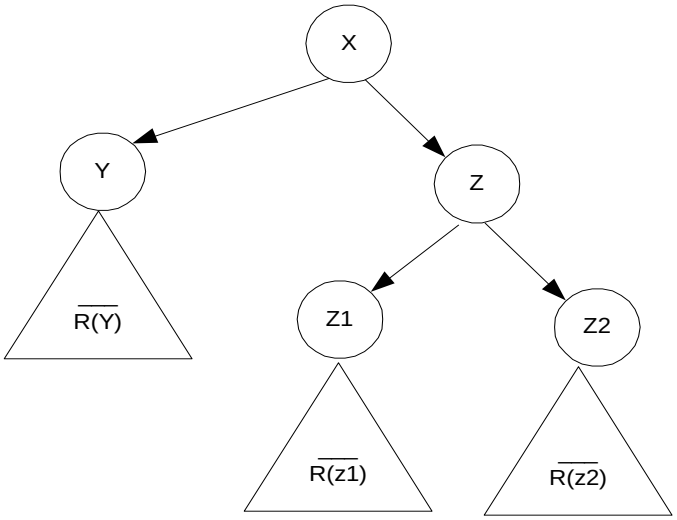
Arbol Balanceado

Sea x,y,z ∈ P y R(x) = {y,z} decimos que T(P,E) es Balanceado si para todo x se cumple que

$$\left|\ \left|\overline{R(y)}\right| - \left|\overline{R(z)}\right|\ \right| <= 1$$

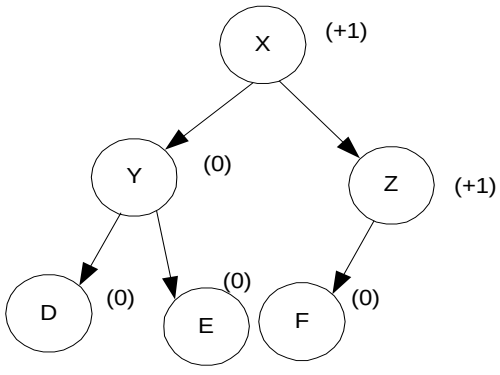
Como se plantea en la definición anterior, esta condición se tiene que cumplir para todos los nodos del árbol.



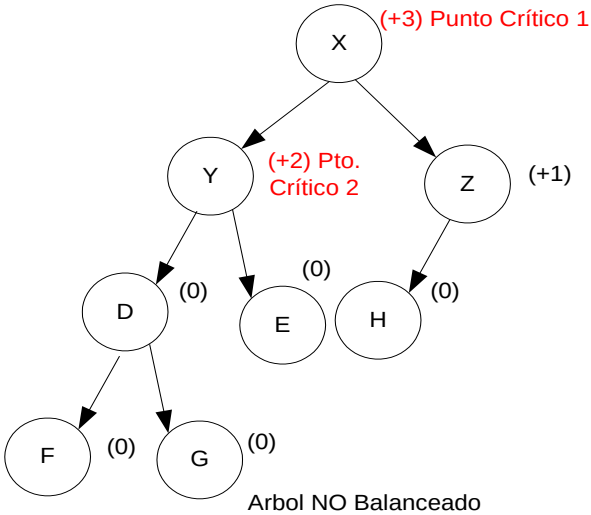


Llamaremos punto crítico al nodo en el cuál el árbol se desbalancea, ver los ejemplos en la próxima página.

Ejemplos de Arbol Balanceado:



Arbol Balanceado



Arbol NO Balanceado

Arbol AVL (Adelson, Velskii, Landis)

Debido al costo existente en mantener un árbol completamente balanceado se crearon los árboles AVL (nearly balanced trees), también llamados árboles balanceados en altura(height balanced trees).

Para poder definir este árbol primero tendremos que definir el término altura.

Altura: Es la longitud de paso máxima existente en el subárbol del cuál el nodo en cuestión es la raíz.

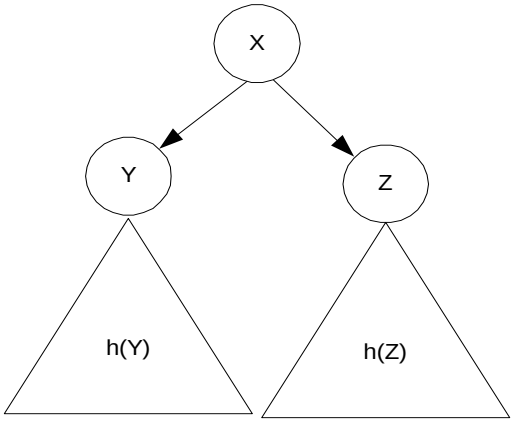
h(x) es una función que nos dá la altura del nodo x.

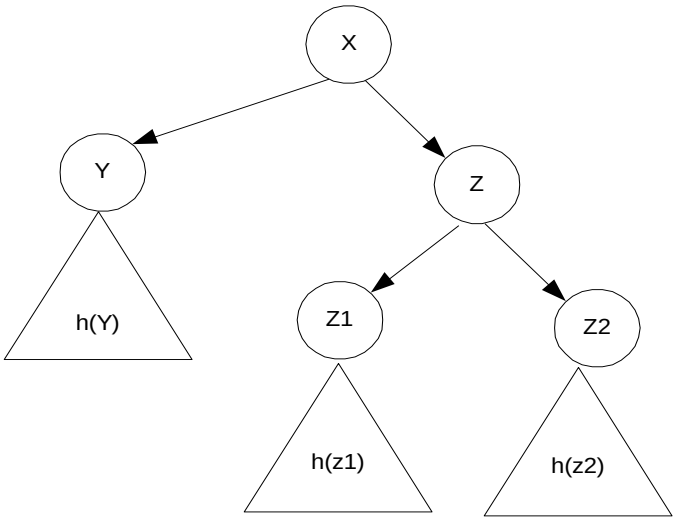
Definición de Árbol AVL

Sea x,y,z $\in$ P y R(x) = {y,z} decimos que T(P,E) es AVL si para todo x se cumple que

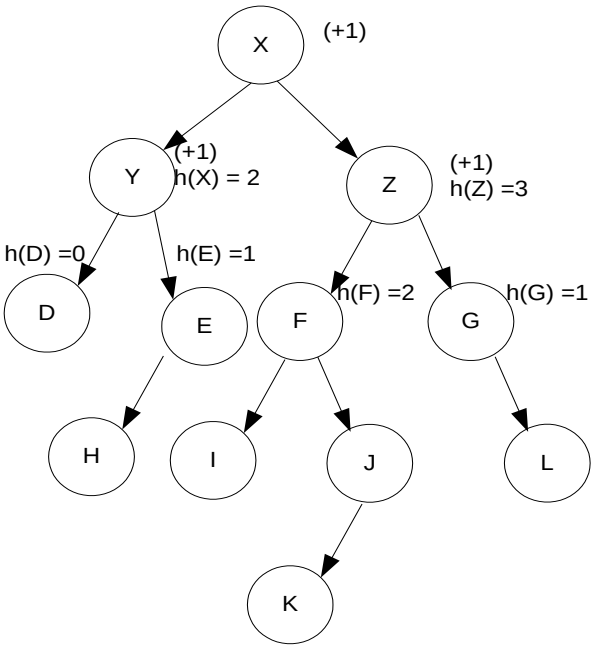
$$\left| h(y) - h(z) \right| <= 1$$

Como se plantea en la definición anterior, esta condición se tiene que cumplir para todos los nodos del árbol.

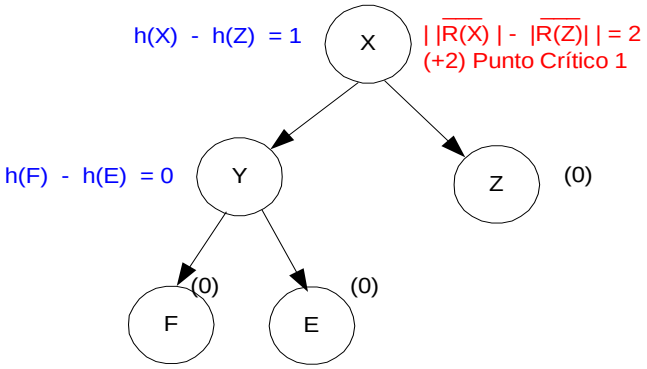




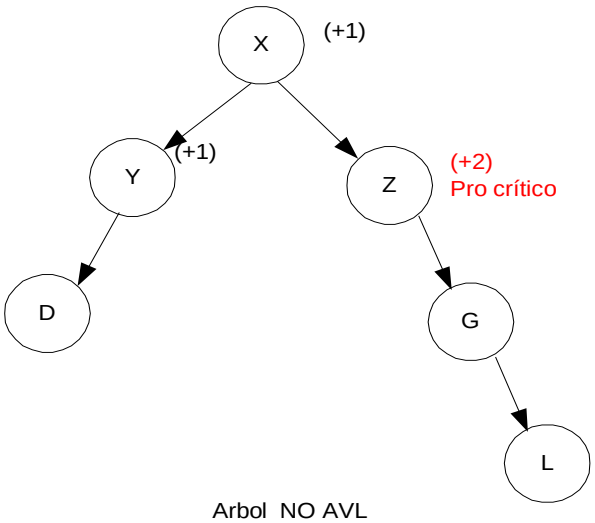
Ejemplos de Arbol AVL:



Arbol AVL



Arbol AVL
Arbol NO Balanceado



Arbol NO AVL

Rotación de árboles

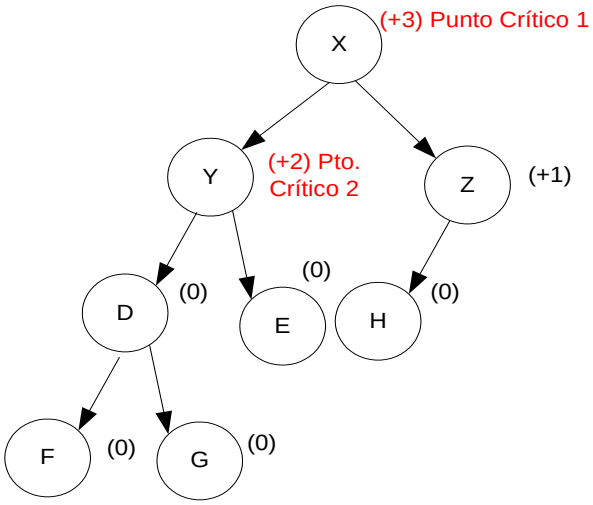
La rotación de árboles a izquierda o derecha es utilizada para balancear un árbol en un punto crítico determinado.

Si el árbol se encuentra desbalanceado a izquierda se deberá realizar una rotación a derecha, y viceversa.

El algoritmo de rotación garantiza que el barrido simétrico del árbol original desbalanceado, sea igual que el del nuevo árbol rotado.

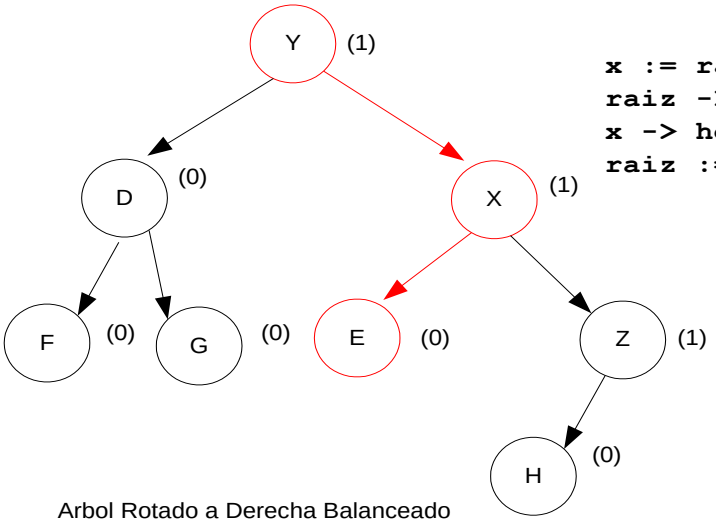
Rotación a Derecha

Tomando como ejemplo el siguiente árbol desbalanceado se aplicará una rotación tomando como raiz su punto crítico.



Arbol Desbalanceado a Izquierda

Aplicamos una ROTACION A DERECHA en el punto crítico X




```
x := raiz -> hizq
raiz -> hizq := raiz -> hizq -> hder
x -> hder := raiz
raiz := x
```

Arbol Rotado a Derecha Balanceado

Comparando los barridos simétricos del árbol desbalanceado original y el árbol balanceado producto de aplicar la rotación a derecha se observa:

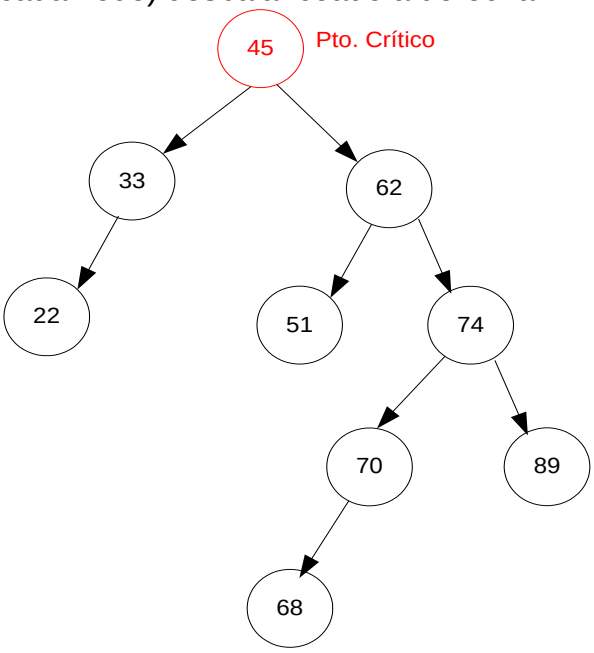
- BARRIDO SIMETRICO ARBOL ORIGINAL: <F,D,G,Y,E,X,H,Z>

- BARRIDO SIMETRICO ARBOL ROTADO: <F,D,G,Y,E,X,H,Z>

Ambos barridos son idénticos.

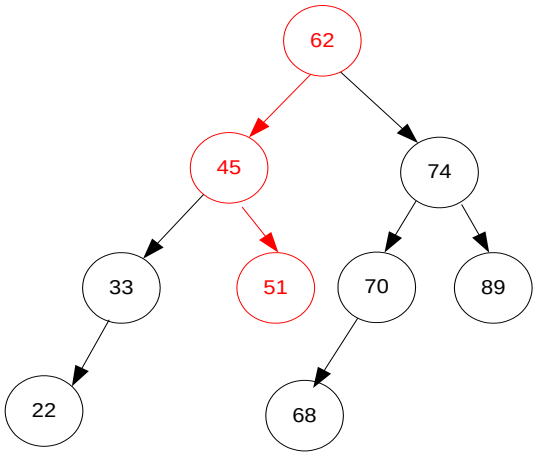
Rotación a Izquierda

Dado el siguiente Arbol Binario de búsqueda (menores a izquierda mayores a derecha de cada nodo) desbalanceado a derecha



Arbol Desbalanceado a Derecha

aplicaremos una ROTACION A IZQUIERDA con el objetivo de balancearlo.




```
x := raiz -> hder
raiz -> hder := raiz -> hder -> hizq
x -> hizq := raiz
raiz := x
```

Arbol Rotado a Izquierda Balanceado

Comparando los barridos simétricos del árbol binario de búsqueda desbalanceado original y el árbol balanceado producto de aplicar la rotación a izquierda se observa:

- BARRIDO SIMETRICO ARBOL ORIGINAL: <22,33,45,62,68,70,74,89>

- BARRIDO SIMETRICO ARBOL ROTADO: <22,33,45,62,68,70,74,89>

Observamos que ambos barridos son iguales y ambos árboles son ABB (Arboles binarios de búsqueda).