

I1 2018-2

1. Árboles AVL

- a) Queremos almacenar las claves 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9 en un árbol AVL inicialmente vacío. ¿Es posible insertar estas claves en el árbol en algún orden tal que **nunca** sea necesario ejecutar una rotación? Si tu respuesta es "sí", indica el orden de inserción y muestra al árbol resultante después de insertar cada clave. Si tu respuesta es "no", da un argumento convincente (p.ej., una demostración) de que efectivamente no es posible insertar las claves sin que haya que ejecutar al menos una rotación.

Sí es posible. La idea es hacer las inserciones de manera de mantener todo el tiempo la propiedad de balance; p.ej., procurar que el árbol se vaya llenando "por niveles". La primera clave que insertemos va a ser la raíz del árbol (ya que la idea es que no va a haber rotaciones). Por lo tanto, tiene que ser una clave k tal que el número de claves menores que k —que van a ir a parar al subárbol izquierdo— y el número de claves mayores que k —que van a ir a parar al subárbol derecho— sean parecidos. Si elegimos la clave 4, entonces hay 4 claves menores y 5 claves mayores (también podemos elegir la clave 5 y dejar 5 claves menores y 4 mayores). A continuación elegimos la raíz del subárbol izquierdo y la raíz del subárbol derecho (o viceversa). Para esto, aplicamos recursivamente la misma "regla", sobre las claves 0, 1, 2 y 3, para el subárbol izquierdo, y sobre las claves 5, 6, 7, 8 y 9, para el subárbol derecho; p.ej., insertamos 2 y luego 7. Repitiendo la estrategia, luego insertamos 1, 3, 6 y 8, y finalmente 0, 5 y 9. Así, un orden de inserción posible es 4, 2, 7, 1, 3, 6, 8, 0, 5, 9.

- b) Considera la inserción de una clave x en un árbol AVL T . Definimos la *ruta de inserción* de x como la secuencia de nodos, empezando por la raíz de T , cuyas claves son comparadas con x durante la inserción. El **procedimiento de rebalanceo** —una vez hecha la inserción— primero sube (de vuelta, desde el nodo recién insertado) por la ruta de inserción examinando cada nodo r que está en la ruta: si r es raíz de un (sub)árbol AVL-balanceado, entonces se sigue subiendo; de lo contrario, se ejecuta la rotación que vimos en clase en torno a la arista r —hijo izquierdo o r —hijo derecho, según corresponda. Muestra los árboles AVL que se van formando al insertar las claves 3, 2, 1, 4, 5, 6, 7 y 16, en este orden, en un árbol AVL inicialmente vacío.

Inicialmente se genera el árbol con raíz 3, y se agrega como hijo izquierdo el nodo con key 2. Luego se inserta a la izquierda de 2 la clave 1 y se hace una rotación simple en la que queda 2 como raíz y sus hijos 1 y 3 a la izquierda y derecha respectivamente.

Al insertar el 4 este queda a la derecha del 3, y luego al insertar el 5 este queda a la derecha del 4 y se hace una rotación simple entre 3, 4 y 5 y queda 4 como hijo derecho de 2 y 3 y 5 como hijos de 4 a la izquierda y derecha respectivamente.

Luego se inserta el 6 a la derecha del 5 y esto crea un desbalance en el nodo raíz, por lo que se hace una rotación simple entre 2, 4 y 5 quedando como raíz el nodo 4. Del nodo cuatro cuelga a la izquierda el nodo 2, el cual tiene a su izquierda el nodo 1 y a su derecha el nodo 3. A la derecha de 4 cuelga el nodo 5, el cual tiene el nodo 6 como hijo derecho.

Al insertar el nodo 7 este queda a la derecha del nodo 6 y se hace una rotación simple entre los nodos 5, 6 y 7, quedando el nodo 6 como padre de 5 y 7.

Al insertar el nodo 16 queda como hijo derecho del nodo 7 y no hay rotación.

2. Árboles de búsqueda binarios (no necesariamente balanceados)

En clase vimos cómo se elimina una clave de un árbol de búsqueda binario (ABB). Eliminar la clave cuando el nodo que ocupa no tiene hijos o tiene sólo un hijo, Ti o Td , es fácil.

- a) Es más difícil eliminar una clave cuando el nodo que ocupa tiene ambos hijos, Ti y Td ; describe las acciones correspondientes. Esta forma de eliminación se llama **eliminación por copia**. [1 pt.]

Se busca la clave sucesora (o predecesora) de la clave eliminada, y se la coloca, junto con su descendencia, en lugar de ésta (de la eliminada); luego, se elimina la clave sucesora, que a lo más tiene un hijo.

- b) ¿Es la eliminación por copia "conmutativa" en el sentido de que eliminar x y luego y de un ABB deja el mismo árbol que eliminar y y luego x ? Demuestra que lo es o da un contraejemplo. [1 pt.]

Contraejemplo: Supongamos que al eliminar un nodo con dos hijos, lo reemplazamos por su sucesor. Considéremos una raíz con clave 5, y dos hijos, con claves 3 y 11, respectivamente; el nodo con clave 11 a su vez tiene un hijo izquierdo con clave 7. Si eliminamos el nodo con clave 5 (dos hijos) y luego el nodo con clave 3 (hoja), dejamos un abb —raíz 7 e hijo derecho 11— distinto que si eliminamos el nodo con clave 3 (hoja) y luego el nodo con clave 5 (ahora sólo un hijo) —raíz 11 e hijo izquierdo 7.

- c) Otra forma de eliminar una clave cuyo nodo tiene ambos hijos es **eliminación por mezcla**: el nodo es ocupado por su (hijo y) subárbol izquierdo, Ti , mientras que su subárbol derecho, Td , se convierte en el subárbol derecho del nodo más a la derecha de Ti . Justifica que esta eliminación respeta las propiedades de ABB. [2 pts.]

A partir de la regla para insertar claves, que, a su vez, cumple la propiedad fundamental de ABB, sabemos que en el proceso de inserción de cualquiera de las claves que están en Ti o Td —llamemos k a una clave cualquiera en Ti o Td — pasamos por la clave —llámémosla j — que estamos eliminando, y que, por lo tanto, k podría haber ido a parar al lugar de j , posición en la que habría cumplido la propiedad de ABB con respecto al resto del árbol. Por lo tanto, poner el subárbol Ti en la posición que ocupaba el nodo con la clave j es válido.

La pregunta entonces es, ¿qué hacemos con Td ? De nuevo, por la propiedad de ABB, las claves de Td son todas mayores que las claves de Ti . La única posición que corresponde a claves mayores que todas las claves de Ti , pero al mismo tiempo menores que las otras claves del árbol mayores que j es como hijo derecho de la clave más a la derecha de Ti —llámémosla m ; obviamente, esta posición está "desocupada": m sólo puede ser la clave más a la derecha de Ti si no tiene hijo derecho.

- d) Muestra con ejemplos que la eliminación por mezcla puede tanto aumentar como reducir la altura del árbol original. [2 pts.]

Para simplificar (y generalizar un poco), eliminamos la raíz del árbol. Entonces, Ti "sube" a esta posición y agregamos Td como hijo derecho del nodo más a la derecha de Ti .

La altura del árbol original, T , era $H(T) = \max\{H(Ti), H(Td)\} + 1$. La altura del nuevo árbol, T' , puede ser desde $H(T') = H(Ti)$ hasta $H(T') = H(Ti) + H(Td)$, dependiendo de la profundidad del nodo más a la derecha de Ti . En el primer caso, $H(T')$ es claramente menor que $H(T)$; en el segundo, $H(T')$ claramente puede ser mayor que $H(T)$.

EXAMEN 2018-2

1. Árboles binarios de búsqueda

Las claves en un árbol binario de búsqueda están ordenadas. Por lo tanto, debería ser fácil encontrar la k -ésima clave más pequeña o determinar cuántas están en el rango $[a, b]$. Explica cómo podría hacerse:

- a) ¿Qué información adicional habría que mantener en cada nodo?

En cada nodo x podemos almacenar en un campo adicional $x.size$ el número total de nodos que tiene el árbol cuya raíz es x , incluyendo a x . Así, p.ej., una hoja h tiene $h.size = 1$, y si el árbol almacena 1000 claves, entonces la raíz r del árbol tiene $r.size = 1000$.

- b) Describe los algoritmos necesarios para responder las consultas mencionadas más arriba.

Suponemos que cada nodo x tiene punteros $left$, $right$ y p , a sus hijos izquierdo y derecho y a su padre.

k -esima(x, k): *esta operación es $O(\text{altura del árbol})$*

```
r = x.left.size + 1
if k == r:
    return x
else: if k < r:
    return k-esima(x.left, k)
else:
    return k-esima(x.right, k-r)
```

Para determinar cuántas claves están en el rango $[a, b]$, hay que saber qué posición ocupan a y b en un orden lineal de todas las claves (recorrido *inorder* del árbol T) —esto es, el ranking de a y el ranking de b — y luego restar, $\text{ranking}(b, T) - \text{ranking}(a, T)$:

ranking(x, T): *esta operación también es $O(\text{altura del árbol})$*

```
r = x.left.size + 1
y = x
while y != T.root:
    if y == y.p.right:
        r = r + y.p.left.size + 1
    y = y.p
return r
```

- c) ¿Cuánto cuesta mantener la información adicional? En particular, si se hace una inserción en el árbol; y si se hace una eliminación. Justifica.

En ambos casos, el costo es $O(\text{altura del árbol})$. En una inserción, hay que incrementar $x.size$ para cada nodo x en el camino desde la raíz al punto de inserción; el nuevo nodo (que es una hoja) tiene $size = 1$.

En una eliminación, si eliminamos una hoja o un nodo con un solo hijo, hay que decrementar $x.size$ para cada nodo x en el camino desde el parente del nodo eliminado hasta la raíz; si eliminamos un nodo con dos hijos, primero lo reemplazamos por su sucesor y luego eliminamos el sucesor (de su posición original).

I1 2018-1

1. Árboles AVL

a) Considera un árbol AVL que almacena las claves 1, ..., 6 de la siguiente manera: 4 está en la raíz, 2 y 5 son sus hijos; 1 y 3 son hijos de 2; y 6 es hijo de 5. En este árbol, inserta las siguientes claves, en el orden dado: 7, 16, 15, 14 y 13.

En cada caso, muestra el árbol resultante justo después de la inserción, pero antes de rebalancear el árbol (si fuera necesario), e indica si es necesario o no rebalancear el árbol. Si es necesario rebalancear el árbol, entonces explica por qué es necesario, identifica la acción que hay que realizar, y muestra el árbol resultante después de realizarla.

Respuesta

Denotaremos las aristas como “nodo padre”–“nodo hijo”. La inserción de 7 produce un desbalance en 5 y requiere una rotación a la izquierda en torno a la arista 5-6 [1 pt.]. La inserción de 16 no produce desbalances [0.5 pts.]; pero la de 15 produce un desbalance en 7 y exige una doble rotación: a la derecha en torno a 16-15, y a la izquierda en torno a 7-15 [1.5 pts.]. La inserción de 14 produce un desbalance en 6 y también exige una rotación doble: a la derecha en torno a 15-7, y a la izquierda en torno a 6-7 [2 pts.]. Finalmente, la inserción de 13 produce un desbalance en 4 (la raíz) y exige una rotación en torno a 4-7.

El árbol resultante tiene a 7 en la raíz, con hijos 4 y 15. Los hijos de 4 son 2 y 6, los hijos de 2 son 1 y 3, y el único hijo de 6 es 5. Los hijos de 15 son 14 y 16, y el único hijo de 14 es 13.

b) Considera la inserción de una clave x en un árbol T . Definamos la *ruta de inserción* de x como la secuencia de nodos, empezando por la raíz de T , cuyas claves son comparadas con x durante la inserción. Como vimos en clase, el algoritmo de rebalanceo primero sube (de vuelta) por la ruta de inserción examinando cada nodo que está ahí. Con respecto a esta “subida” (es decir, todavía antes de rebalancear propiamente), explica claramente:

- en qué consiste realmente “examinar” el nodo;
- qué casos pueden darse y qué hay que hacer en cada uno de ellos (para mantener el nodo actualizado); y
- cuándo se detiene.

Por último, al momento de detenerse la subida, y ya actualizado el nodo correspondiente, una posibilidad es que este nodo tenga el rol de “pivot” en el proceso de rotación que habría que hacer a continuación. Como también vimos en clase, la rotación necesaria puede ser simple o doble:

- ¿cómo sabe el algoritmo cuál rotación hay que aplicar?

Respuestas

Examinar el nodo consiste en revisar el valor del balance del nodo.

El valor del balance puede ser -1 , 0 o $+1$. Si es 0 , entonces hay que cambiarlo a -1 o $+1$, según corresponda, y seguir subiendo; solo en este caso se sigue subiendo.

Si, en cambio, el valor del balance del nodo examinado x es -1 o $+1$, entonces hay dos posibilidades, dependiendo del hijo de x desde el cual, al subir, se llegó a x : se pudo haber llegado a x por el lado que corrige el -1 o $+1$, y lo cambia a 0 ; o se pudo haber llegado por el lado que aumenta el desbalance (dejándolo temporalmente en -2 o $+2$):

- El primer caso es cuando llegamos a x desde su hijo derecho y el balance de x es -1 , o cuando llegamos a x desde su hijo izquierdo y el balance de x es $+1$. En este caso, solo hay que cambiar el balance de x a 0 (y no se sigue subiendo).
- El segundo caso es cuando llegamos a x desde su hijo derecho y el balance de x es $+1$, o cuando llegamos a x desde su hijo izquierdo y el balance de x es -1 . Lo que ocurre aquí es que x queda desbalanceado según el criterio de balance AVL; x es el pivote que vimos en clase. En este caso, hay que efectuar un rebalanceo por la vía de una rotación, ya sea simple o doble (y tampoco se sigue subiendo).

Así, la “subida” se detiene cuando encuentra un nodo con balance -1 o $+1$. Entonces, se procede a actualizar el balance, tal como está explicado.

La “subida” también se detiene si se llega a la raíz.

Finalmente, en el caso en que hay que realizar una rotación (el “segundo caso”), para saber cuál es la rotación que hay que realizar, hay que recordar los dos últimos nodos examinados antes de llegar a x , es decir, el hijo y y el nieto z de x : si y y z son ambos hijos izquierdos o son ambos hijos derechos, entonces la rotación necesaria es simple; en cambio, si uno es hijo izquierdo y el otro es hijo derecho, entonces la rotación necesaria es doble.

I2 2018-1

1. Árboles de búsqueda balanceados

a) Como vimos en clase, la inserción de un nodo en un árbol rojo-negro se puede dividir en tres casos; en todos los casos, el nodo x recién insertado se pinta inicialmente de rojo:

- 1) el padre p de x es negro → estamos listos
- 2) el padre p de x es rojo, pero el hermano q de p es negro (si p no tiene hermano, lo suponemos negro) → hacemos una o dos rotaciones y le cambiamos el color a dos nodos [ustedes tienen que saber los detalles de este caso]
- 3) el padre p de x es rojo, y el hermano q de p también es rojo (es decir, el nodo recién insertado, su padre y su tío son todos rojos) → como el abuelo r de x necesariamente es negro, cambiamos los colores de r y de sus hijos p y q ; ahora r es rojo (y sus hijos p y q son negros), por lo que hay que revisar el color del padre s de r :
 - s es negro → estamos listos
 - s es rojo → repetimos el caso 2) o el caso 3), según corresponda, pero ahora con s en lugar de p , y el hermano t de s en lugar de q

El caso 3) significa que potencialmente podemos llegar de vuelta hasta la raíz del árbol. Una manera de evitar tener que hacer este recorrido “de vuelta” es ir preparando el árbol a medida que vamos bajando (desde la raíz, cuando estamos buscando el punto en que hay que hacer la inserción), de modo de garantizar que q no será rojo —algoritmo de inserción *top-down*:

Al ir bajando, cuando encontramos un nodo U que tiene dos hijos rojos, cambiamos los colores de U (a rojo) y de sus hijos (a negro); esto producirá un problema solo si el padre V de U también es rojo, en cuyo caso aplicamos el caso 2) anterior.

Muestra la ejecución de este algoritmo de inserción *top-down* al insertar la clave 22 en el siguiente árbol rojo-negro: la raíz tiene la clave 35; sus hijos, las claves 30 (rojo) y 42 (negro); los dos hijos negros de 30 tienen las claves 25 y 33; los dos hijos rojos de 42 tienen las claves 40 y 45; y los dos hijos rojos de 25, tienen las claves 20 y 27.

Respuesta (estos son, más o menos, los pasos importantes)

Al bajar desde la raíz, con clave 35, no encontramos ningún caso especial hasta que llegamos al nodo negro con clave 25, que tiene ambos hijos rojos: 20 y 27. **[0.5 pts]**

Entonces aplicamos la nueva regla: cambiamos los colores de 25 a rojo y de sus dos hijos, 20 y 27, a negros. **[0.5 pts.]**

Como el padre de 25 también es rojo (el nodo con clave 30), aplicamos el caso 2: hacemos una rotación simple a la derecha de la arista 35–30 —con lo que queda 30 como raíz (roja), con hijos 25 y 35 negros— e intercambiamos colores entre padre e hijos, dejando a 30 negro, y a 25 y 35 ambos rojos. **[1.5 pt.]**

Ahora el nodo con clave 25 tiene ambos hijos —con claves 20 y 27— negros; y como 20 es una hoja, insertamos allí el nuevo nodo con clave 22, que pintamos de rojo. **[0.5 pts.]**

- b)** Determina un orden en que hay que insertar las claves 1, 3, 5, 8, 13, 18, 19 y 24 en un árbol 2-3 inicialmente vacío para que el resultado sea un árbol de altura 1, es decir, una raíz y sus hijos.

Respuesta

Un árbol 2-3 con una raíz y sus hijos tiene a lo más 4 nodos y puede almacenar a lo más 8 claves (dos claves por nodo). Como son exactamente 8 claves las que queremos almacenar, éstas tienen que quedar almacenadas de la siguiente manera: la raíz tiene las claves 5 y 18; el hijo izquierdo, 1 y 3; el hijo del medio, 8 y 13; y el hijo derecho, 19 y 24. **[1 pt.]**

Para lograr esta configuración final hay varias posibilidades; aquí vamos a ver una. **[2 pts.]**

Podemos insertar primero las claves 1, 5 y 13, en cualquier orden, con lo cual queda 5 en la raíz, y 1 y 13 como hijos izquierdo y derecho. (En vez de 1 puede ser 3 y en vez de 13 puede ser 8. Por otra parte, en vez de empezar con 1, 5 y 13, es decir, empezar "por la izquierda", podríamos empezar por la derecha con 8-13, 18 y 19-24.) A partir de ahora, podemos insertar 3 en cualquier momento.

Ahora tenemos que conseguir que 18 quede en la raíz, junto con 5. Insertamos 18, que va a acompañar a 13, y a continuación insertamos 24, que va al mismo nodo de 13 y 18; como este nodo tiene ahora tres claves —13, 18 y 24 (lo que no puede ser)— lo sepáramos en dos nodos con las claves 13 y 24, respectivamente, y subimos la clave 18. Ahora podemos insertar 8 y 19, en cualquier orden.

EXAMEN 2017-1

1. Para cada una de las siguientes afirmaciones, diga si es *verdadera* o *falsa*, **siempre justificando** su respuesta.
 - d) Sea A un árbol rojo-negro en donde cada rama tiene n nodos negros y n nodos rojos. Si al insertar una clave nueva en A se obtiene el árbol A' , entonces cada rama de A' tiene $n + 1$ nodos negros. **Respuesta:** Verdadero. Un árbol rojo negro como A corresponde a un árbol 2-4 “completamente saturado”; es decir, uno que contiene nodos con 3 claves. Al insertar una clave nueva, el árbol 2-4 aumentará su altura en 1.
 - f) Sea A un árbol AVL y ℓ_1 y ℓ_2 los largos de dos ramas de A . Entonces $|\ell_1 - \ell_2| \leq 1$. **Respuesta:** Falso. Basta dar un contraejemplo
 - g) La operación de inserción en un árbol AVL con n datos realiza a lo más una operación *restructure* pero toma tiempo $O(\log n)$. **Respuesta:** Verdadero, puesto que la inserción debe revisar que el balance esté correcto a lo largo de la rama donde se insertó el dato. Y esa rama tiene tamaño $O(\log n)$.
 - h) Si A es un ABB y n es un nodo de A , el *sucesor* de n no tiene un hijo izquierdo. **Respuesta:** Falso. La propiedad no se cumple en general cuando n no tiene un hijo derecho.

I1 2016-2

2. Describe un algoritmo para eliminar una clave de un árbol 2-3; el algoritmo recibe como parámetros un puntero a la raíz del árbol y la clave que se va a eliminar. Por supuesto, tu algoritmo debe dejar como resultado un árbol 2-3. (En este problema, el algoritmo se puede describir en prosa y con dibujos.)

La idea general es primero encontrar la clave en el árbol [**a**) describe esta parte del algoritmo en términos de pasos más básicos: **1**].

Si la clave está en una hoja, entonces simplemente la eliminamos.

Si la clave está en la raíz o en un nodo interior, entonces la reemplazamos por la clave predecesora o la clave sucesora [**b**) describe esta parte del algoritmo en términos de pasos más básicos: **2**],

... y eliminamos (recursivamente) esta clave del árbol (hasta llegar a una hoja).

Si la hoja es un nodo 3, entonces estamos listos;

... si es un nodo 2, entonces hay que ser más creativos: considera que se produjo un "hoyo" en árbol y hay que hacerlo subir por el árbol hasta que pueda ser eliminado [**c**) describe esta parte del algoritmo en términos de pasos más básicos: **3**].

Respuestas:

- a) Buscamos la clave en el nodo que estamos mirando; el primer nodo que miramos es obviamente la raíz del árbol. Si encontramos la clave en este nodo, entonces pasamos a **b**); de lo contrario, tenemos que descender a uno de los hijos del nodo y, recursivamente, buscar ahí. En este caso, usamos el hecho de que las claves del nodo están ordenadas para decidir a cuál hijo tenemos que descender: si la clave buscada es menor que la menor de las claves del nodo, entonces bajamos al hijo izquierdo del nodo; si la clave buscada es mayor que la mayor de las claves del nodo, entonces bajamos al hijo derecho del nodo; si la clave buscada está entre la menor y la mayor claves del nodo (en un nodo 3), entonces bajamos al hijo del medio.
- b) Como es un nodo interior, entonces la clave predecesora (o la sucesora, se puede elegir cualquiera) está siempre en una hoja, más abajo en el árbol: la clave predecesora/sucesora es la más grande/pequeña de las claves almacenadas en el subárbol izquierdo/derecho (respecto de la clave que estamos eliminando). Para encontrarla (supongamos que buscamos la predecesora), bajamos a la raíz del subárbol correspondiente; si es una hoja, entonces es la clave más a la derecha en la hoja; de lo contrario, bajamos por el puntero más a la derecha del nodo y buscamos recursivamente ahí. Esta clave predecesora la ponemos en el lugar de la clave que queremos eliminar, sacándola de la hoja en la que estaba.

c) Si el "hoyo" es hijo de un nodo 2, con clave X, y tiene como hermano un nodo 2, con clave Y, entonces reestructuramos este subárbol de la siguiente manera: ponemos el "hoyo" en el lugar en que está X y hacemos que X y Y formen un nodo 3 hijo del "hoyo"; con esto, el "hoyo" sube un nivel y hay que lidiar con él en este nuevo nivel.

Si el "hoyo" es hijo de un nodo 2, con clave X, y tiene como hermano un nodo 3, con claves Y y Z, entonces reestructuramos así: Y queda en el lugar de X, X queda en el lugar del "hoyo", y uno de los hijos del nodo 3 original pasa a ser hijo de X. En este caso, el "hoyo" es efectivamente eliminado (no sube).

Si el "hoyo" es hijo de un nodo 3 y su hermano inmediato —a su derecha o su izquierda— es un nodo 2 (el otro hermano puede ser un nodo 2 o un nodo 3), entonces cambiamos el padre por un nodo 2 (dejamos en el padre sólo una de las dos claves originales) y bajamos la otra clave, que pasa a formar un nodo 3 con el hermano inmediato del "hoyo". El "hoyo" es eliminado.

Finalmente, si el hermano inmediato del "hoyo" es un nodo 3, entonces separamos este hermano en dos nodos 2, de modo que uno de ellos reemplaza al "hoyo"; en este proceso, hay que redistribuir las claves de los dos nodos 3. El "hoyo" es eliminado.

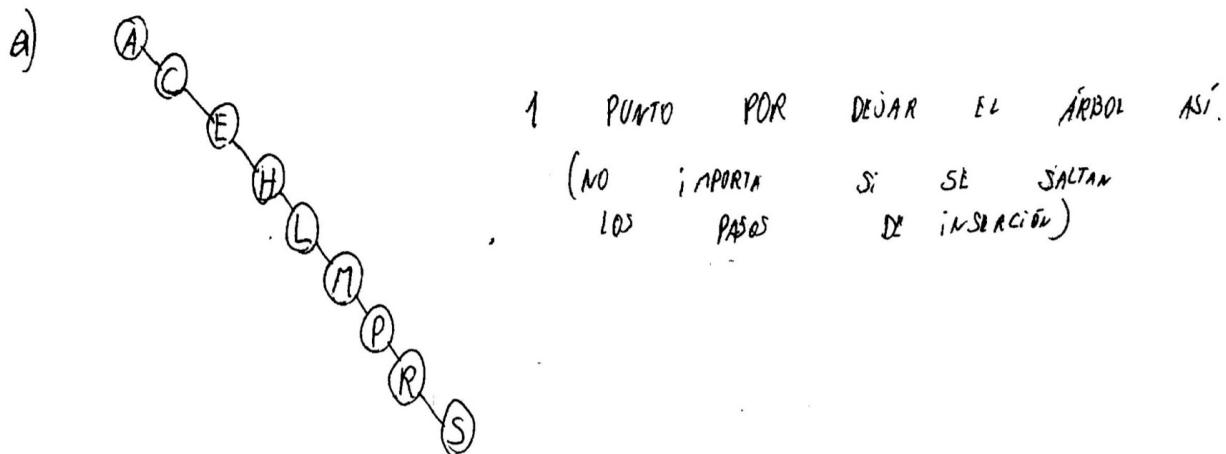
3. Considera un **árbol de búsqueda** inicialmente vacío; y considera las siguientes 9 letras como claves a ser insertadas en el árbol: A, C, E, H, L, M, P, R y S. Ejecuta la inserción, letra por letra y en el orden dado, para cada uno de los siguientes tipos de árbol:

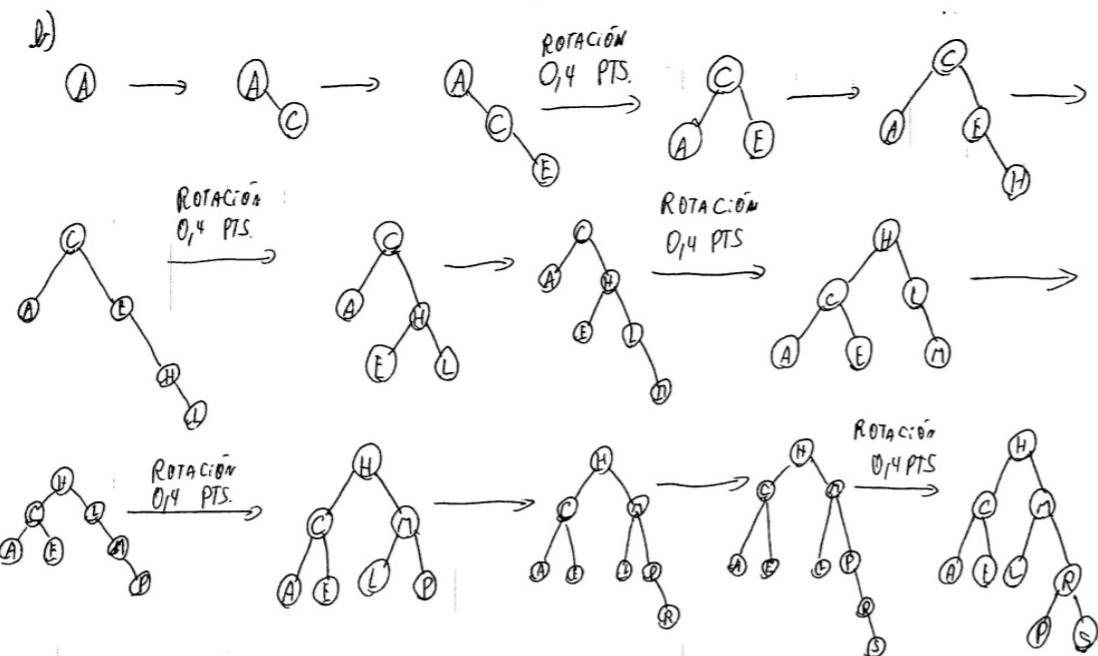
a) [1] Un árbol de búsqueda binario sin propiedades de balance.

b) [2.5] Un árbol AVL.

c) [2.5] Un árbol 2-3.

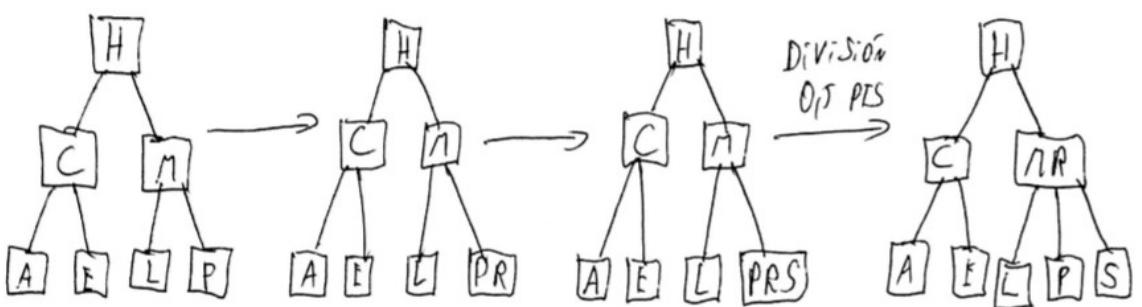
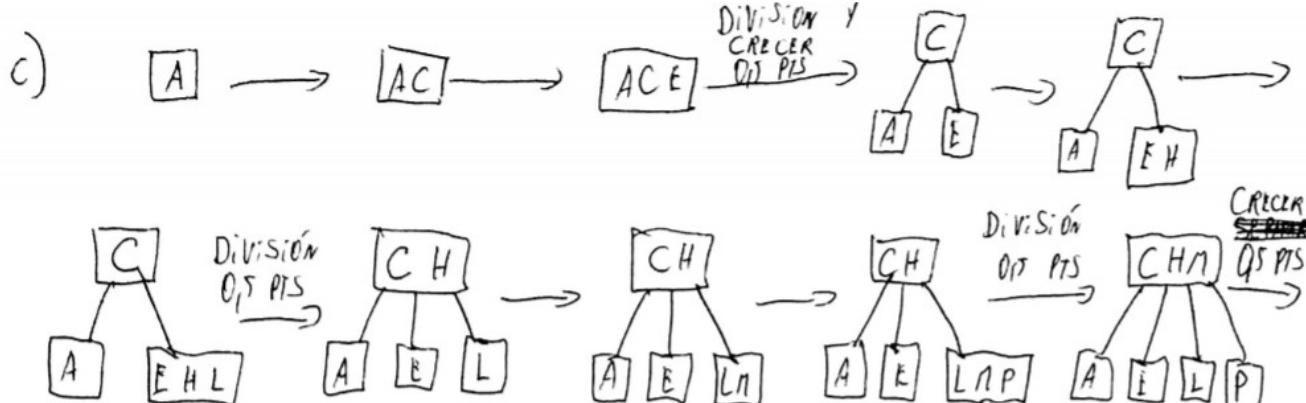
PAUTA PREGUNTA 3.





$0,4 \cdot 5$ + $0,5 = 2,5$ PTS.
ROTACIONES | HACER TODAS LAS INSERCIOS EN LAS HOJAS CORRESPONDIENTES

SE PUEDEN HACER LAS INSERCIOS MIENTRAS QUEDA CLARO SALTAR ROTACIONES PASOS COMO E



Si SE SALTAN PASOS PERO SI ENTENDENOS, ESTA BIEN.

I1 2015-2

4. Sobre árboles binarios de búsqueda (ABB's)

a) Describe un algoritmo de certificación para ABB's, suponiendo que sabemos que el árbol es binario.

Es decir, tu algoritmo debe verificar que la propiedad de ABB se cumple. Tu algoritmo debe correr en tiempo $O(\text{número de nodos en el árbol})$.

P.ej., un algoritmo recursivo que, a partir de un nodo x , verifica que el hijo izquierdo de x sea abb, que el hijo derecho de x sea abb, y que la clave más grande del hijo izquierdo sea menor que la clave de x y que ésta sea menor que la clave más pequeña del hijo derecho.

b) Supongamos que la búsqueda de una clave k en un ABB termina en una hoja. Consideremos tres conjuntos: A , las claves a la izquierda de la ruta de búsqueda; B , las claves en la ruta de búsqueda; y C , las claves a la derecha de la ruta de búsqueda. Tomemos tres claves: $a \in A$, $b \in B$ y $c \in C$. ¿Es cierto o es falso que $a < b < c$? Justifica.

Falso. Para justificar, basta un ejemplo: Supongamos que la búsqueda pasa por un nodo con clave 7, baja al hijo derecho con clave 13 y baja al hijo derecho con clave 17, que es una hoja; así, las últimas tres claves en la ruta de búsqueda son 7, 13 y 17. Entonces, basta que el nodo con clave 13 tenga un hijo izquierdo, cuya clave debe ser mayor que 7 (y menor que 13), p.ej., 11; esta clave queda a la izquierda de la ruta de búsqueda, pero no es menor que cualquier clave en la ruta de búsqueda, en particular, no es menor que 7.

c) ¿Es la operación de eliminación "comutativa" en el sentido de que eliminar x y luego y de un ABB deja el mismo árbol que eliminar y y luego x ? Demuestra que efectivamente lo es o da un contraejemplo.

Contraejemplo: Supongamos que al eliminar un nodo con dos hijos, lo reemplazamos por su sucesor. Consideremos una raíz con clave 5, y dos hijos, con claves 3 y 11, respectivamente; el nodo con clave 11 a su vez tiene un hijo izquierdo con clave 7. Si eliminamos el nodo con clave 5 (dos hijos) y luego el nodo con clave 3 (hoja), dejamos un abb —raíz 7 e hijo derecho 11— distinto que si eliminamos el nodo con clave 3 (hoja) y luego el nodo con clave 5 (ahora sólo un hijo) —raíz 11 e hijo izquierdo 7.

I2 2015-2

1. a) ¿Es todo árbol AVL un árbol rojo-negro? Justifica.

Sí. Hay justificar que en un árbol AVL la ruta (simple) más larga de la raíz a una hoja no tiene más del doble de nodos que la ruta (simple) más corta de la raíz a una hoja. Esto es así:

- El árbol AVL más “desbalanceado” es uno en el que todo subárbol derecho es más alto (en uno) que el correspondiente subárbol izquierdo (o vice versa); en tal caso, el número de nodos en la ruta más larga crece según $2h$, y el de la ruta más corta, según $h+1$, en que h es la altura del árbol.
- Así, para cualquier ruta (simple) desde la raíz a una hoja, sea d la diferencia entre el número de nodos en esa ruta y el número de nodos en la ruta más corta. Si todos los nodos de la ruta más corta son negros, entonces los otros d nodos deben ser rojos: pintamos la hoja roja y, de ahí hacia arriba, nodo por medio hasta completar d nodos rojos.

b) ¿Cuántos cambios de color y cuántas rotaciones pueden ocurrir a lo más en una inserción en un árbol rojo-negro? Justifica.

Recuerda que al insertar un nodo, x , lo insertamos como una hoja y lo pintamos rojo. Si el padre, p , de x es negro, terminamos. Si p es rojo, hay dos casos: el hermano, s , de p es negro; s es rojo. Si s es negro, hacemos algunas rotaciones y algunos cambios de color. Si s es rojo, sabemos que el padre, g , de p y s es negro; entonces, cambiamos los colores de g , p y s , y revisamos el color del padre de g .

Hacemos $O(\log n)$ cambios de color y $O(1)$ rotaciones. Como dice arriba, esto ocurre solo cuando p es rojo.

Si s es negro, hacemos exactamente una o dos rotaciones, dependiendo de si x es el hijo izquierdo o el hijo derecho de su padre; y hacemos exactamente dos cambios de color.

Si s es rojo, no hacemos rotaciones, solo cambios de color. Inicialmente, cambiamos los colores de tres nodos: g , que queda rojo, y sus hijos p y s , que quedan negros. Como g queda rojo, hay que revisar el color de su padre: si es negro, terminamos; si es rojo, repetimos estos últimos cambios de color, pero más “arriba” en el árbol.

EXAMEN 2015-2

6. Con respecto a los árboles rojo-negros:

- a) [1 pt.] Considere un árbol formado únicamente mediante la inserción de n nodos usando el método de inserción visto en clases. Justifica que si $n > 1$, el árbol tiene a lo menos un nodo rojo.
- b) [1 pt.] Muestra con un ejemplo que un árbol de más de un nodo puede tener sólo nodos negros si su construcción ha incluido eliminaciones.
- c) [2 pts.] Supón que insertamos un nodo x en un árbol rojo-negro y luego lo eliminamos inmediatamente. ¿Es el árbol resultante el mismo que el inicial? Justifica tu respuesta.
- d) [2 pts.] Un árbol con 8 nodos tiene la siguiente configuración: la raíz, con clave Q , es negra; sus hijos tienen las claves D y T , y son rojo y negro, respectivamente; D tiene dos hijos negros: A y L ; T tiene un único hijo, W ; y L tiene dos hijos: J y N . Inserta en este árbol la clave G ; en particular
 - i) [0.5 pts.] Dibuja el árbol original, antes de la inserción de G ; deduce los colores de los nodos J , N y W .
 - ii) [0.5 pts.] Dibuja el árbol justo después de la inserción de G , pero antes de cualquier operación de restauración de las propiedades del árbol. ¿Cuál propiedad **no** se cumple?
 - iii) [1 pt.] Dibuja el árbol después de cada operación de cambio de colores y de cada operación de rotación, explicando qué problema se produce en cada caso, hasta llegar al árbol final.

I1 2015-1

3. Árboles de búsqueda.

- a) La especificación de un árbol de búsqueda (no necesariamente balanceado) indica que cada nuevo elemento que se inserta debe quedar como la raíz del árbol. ¿En qué situaciones puede ser esto útil? ¿Qué tan eficientemente, en notación $O()$, puede implementarse una operación **Insertar-Nodo()** que logre este objetivo? **Justifica.**

[50%] La raíz del árbol es el elemento que se encuentra con una comparación. Por lo tanto, me conviene que un nuevo elemento que acabo de insertar quede como raíz si a continuación lo voy a buscar varias veces; y si en el mediano plazo la probabilidad de buscar ese elemento se mantiene alta, ya que cuando insertemos otro elemento, que va a quedar como nueva raíz, la raíz anterior va a quedar como hija de la nueva raíz (o sea, se va a mantener en la parte "de arriba" del árbol por un tiempo).

[50%] Para lograr esto, **Insertar-Nodo()** debe hacer lo siguiente. Primero, una inserción "normal", en que el nodo insertado queda como hoja del árbol. Luego, "hacer subir" este nodo mediante rotaciones simples hasta dejarlo en la raíz del árbol: si el nodo es un hijo izquierdo, entonces se hace una rotación a la derecha, y viceversa. Recordemos que las rotaciones preservan la propiedad de árbol de búsqueda. Con cada rotación, el nodo "sube" un nivel en el árbol; por lo tanto, se necesita un número de rotaciones igual a la altura del árbol para que el nodo finalmente quede como raíz, es decir, $O(\text{altura de árbol})$.

- b) Considera un **árbol AVL** inicialmente vacío, en el que queremos almacenar las claves 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. ¿Es posible insertar estas claves en el árbol en algún orden tal que **nunca** sea necesario ejecutar una rotación? Si tu respuesta es "sí", indica el orden de inserción y muestra al árbol resultante después de insertar cada clave. Si tu respuesta es "no", da un argumento convincente (p.ej., una demostración) de que efectivamente no es posible insertar las claves sin que haya que ejecutar al menos una rotación.

Sí es posible. La idea es hacer las inserciones de manera de mantener todo el tiempo la propiedad de balance; p.ej., procurar que el árbol se vaya llenando "por niveles". La primera clave que insertemos va a ser la raíz del árbol (ya que la idea es que no va a haber rotaciones). Por lo tanto, tiene que ser una clave k tal que el número de claves menores que k —que van a ir a parar al subárbol izquierdo— y el número de claves mayores que k —que van a ir a parar al subárbol derecho— sean muy parecidos. Si elegimos la clave 4, entonces hay 4 claves menores y 5 claves mayores (también podemos elegir la clave 5 y dejar 5 claves menores y 4 mayores). A continuación elegimos la raíz del subárbol izquierdo y la raíz del subárbol derecho (o viceversa). Para esto, aplicamos recursivamente la misma "regla", sobre las claves 0, 1, 2 y 3, para el subárbol izquierdo, y sobre las claves 5, 6, 7, 8 y 9, para el subárbol derecho; p.ej., insertamos 2 y luego 7. Repitiendo la estrategia, luego insertamos 1, 3, 6 y 8, y finalmente 0, 5 y 9. Así, un orden de inserción posible es 4, 2, 7, 1, 3, 6, 8, 0, 5, 9.

- c) Considera un **árbol rojo-negro** que corresponda a un **árbol 2-3**, tal como lo vimos en clase. Escribe el **algoritmo** que debe seguir el procedimiento de inserción en el árbol rojo-negro de manera que corresponda al procedimiento que seguiría en el árbol 2-3.

Correspondencia básica. En un árbol rojo-negro, un nodo rojo contiene la clave izquierda (menor) y los hijos izquierdo y del medio de un nodo 3 del árbol 2-3. La clave derecha y el hijo derecho del nodo 3 se convierten en la clave e hijo derecho de un nodo negro en el árbol rojo-negro; el hijo izquierdo de este nodo negro es el nodo rojo anterior. Todos los otros nodos 2 del árbol 2-3 corresponden a nodos negros en el árbol rojo-negro.

Dividimos la respuesta en tres partes.

a) Al insertar en el árbol 2-3, el caso más simple es cuando insertamos en un nodo 2 (que es una hoja), que así se convierte en un nodo 3 (y sigue siendo una hoja).

En el correspondiente árbol rojo-negro, insertamos un hijo a un nodo negro. La clave del hijo puede ser menor o mayor que la clave de su padre, es decir, puede quedar como hijo izquierdo o hijo derecho, respectivamente; además, el hijo debe ser pintado de rojo. Sin embargo, estos dos nodos deben quedar finalmente en una configuración que corresponda al nodo 3 (según la descripción de arriba). Esto será así si el hijo insertado es un hijo izquierdo. Pero si es un hijo derecho, es necesario hacer una rotación a la izquierda.

b) Un caso menos simple es cuando en el árbol 2-3 insertamos en un nodo 3, cuyo padre es un nodo 2; supongamos que el otro hijo de este nodo 2 también es un nodo 2. Es decir, en el árbol rojo-negro, insertamos en una configuración de un nodo negro con un hijo izquierdo rojo y sin hijo derecho (corresponde al nodo 3); el padre del nodo negro también es negro (corresponde al nodo 2) y tiene otro hijo negro (el otro nodo 2). Dependiendo del valor de la clave insertada, el nuevo nodo puede ir a parar como hijo izquierdo o derecho del nodo rojo, o como el hijo derecho, hasta ahora inexistente, del nodo negro: configuración inicial, justo después de la inserción.

En el árbol 2-3 el resultado es el siguiente: el nodo 2 se convierte en un nodo 3 (con dos claves y tres hijos) y el nodo 3 se convierte en tres nodos 2 (con una clave cada uno y sin hijos).

Por lo tanto, en el árbol rojo-negro la configuración final debe ser un nodo negro con dos hijos: el izquierdo rojo y el derecho negro; el hijo izquierdo rojo a su vez tiene dos hijos negros. A través de una o dos rotaciones, a uno u otro lado, y algunos cambios de colores, convertimos la configuración inicial en esta configuración final.

c) Finalmente, el caso más complicado es cuando en el árbol 2-3 insertamos en un nodo 3, cuyo padre también es un nodo 3. En el árbol rojo-negro correspondiente, la inserción es en una configuración de un nodo negro con dos hijos, izquierdo rojo y derecho negro, en que el hijo izquierdo tiene a su vez dos hijos negros, uno de los cuales, el izquierdo, tiene finalmente un hijo rojo. El nuevo nodo va a ir a parar inicialmente como hijo izquierdo o derecho de este nodo rojo, o como hijo derecho de su padre negro, y en cualquier caso pintado de rojo. En la configuración final, las hojas y sus padres son negros; pero la parte complicada se da porque alguna clave sube de nivel, potencialmente repitiendo el problema original dos niveles más arriba. Por supuesto, este problema se puede resolver aplicando recursivamente el algoritmo descrito (pero esto es más fácil decirlo que especificarlo precisamente).

EXAMEN 2015-1

1. Considera un árbol AVL en el que acabamos de insertar una nueva clave; sea Q (un puntero a) el nodo en que quedó esta nueva clave, y sea P (un puntero a) el padre de Q . Si cada nodo tiene un campo *balance* (o simplemente *bal*), además de punteros a su padre e hijos izquierdo y derecho (p, izq, der , ...)

... escribe un algoritmo —lo más parecido a código posible— eficiente que actualice el campo *balance* de cada nodo que lo necesite, y determine el nodo "pivot", o raíz del subárbol que debe ser rebalanceado mediante una rotación, si es que lo hay. ¿Qué complejidad tiene tu algoritmo? Justifica.

[2/3] Algoritmo:

```
Q = nodo recién insertado
P = Q.p
if Q es hijo izquierdo de P
    P.balance --
else
    P.balance ++
while P no es la raíz y P.balance ≠ 2 y P.balance ≠ -2
    Q = P
    P = P.p
    if Q.balance = 0
        return
    if Q es hijo izquierdo de P
        P.balance --
    else
        P.balance ++
if P.balance = 2 o P.balance = -2
    P es el nodo pivot
```

[1/3] Complejidad = $O(\log n)$, ya que hace un número constante de operaciones en el nodo que está mirando y luego "sube" al padre del nodo, en donde repite las operaciones; a lo más llega a la raíz, que está a altura $\log n$, en que n es el número de nodos en el árbol, ya que los árboles AVL son balanceados. ¡Ojo: Se asigna puntaje sólo si está bien justificado!

3. Árboles de búsqueda binarios (ABB).

- a) Supongamos que tenemos una estimación por adelantado de cuán a menudo nuestro programa va a tener acceso a las claves de los datos (es decir, conocemos la frecuencia de acceso a las claves). Si queremos emplear un ABB, inicialmente vacío, para almacenar los datos (según sus claves), ¿en qué orden deberíamos insertar las claves en el árbol? Justifica.

La idea es encontrar lo más rápidamente posible cada clave, cuando la busquemos. Como el número de comparaciones para encontrar una clave depende directamente de su profundidad en el árbol, las claves que están cerca de la raíz se encuentran haciendo pocas comparaciones; y, si el árbol está más o menos balanceado, las claves que están cerca de las hojas se encuentran haciendo varias comparaciones. Luego, nos conviene que las claves que se van a buscar más frecuentemente estén más cerca de la raíz, y que las claves que se van a buscar con menos frecuencia estén más cerca de las hojas. En un árbol que se construye sólo con inserciones (y sin rotaciones), el primer nodo que se inserta se convierte en la raíz del árbol; por lo tanto, en nuestro caso, éste debería ser el de más alta frecuencia esperada de búsqueda.

- b)** Supongamos que cada vez que nuestro programa tiene acceso a una clave, k , necesita saber cuántas claves en el árbol son menores que k ; esto se conoce como el **rango** de k . ¿Qué información adicional sería necesario almacenar en el árbol? ¿Cómo se determinaría el rango de k y cuál sería la complejidad de esta operación? ¿Cuánto costaría mantener actualizada la información adicional del árbol cuando se produce una inserción o una eliminación de una clave?

Como los ABB son ordenados, el rango de k es el número de claves almacenadas "a la izquierda" de la clave k . Así, si k está justamente en la raíz del árbol, entonces su rango es igual al número de nodos en el subárbol izquierdo. En cambio, si k es menor que la clave de la raíz (y, por lo tanto, está en el subárbol izquierdo), entonces su rango (en todo el árbol) es igual a su rango dentro del subárbol izquierdo, ya que las claves en el subárbol derecho son todas mayores que k . Finalmente, si k es mayor que la clave de la raíz (y, por lo tanto, está en el subárbol derecho), entonces su rango es igual al número de claves en el subárbol izquierdo (ya que todas ellas son menores que k) más uno (la clave de la raíz también es menor que k) y más el rango de k dentro del subárbol derecho (el número de claves del subárbol derecho que son menores que k). Por supuesto, el rango de k en el subárbol izquierdo y el rango de k en el subárbol derecho se determina recursivamente de la misma forma.

Por lo tanto, la información adicional que conviene almacenar en el árbol para poder aplicar el algoritmo anterior es, para cada nodo t , almacenar en un campo *size* el número de nodos que hay en el subárbol cuya raíz es t . Esta información se mantiene actualizada fácilmente. Durante una inserción, sumamos uno al campo *size* de cada nodo en la ruta de inserción, y para el nodo insertado hacemos *size* = 1. Luego de hacer una eliminación, restamos uno al campo *size* de cada nodo en la ruta desde el padre del nodo eliminado hasta la raíz. En ambos casos, esto cuesta $O(\text{altura del árbol})$.

4. Árboles de búsqueda balanceados.

- a)** Describe un algoritmo de certificación para árboles AVL, suponiendo que sabemos que el árbol es un ABB. Es decir, tu algoritmo debe verificar que la propiedad de balance del árbol se cumple y que la información de balance mantenida en cada nodo está correcta. Tu algoritmo debe correr en tiempo $O(\text{número de nodos en el árbol})$.

El algoritmo asigna una "altura" a cada nodo: una hoja tiene altura 0; la altura de un nodo que no es una hoja es uno más que la altura de su subárbol más alto. El algoritmo comienza en la raíz del árbol.

Si el nodo es una hoja, entonces su balance debe ser 0. Si el balance no es 0, entonces el algoritmo responde "falso" y termina; de lo contrario, el algoritmo devuelve la altura del nodo: 0.

Si el nodo no es una hoja, entonces el algoritmo se ejecuta recursivamente primero en el subárbol izquierdo y luego el subárbol derecho del nodo. Si estas ejecuciones recursivas vuelven con los valores h_{izq} y h_{der} , respectivamente, entonces el algoritmo verifica la propiedad de árbol AVL (h_{izq} y h_{der} no pueden diferir en más de uno) y el balance del nodo (debe ser 0, -1 o +1, dependiendo de la relación entre h_{izq} y h_{der}). Si cualquiera de estas verificaciones no se cumple, entonces el algoritmo responde "falso" y termina; de lo contrario, el algoritmo devuelve la altura del nodo: uno más que el mayor entre h_{izq} y h_{der} y, si el nodo es la raíz del árbol, el algoritmo responde "verdadero".

EXAMEN 2014-1

2) Un árbol rojo-negro con 8 nodos tiene la siguiente configuración: la raíz, con clave Q , es negra; sus hijos tienen las claves D y T , y son rojo y negro, respectivamente; D tiene dos hijos negros: A y L ; T tiene un único hijo, W ; y L tiene dos hijos: J y N . Inserta en este árbol la clave G ; en particular,

a) [1 pt.] Dibuja el árbol original, antes de la inserción de G ; deduce los colores de los nodos J , N y W .

J , N y W son rojos. Si alguno fuera negro, entonces se violaría la propiedad de la altura negra del árbol: la ruta $Q-D-A$, en que A es una hoja, tiene sólo dos nodos negros, y cualquiera de las rutas desde la raíz a J , N o W ya tiene dos nodos negros.

b) [2 pts.] Dibuja el árbol justo después de la inserción de G , pero antes de cualquier operación de restauración de las propiedades del árbol. ¿Cuál propiedad no se cumple?

G se inserta en la forma habitual en un árbol de búsqueda, queda como hijo izquierdo de J , y se pinta rojo; pero J es rojo, por lo que ahora hay un nodo y su hijo ambos rojos.

c) [3 pts.] Dibuja el árbol después de cada operación de cambio de colores y de cada operación de rotación, explicando qué problema se produce en cada caso, hasta llegar al árbol final.

Primero, como el hermano N de J también es rojo, cambiamos colores: J y N quedan negros, y su padre, L , rojo. Pero el padre, D , de L también es rojo; es decir, "subimos" el problema de dos nodos rojos consecutivos.

Pero ahora el hermano, T , de D es negro, por lo que no podemos aplicar el cambio de colores anterior. Como L es hijo derecho de D , rotamos $D-L$ a la izquierda, sin cambiar colores, dejando L como hijo izquierdo de la raíz, y D como hijo izquierdo de L .

De nuevo hay un nodo y su hijo rojos, L y D , y el hermano, T , de L es negro. Pero ahora D es hijo izquierdo de L ; por lo que rotamos $Q-L$ a la derecha y pintamos Q de rojo.

I2 2013-1

4) Con respecto a los árboles AVL:

a) Muestra la secuencia de árboles AVL's que se forman al insertar las claves 3, 2, 1, 4, 5, 6, 7, 16, 15 y 14, en este orden, en un árbol AVL inicialmente vacío.

Las inserciones de 3 y 2 son triviales (no tienen puntaje). La inserción de 1 exige una rotación simple a la derecha. La inserción de 4 nuevamente es trivial. Las inserciones de 5, 6 y 7 exigen rotaciones simples a la izquierda. La inserción de 16 es trivial. Finalmente, las inserciones de 15 y 14 exigen rotaciones dobles: en ambos casos, una rotación simple a la derecha seguida de una rotación simple a la izquierda. El árbol AVL resultante tiene como raíz a 4, cuyos hijos son 2 y 7. Los hijos de 2 son 1 y 3; los de 7 son 6 y 15. El único hijo de 6 es 5; los hijos de 15 son 14 y 16.

EXAMEN 2013-1

4) ¿Cuál es la propiedad de árbol de búsqueda binario?

Cualquier clave en el subárbol izquierdo es menor que la clave en la raíz, que a su vez es menor que cualquier clave en el subárbol derecho

5) Escribe un método *esABB()*, que reciba un nodo como parámetro y devuelva *true* si el nodo es la raíz de un árbol de búsqueda binario; *false*, en otro caso.

Lo más simple puede ser escribir un método recursivo:

```
esABB(x):
    if ( null(x) ) return true
    if ( hoja(x) ) return true
    if ( esABB(x.derecho) && esABB(x.izquierdo) )
        if ( null(x.derecho) ) return x.key < x.izquierdo.key
        if ( null(x.izquierdo) ) return x.derecho.key < x.key
        return x.derecho.key < x.key && x.key < x.izquierdo.key
    else return false
```

6) ¿Qué propiedades adicionales debe cumplir un árbol rojo-negro?

- i) Todo nodo es ya sea rojo o negro.
- ii) Si un nodo es rojo, entonces sus hijos son negros.
- iii) Para cada nodo, todas las rutas desde el nodo a las hojas descendientes contienen el mismo número de nodos negros.

7) Escribe un método *esRojo-Negro()*, que reciba un nodo como parámetro y devuelva *true* si el nodo es la raíz de un árbol de búsqueda binario rojo-negro; *false*, en otro caso.

Tal vez, lo más simple es escribir dos métodos recursivos, uno para verificar ii) y el otro para verificar iii); cualquiera de ellos puede, además, verificar i):

```
esRojo-Negro(x):
    if ( esABB(x) ) return cumple-ii(x) && cumple-iii(x)!=-1
    else return false

cumple-ii(x): —también verifica i)
    if ( null(x) ) return true
    if ( hoja(x) ) return es rojo o negro
    if ( es negro ) return cumple-ii(x.derecho) && cumple-ii(x.izquierdo)
    if ( es rojo )
        if ( null(x.izquierdo) ) return cumple-ii(x.derecho) && x.derecho es negro
        if ( null(x.derecho) ) return cumple-ii(x.izquierdo) && x.izquierdo es negro
        return cumple-ii(x.derecho) && x.derecho es negro && cumple-ii(x.izquierdo) && x.izquierdo es negro
    else return false

cumple-iii(x)
    if( null(x) ) return 0
    if( es rojo && hoja(x) ) return 0
    if( es negro && hoja(x) ) return 1
    p = cumple-iii(x.izquierdo)
    q = cumple-iii(x.derecho)
    if (p!=q) return -1
    if(p== -1 || q== -1) return -1
    if( es rojo ) return p
    if( es negro) return p+1
```

I1 2011-1

3. Un árbol binario de búsqueda (ABB) se dice **balanceado** si la diferencia en altura de ambos subárboles de cualquier nodo es cero o uno (p.ej., un árbol AVL). Un ABB se dice **perfectamente balanceado** si es balanceado y todas sus hojas están en un mismo nivel o, a lo más, en dos niveles (p.ej., un *heap* binario). Da un algoritmo para convertir cualquier ABB en un ABB perfectamente balanceado.

Para esto, considera un ABB totalmente desbalanceado, convertido en una lista ligada hacia la derecha, con $2^k - 1$ nodos, para algún entero k . Partiendo desde la raíz, baja por la lista y haz una rotación simple a la izquierda cada dos nodos (la primera rotación es del hijo derecho de la raíz con respecto a la raíz; la segunda rotación involucra a los dos nodos siguientes en la lista; y así sucesivamente).

¿Qué ha pasado con los desbalances cuando llegas abajo? ¿Qué tendrías que hacer a continuación para convertir finalmente el árbol en uno perfectamente balanceado y completo? ¿Cómo manejarías el caso de un árbol que no puede ser completo, es decir, que su número de nodos no puede escribirse como $2^k - 1$?

La sugerencia que parte en el segundo párrafo es como la ‘segunda’ parte del algoritmo pedido. La primera parte consiste en convertir el árbol original en una lista ligada hacia la derecha; esto lo vimos en la ayudantía de 25/8:

```
while ( hay un nodo x con un hijo izquierdo y )
    hacer una rotación simple a la derecha, de y con respecto a x
```

Como cada rotación simple a la derecha reduce en uno el número de hijos izquierdos en el árbol, a lo más $n-1$ rotaciones son necesarias para convertir todo el árbol en una lista ligada hacia la derecha.

Una vez hecho esto, se aplica el algoritmo sugerido. Las rotaciones sugeridas, a lo largo de la lista y ‘nodo por medio’, van reduciendo paulatinamente el desbalance: las primera rotación reduce el desbalance de la raíz en dos; las siguientes, en uno cada una. Así, al llegar abajo, el desbalance de la raíz se ha reducido de $n-1$ a $n-3-\lfloor n/2 - 1 \rfloor$. También se han reducido los desbalances de los subárboles. Lo que hay que hacer a continuación es repetir el proceso, desde la raíz, por la rama más a la derecha del árbol; el número rotaciones a realizar en esta segunda pasada va a ser la mitad de las de la primera pasada, y el árbol va a quedar aún un poco mejor balanceado. Y así sucesivamente, hasta formar un árbol completo (ya que n es de la forma $2^k - 1$) perfectamente balanceado.

Si el árbol original no es completo, hay que tratar el número de nodos ‘sobrantes’ como casos especiales, p.ej., haciendo ese número de rotaciones a la izquierda justo antes de aplicar el algoritmo sugerido.

4. Uno de los ejercicios propuestos en el texto de Cormen et al. afirma que si un árbol rojo-negro de más de un nodo ha sido construido sólo a través de operaciones de inserción (sin eliminaciones), entonces el árbol tendrá al menos un nodo rojo. Muestra con un ejemplo que un árbol rojo-negro de más de un nodo puede tener sólo nodos negros si su construcción ha incluido eliminaciones.

Supongamos un árbol inicialmente vacío, al que insertamos las claves 3, 2 y 4, en este orden: 3 queda en la raíz, y es negro, y 2 y 4 son sus hijos izquierdo y derecho, respectivamente, ambos rojos. Si ahora insertamos la clave 1, esta va a parar como hijo izquierdo de 2, también rojo. Al ser rojos 2 y su hijo 1, hay que hacer algo; en este caso, como el hermano de 2, 4, también es rojo, intercambiamos colores por niveles: pintamos negros 2 y 4 y pintamos rojo 3. Pero como 3 es la raíz, lo volvemos a pintar negro. En este momento, sólo 1 es rojo.

Si ahora eliminamos 1, simplemente eliminamos una hoja roja, por lo que no es necesario hacer ninguna otra operación en el árbol, que así queda sólo con tres nodos negros.

I1 2010-2

2. Demuestra que cualquier ABB arbitrario de n nodos puede ser transformado en cualquier otro ABB arbitrario de n nodos usando $O(n)$ rotaciones.

a) Primero, prueba que con a lo más $n-1$ rotaciones a la derecha puedes convertir cualquier ABB en uno que es simplemente una lista de n nodos ligada por los punteros a los hijos derechos.

Llamemos **lista derechista** a la lista de nodos ligada por los punteros a los hijos derechos; queremos convertir el árbol en una lista derechista de n nodos. Inicialmente, la lista derechista tiene al menos un nodo: la raíz del árbol.

Cada rotación a la derecha respecto de un nodo que está en la lista derechista, y tiene un hijo izquierdo, agrega ese hijo izquierdo a la lista derechista. La lista comienza con al menos un nodo, que es la raíz. Por lo tanto, con a lo más $n-1$ rotaciones a la derecha, todos los nodos han pasado a la lista.

Ahora observa que si conocieras la secuencia de rotaciones a la derecha que transforman un ABB arbitrario T en una lista ligada T' , entonces podrías ejecutar esta secuencia en orden inverso, cambiando cada rotación a la derecha por su rotación inversa a la izquierda, para transformar T' de vuelta en T .

b) Explica cómo transformar un ABB T_1 en otro T_2 , pasando por la lista ligada única T' de nodos de T_1 (que es la misma que la de los nodos de T_2). ¿Cuántas rotaciones a lo más es necesario hacer para esto?

T' es la lista derechista. Como vimos en a), podemos convertir T_1 en T' con a lo más $n-1$ rotaciones a la derecha. Como los nodos de T_2 son los mismos que los de T_1 , entonces también podríamos convertir T_2 en T' con a los más $n-1$ rotaciones a la derecha. Si a partir de T' aplicamos estas últimas $n-1$ rotaciones en orden inverso, y cada vez rotamos a la izquierda en vez de la derecha, obtenemos T_2 .

3. ¿Cuántos cambios de color y cuántas rotaciones pueden ocurrir a lo más en una inserción en un árbol rojo-negro? Justifica tus respuestas.

Recuerda que al insertar un nodo, x , lo insertamos como una hoja y lo pintamos de rojo. Si el padre, p , de x es negro, terminamos. Si p es rojo, tenemos dos casos: el hermano, s , de p es negro; s es rojo. Si s es negro, realizamos algunas rotaciones y algunos cambios de color. Si s es rojo, sabemos que el padre, g , de p y s es negro; entonces, cambiamos los colores de g , p y s , y revisamos el color del padre de g .

Hacemos $O(\log n)$ cambios de color y $O(1)$ rotaciones. Como dice arriba, esto ocurre solo cuando p es rojo.

Si s es negro, hacemos exactamente una o dos rotaciones, dependiendo de si x es el hijo izquierdo o el hijo derecho de su padre; y hacemos exactamente dos cambios de color.

Si s es rojo, no hacemos rotaciones, solo cambios de color. Inicialmente, cambiamos los colores de tres nodos: g , que queda rojo, y sus hijos p y s , que quedan negros. Como g queda rojo, hay que revisar el color de su padre: si es negro, terminamos; si es rojo, repetimos estos últimos cambios de color, pero más "arriba" en el árbol.