Universidad Nacional Autónoma de México

Facultad de Ciencias

Implementaciones funcionales de árboles roji-negros

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADA EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
GRACIELA LÓPEZ CAMPOS

DIRECTOR DE TESIS:
DR. FAVIO EZEQUIEL MIRANDA PEREA

2015

Hoja de Datos del Jurado

1. Datos del alumno
López
Campos
Graciela
56547026
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
306307966

2. Datos del tutor
Dr.
Favio Ezequiel
Miranda
Perea

3. Datos del sinodal 1
Dr.
Carlos Bruno
Velarde
Velázquez

4. Datos del sinodal 2
Dra.
Verónica Esther
Arriola
Ríos

5. Datos del sinodal 3
M. en C.
Noé Salomón
Hernández
Sánchez

6. Datos del sinodal 4
L. en C.C.
Pilar Selene
Linares
Arévalo

7. Datos del trabajo escrito
Implementaciones funcionales de árboles roji-negros
100p
2015

Agradecimiento

El desarrollo del presente trabajo de tesis tuvo lugar en el marco del proyecto

"Formalismos lógico - categóricos para la programación funcional"
(PAPIIT, IN117711)

otorgado por la Dirección General de Asuntos del Personal Académico
de la Universidad Nacional Autónoma de México.

Agradecimientos

Quiero decir que no fue fácil llegar hasta aquí, hubo muchos obstáculos, caídas, lágrimas y sacrificios, pero al final estoy aquí y ahora presentando este trabajo y puedo expresar que me siento orgullosa de haberlo logrado. Quiero decir además que no hubiera podido llegar hasta este punto de no ser por:

Mis padres Graciela y Agustín quienes me han apoyado todo lo que va de mi vida y quienes estoy segura me seguirán apoyando el resto de mi días. Papás, quiero agradecerles que nunca me falto amor y economicamente siempre me dieron lo que necesité y hasta más, quiero agradecerles los valores inculcados desde mi niñez mismos que me convirtieron en la mujer que soy el día de hoy, también quiero agradecerles que compartieran conmigo sus experiencias y conocimientos, esperando nunca dejen de enseñarme por que aún puedo aprender mucho de ustedes. Quiero darles las gracias por la paciencia brindada y la espera ya que estoy conciente de que tarde un poco en darles este obsequio pero quiero que sepan que no lo hubiera conseguido sin ustedes. Los amo, mil gracias por todo.

Mi hermano Cristian, que siempre me animaba diciendo que ya era el último esfuerzo para terminar la licenciatura que no lo dejara y que no me rindiera.

Mi tutor el Dr. Favio, por todo su apoyo brindado no solo en la elaboración de este trabajo, sino desde que comencé a tomar clases con él como mi profesor. Favio, quiero decirte que siempre me gustaron tus clases, los cursos de análisis lógico y programación funcional y lógica fueron mis cursos preferidos aunque teoría de las categorías, verificación en Coq y lenguajes de programación no se quedaron atrás, gracias por todas tus enseñanzas y quiero agradecerte toda la paciencia tenida. Mil gracias por todas las oportunidades que me proporcionaste no solo para aprender, sino también para crecer como persona y profesionalmente.

Mis sinodales, gracias por el tiempo brindado a la revisión de este trabajo, por sus observaciones y por su apoyo para que la publicación fuera lo mejor posible.

Por último me gustaría agradecer a mi Universidad en particular a la Facultad de Ciencias por haberme formado académicamente y enseñarme lo necesario para desarrollarme profesionalmente.

Índice general

Introducción

De acuerdo con [5], una *estructura de datos* es una manera de almacenar y analizar datos de forma que se simplifique el acceso y modificación a los mismos. Por otra parte [4], nos dice que una estructura de datos es un método para realizar un conjunto de operaciones sobre algunos datos. En cualquier caso, una estructura de datos es una manera particular de implementar una definición matemática, llamada *tipo de datos abstracto*, la cual consiste de una clase de individuos cuyo comportamiento se define mediante una serie de operaciones que deben tener ciertas propiedades. Ejemplos de estructuras de datos que implementan tipos de datos abstractos son: las pilas, las colas o los conjuntos. Obsérvese que estos tipos de datos abstractos pueden implementarse mediante ciertas estructuras de datos como listas o arreglos que también se pueden definir como tipos de datos abstractos.

Ninguna estructura de datos es útil para todos los propósitos, por lo que es importante conocer las ventajas y desventajas de cada una en particular. Por ejemplo, si deseamos almacenar información jerárquica que depende de una relación de orden absoluto entre los elementos para poder ubicarlos más facilmente, como la de un directorio en un sistema de archivos de computadora, no es buena idea hacerlo utilizando una lista. Una familia importante de estructuras de datos que sirven de manera natural para guardar información de esta clase son los árboles.

Un tipo particular de árboles son aquellos llamados *árboles binarios de búsqueda*, los cuales son árboles binarios que guardan la información de manera ordenada de tal forma que, dado cualquier nodo del árbol, su subárbol izquierdo es un árbol binario de búsqueda, que contiene únicamente elementos menores o iguales que él, mientras que su subárbol derecho es un árbol binario de búsqueda, que aloja solamente elementos mayores o iguales.

En este trabajo nos enfocaremos al estudio e implementación de una clase especial de árbol binario de búsqueda, denominado *árbol roji-negro* (red-black tree, en inglés) conocido también como B-árbol binario simétrico, nombre impuesto por su inventor Rudolf Bayer [3, 7]. En un árbol roji-negro cada nodo es de color rojo o negro, el uso de estos colores permite definir una restricción que garantiza

un balanceo aproximado. Esta clase de árboles ofrece operaciones eficientes (en tiempo logarítmico) de inserción, borrado y búsqueda, por lo que son de gran utilidad en ciencias de la computación. Por ejemplo, el calendarizador CFS del kernel del sistema operativo Linux utiliza árboles roji-negros.

En programación funcional, paradigma utilizado en este trabajo, los árboles roji-negros son de importancia al ser una de las estructuras de datos persistentes más comunes utilizada para construir arreglos asociativos (diccionarios) y conjuntos. Nuestro objetivo es presentar algunas implementaciones funcionales de árboles roji-negros en el lenguaje de programación Haskell, haciendo enfásis en la operación de borrado, dado que ésta se omite en el libro más importante acerca de estructuras de datos funcionales [13]. Veremos aquí a detalle diversas soluciones de esta omisión.

El contenido del trabajo es como sigue:

- En el primer capítulo damos un breve panorama de la programación funcional y del lenguaje de programación Haskell. Mostramos también una implementación simple de árboles binarios y árboles binarios de búsqueda.
- En el capítulo dos comenzamos nuestro estudio sobre árboles roji-negros revisando la implementación dada por Okasaki [13] para las operaciones de balanceo e inserción.
- En el tercer capítulo nos dedicamos a desarrollar cuatro implementaciones de la operación de borrado. Se utilizan, desde constructores inteligentes y aritmética de colores, hasta el uso de conceptos avanzados de la programación funcional como son los tipos anidados, los cuales garantizan de manera estática la construcción de un árbol roji-negro válido, es decir, en tiempo de compilación.
- Finalmente, en el capítulo cuatro se muestra una aplicación sencilla de árboles roji-negros para implementar conjuntos utilizando el mecanismo de clases de Haskell.

Capítulo 1

Preliminares

1.1. ¿Por qué programación funcional?

Se decidió usar programación funcional porque es elegante y concisa ya que utiliza conceptos de alto nivel, además nos brinda varias ventajas con respecto a la programación imperativa; por ejemplo, la definición de nuestros programas se hace mediante funciones (en contraste con los lenguajes imperativos que utilizan la asignación de variables), esto en principio parecería ser una limitante pero no es así, escribir los programas mediante funciones nos permite razonar matemáticamente acerca del comportamiento de un programa y deducir con cierta facilidad cuándo un programa es correcto.

Dentro de los lenguajes funcionales tenemos a Haskell, el cual es un lenguaje de programación de amplio espectro inspirado fuertemente en el *cálculo lambda* y será el lenguaje a utilizar en nuestra implementación. En Haskell las funciones son consideradas entidades de primera clase, por lo que podemos tratarlas como cualquier valor. Así es posible construir funciones más complejas anidando funciones simples y devolviendo funciones como resultados. Un ejemplo de esto son las *funciones de orden superior*, las cuales reciben funciones como argumentos, pudiendo devolver funciones como resultado.

Los programas en Haskell, a diferencia de los programas imperativos, en general son más cortos, esto es porque en un lenguaje imperativo, para dar solución a algún problema se transforma un algoritmo en una secuencia de instrucciones del lenguaje, mientras que en un lenguaje funcional se describen las propiedades que debe cumplir la solución al problema, sin importar el algoritmo necesario para conseguir dicha solución. Debido a esto podemos decir que las implementaciones funcionales como en el caso de Haskell son más fáciles de comprender y por tanto más fáciles de mantener.

HASKELL es un lenguaje puro, perezoso y fuertemente tipado. Con el término *puro* nos referimos a que en HASKELL las expresiones pueden ser reemplazadas libremente por sus valores y viceversa sin alterar el resultado del programa, lo que se conoce como *transparencia referencial*, haciendo matemáticamente más tratables los programas. El término *perezoso* quiere decir que HASKELL no ejecutará funciones ni calculará resultados hasta que se vea realmente forzado a hacerlo, esto ayuda a que la evaluación de un programa sea más rápida y eficiente. El término *fuertemente tipado* se refiere a que el compilador, al analizar un programa sabe qué trozos del código de tipo son entero, flotante, cadena, etc., es decir, cuenta con un sistema de tipado estático, asegurando que los argumentos que recibe una función corresponden al tipo esperado. Por ejemplo, si definimos una función que recibe dos valores numéricos y posteriormente pretendemos emplearla con valores no numéricos, el compilador nos devolverá un mensaje de error, informando que dicha operación no se puede realizar por conflicto de tipos. Además del sistema de tipado estático, HASKELL cuenta con un mecanismo de inferencia de tipos, el cual nos permite omitir la declaración explícita de tipos en la definición de funciones.

1.2. Estructuras de datos funcionales

Una estructura de datos funcional es una estructura de datos adecuada para implementarse en un lenguaje de programación funcional, o bien, para codificarse en un lenguaje no funcional como C o JAVA utilizando un estilo funcional. Las estructuras de datos funcionales están relacionadas a las estructuras de datos persistentes o inmutables. Aunque estos tres términos se usan frecuentemente de manera intercambiable, hay diferencias sutiles que puntualizamos a continuación:

- El término *persistente* se refiere a la clase de estructuras de datos en donde una actualización no destruye versiones previas de la estructura, sino que crea una nueva versión que coexiste con la versión previa.

- El término *inmutable* se refiere a una técnica particular de implementación, la cual garantiza que la memoria ocupada para una versión particular de la estructura, una vez inicializada, nunca se altera. Observése que la inmutabilidad de los datos produce una gran cantidad de datos temporales, pero ésta también ayuda a la rápida recolección de basura, debido a que conforme se van evaluando las expresiones del programa se va reservando la memoria, en caso de que la memoria se agote, el recoletor determina qué partes de la memoria ya no son útiles y por tanto se pueden reutilizar.

- El término *funcional* hace énfasis en el lenguaje o estilo de codificación en el que se implementan las estructuras persistentes.

Las estructuras funcionales son siempre inmutables. Más aún, debido a que HASKELL es un lenguaje puro, sus estructuras funcionales son siempre persistentes.

Para obtener un panorama general de las estructuras de datos funcionales recomendamos el libro [13].

Presentamos aquí una implementación de árboles binarios de búsqueda para recordar la sintaxis de HASKELL. Todas las instrucciones utilizadas pueden consultarse en cualquier libro del lenguaje, por ejemplo [10]. Cabe mencionar que todas las implementaciones presentes en este trabajo fueron ejecutadas en el intérprete *Hugs 98 versión de Septiembre 2006*, también pueden ser ejecutadas en *GHCi versión 7.4.1*, cabe mencionar que si se usa *GHCi* para ejecutar la implementación con tipos anidados, hay que escribir la siguiente línea *ghci -W ARNta.hs -XFlexibleInstances -XFlexibleContexts -XUndecidableInstances.*

1.3. Árboles binarios

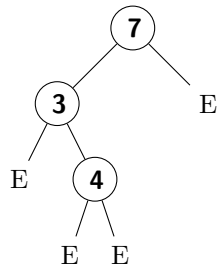
En computación un árbol es una estructura que consta de nodos organizados de manera jerárquica, en los cuales podemos guardar información. Esta estructura corresponde a la definición matemática, donde un árbol es una gráfica minimalmente conexa y maximalmente acíclica. Existen distintos tipos de árboles pero los que nos interesan por ahora son los árboles binarios y dos de sus derivados, árboles binarios de búsqueda y árboles roji-negros, de estos últimos hablaremos en las siguientes secciones.

Definición 1.1 *Un árbol binario es vacío o consiste de un nodo raíz y de dos subárboles ajenos que son también árboles binarios.*

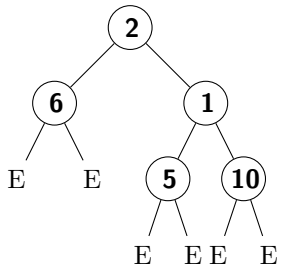
En HASKELL definimos un árbol binario (`AB`) con nodos etiquetados, cuyos elementos son de tipo `a` como:

```
data AB a = E | T (AB a) a (AB a) deriving Show
```

donde `E` representa al árbol vacío y `T (AB a) a (AB a)` representa un árbol no vacío. Por ejemplo el árbol



tiene la representación `T (T E 3 (T E 4 E)) 7 E`
Otro ejemplo es el árbol



que tiene la representación `T (T E 6 E) 2 (T (T E 5 E) 1 (T E 10 E))`
La operación más sencilla que podemos hacer en este tipo de árboles es la de saber si un elemento pertenece al árbol o no, dicha operación es la función *member* que damos a continuación.

```
member :: Eq a => a -> AB a -> Bool

member x E = False
member x (T l y r) | x == y = True
                   | otherwise = member x l || member x r
```

Observemos que la pertenencia debemos verificarla en ambos subárboles pues en este tipo de árboles no contamos con restricciones de orden para insertar los elementos, por lo que no es posible saber mediante comparaciones (`<` o `>`) si el elemento buscado se encuentra en el subárbol izquierdo o derecho, como es el caso en los árboles de la siguiente sección.

La implementación de la función de inserción para árboles binarios debe tomar en cuenta si deseamos tener elementos repetidos o no. En este trabajo nos interesan árboles sin elementos repetidos. A continuación daremos las implementaciones para las operaciones de inserción y borrado.

La inserción se ve como sigue:

```
insert :: Eq a => a -> AB a -> AB a

insert x E = T E x E
```

```
insert x t@(T l y r) | member x t =  t
                     | otherwise = (T l y (insert x r))
```

Observemos que como queremos evitar la inserción de elementos repetidos debemos verificar si el elemento que se desea insertar ya pertenecía al árbol, en cuyo caso la inserción no se realiza. En otro caso el elemento se inserta en el subárbol derecho. Esta elección es arbitraria y por supuesto genera árboles degenerados, a no ser que se tenga otra forma de hacer la inserción.

La operación de borrado la definimos como:

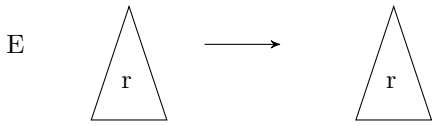
```
remove :: Eq a => a -> AB a -> AB a

remove x E = E
remove x (T l y E) | x==y = l
remove x (T E y r) | x==y = r
remove x (T l y r) | x==y = join l r
                   | member x l = T (remove x l) y r
                   | member x r = T l y (remove x r)
                   | otherwise = (T l y r)
```

La operación *join* usada en esta implementación es la que se muestra a continuación:

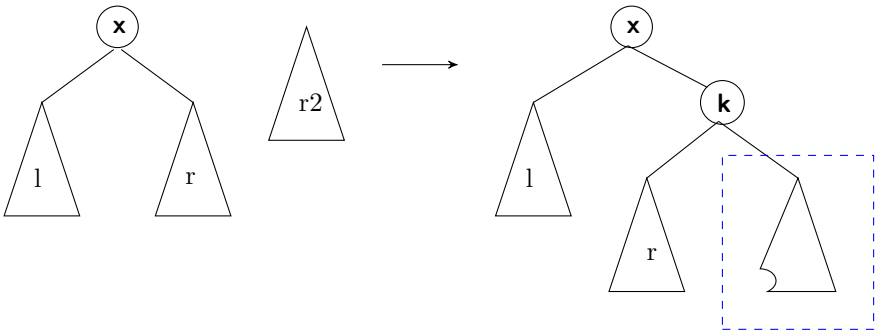
```
join :: Eq a => AB a -> AB a -> AB a

join E r = r
join (T l x r) r2 = let k = mLeft r2
                        in T l x (T r k (remove k r2))

mLeft :: AB a -> a

mLeft (T E x r) = x
mLeft (T l x r) = mLeft l
```

donde la función $mLeft$ devuelve el elemento mínimo del subárbol izquierdo. La función *join* es la encargada de pegar dos árboles, las siguientes imagenes muestran gráficamente la función *join*

El primer caso es



Y el segundo es



remove k r2

Como se ve esta implementación resulta ineficiente debido a que no hay restricción alguna en la manera de guardar los elementos. Por lo que esta estructura resulta inadecuada para aplicaciones. Situación que resolvemos a continuación.

1.4. Árboles binarios de búsqueda

Definición 1.2 *Un árbol binario de búsqueda es un árbol binario vacío o cumple con lo siguiente: Cada elemento en su subárbol izquierdo es menor que el elemento raíz, cada elemento en su subárbol derecho es mayor que el elemento raíz y ambos subárboles son también árboles binarios de búsqueda.*

Para implementar este tipo de árboles en HASKELL, utilizamos la misma definición de árbol binario, aunque por motivos de claridad cambiamos el nombre a `ABB`. Por lo que la definición es:

```
data ABB a = E | T (ABB a) a (ABB a) deriving Show
```

Obsérvese que la definición del tipo no está garantizando la invariante de orden, de manera que necesitamos definir una función que verifique si un árbol `t` de tipo `ABB a` es en realidad un árbol binario de búsqueda de acuerdo con la definición anterior. Dicha función es:

```
checkABB :: Ord a => ABB a -> Bool

checkABB E = True
checkABB (T l x r) = minT x r && maxT x l

minT :: Ord a => a -> ABB a -> Bool

minT x E = True
```

```
minT x (T l y r) = minT x l && minT x r && x < y

maxT :: Ord a => a -> ABB a -> Bool

maxT x E = True
maxT x (T l y r) = maxT x l && maxT x r && x > y
```

Las operaciones de pertenencia, inserción y borrado sobre un árbol binario de búsqueda se definen como sigue, considerando que los árboles binarios de búsqueda dados como argumentos han sido verificados por la función *checkABB*.

```
member :: Ord a => a -> ABB a -> Bool

member x E = False
member x (T tl y tr)
                  | x<y = member x tl
                  | x>y = member x tr
                  | otherwise = True

insert :: Ord a => a -> ABB a -> ABB a

insert x E = T E x E
insert x (T tl y tr)
                  | x < y = T (insert x tl) y tr
                  | x > y = T tr y (insert x tr)
                  | otherwise = T tl y tr

remove :: Ord a => a -> ABB a -> ABB a

remove x E = E
remove x (T l y E) | x==y = l
remove x (T E y r) | x==y = r
remove x (T l y r)
                  | x<y  = T (remove x l) y r
                  | x>y  = T l y (remove x r)
                  | x==y = let k = minTree r in T l k (remove k r)

minTree :: Ord a => ABB a -> a

minTree (T E x r) = x
minTree (T l x r) = minTree l
```

Esta clase de árboles ordenados es la más simple pues la única invariante a respetar, de acuerdo a la definición, depende exclusivamente de la relación de orden. Sin embargo, existen otras estructuras de árbol basadas en árboles binarios de búsqueda que requieren de invariantes más fuertes. Tal es el caso de los

árboles roji-negros, estructura protagonista de este trabajo que discutimos en los siguientes capítulos.

Capítulo 2

Árboles roji-negros

Considerese el problema de representar un conjunto finito de elementos que satisfacen una relación de orden, utilizando un árbol binario de búsqueda, el cual se resuelve insertando uno a uno los elementos de dicho conjunto en un árbol que inicialmente es vacío. Si los elementos del conjunto están desordenados esta solución es satisfactoria pues produce un árbol aproximadamente balanceado. Sin embargo, en el caso en que los elementos del conjunto estén ordenados, el resultado será un árbol completamente desbalanceado hacia la izquierda o hacia la derecha, lo cual causa ineficiencia en las funciones de búsqueda y borrado. Para solucionar este problema necesitamos que, independientemente de si los elementos del conjunto están desordenados o no, el árbol mantenga cierto balance. Los árboles roji-negros garantizan este escenario.

Un árbol roji-negro es un árbol binario de búsqueda en donde cada nodo recibe un color rojo o negro. Estos colores nos permitirán definir dos invariantes que garantizaran que la longitud del camino más largo en un árbol no sea mayor que el doble de la longitud del camino más corto. Por lo que estos árboles no presentan nunca un desbalance extremo, como puede suceder con los árboles binarios de búsqueda sin balancear.

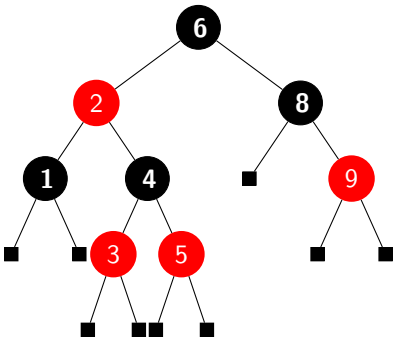
En este capítulo desarrollamos la implementación funcional básica para árboles roji-negros, que incluye las funciones de pertenencia, balanceo e inserción. Empezamos por dar la definición formal de árboles roji-negros.

Definición 2.1 *Un árbol binario de búsqueda es roji-negro si satisface que:*

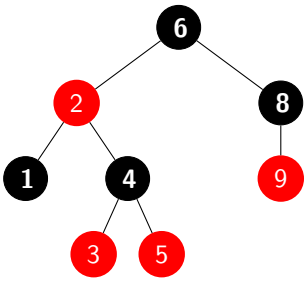
1. *Cada nodo es rojo o negro.*
2. *El árbol vacío es negro.*
3. *La raíz es negra (condición que se impone para simplificar algunas operaciones).*

4. *Se deben cumplir las siguientes invariantes:*
   - *Un nodo rojo debe tener hijos negros.*
   - *Cada camino desde la raíz a cualquier hoja, pasa por el mismo número de nodos negros.*

Los siguientes dibujos muestran un árbol roji-negro de enteros. Se trata del mismo árbol, pero en el caso de la izquierda dibujamos de manera explícita a los árboles vacíos denotados con un cuadrado negro.



Árbol roji-negro con vacíos.



Árbol roji-negro.

Una implementación de árboles roji-negros es la siguiente:

- Se definen los colores de los nodos (rojo, negro) como sigue:

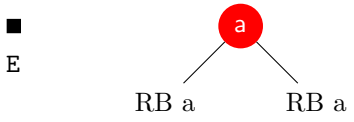
  ```
  data Color = R | B deriving (Show, Eq)
  ```

- El tipo de árboles roji-negros es

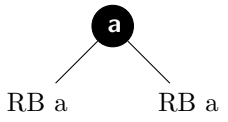
  ```
  data RB a = E | T Color (RB a) a (RB a) deriving Show
  ```

  donde el constructor `E` representa al árbol vacío de color negro, y el operador `T` construye un árbol a partir de dos árboles, una etiqueta para el nodo y un color dados.

La representación gráfica de lo anterior se muestra a continuación.



`T R (RB a) a (RB a)`



`T B (RB a) a (RB a)`

Por ejemplo, la representación formal del árbol en el ejemplo de la primer imagen (página 10) es:

```
T B
  (T R (T B E 1 E) 2 (T B (T R E 3 E) 4 (T R E 5 E)))
  6
  (T B E 8 (T R E 9 E))
```

Una vez definida la estructura para árboles roji-negros, podemos implementar las operaciones de importancia. Por ejemplo, la altura negra.

Definición 2.2 *La altura negra de un árbol roji-negro T denotada como an(T), se define como el número de nodos negros de cualquier trayectoria desde la raíz (sin contarla) hasta cualquier hoja.*

Obsérvese que esta definición es correcta, debido a la segunda invariante de un árbol roji-negro basta con revisar un camino. La función que corresponde al cálculo de la altura negra es la siguiente:

```
alturaNegra :: RB a -> Int

alturaNegra E = 0
alturaNegra (T _ E x E) = 1
alturaNegra (T _ (T c a y b) x tr) = if c == B
                                     then 1+alturaNegra (T c a y b)
                                     else alturaNegra (T c a y b)
```

esta función asume que el árbol de entrada es roji-negro, por lo que, en particular recorremos la rama izquierda para hacer el conteo de nodos negros.

La operación de pertenencia tiene como objetivo verificar si un elemento existe en el árbol roji-negro o no, la implementación es exactamente la misma que para el caso de árboles binarios de búsqueda.

```
member :: Ord a => a -> RB a -> Bool

member x E = False
member x (T _ tl y tr)
                 | x<y = member x tl
                 | x>y = member x tr
                 | otherwise = True
```

Esta función recibe como entrada un elemento `x` y un árbol `T`. Cuando `T` es vacío se devuelve `False` inmediatamente puesto que el árbol vacío no tiene elementos.

Cuando `T` no es vacío, se verifica si el elemento `x` es menor que la raíz del árbol de entrada, en cuyo caso se hace la recursión sobre el subárbol izquierdo; si el elemento es mayor, entonces se hace la recursión sobre el subárbol derecho; y si el elemento es igual, entonces devolvemos `True` puesto que el elemento buscado es la raíz del árbol de entrada.

Como una aplicación interesante, los conjuntos finitos se implementan a través de árboles roji.negros. Para poder implementar este tipo de estructuras son necesarias una serie de operaciones cuya definición no es tan simple como en el caso de árboles binarios de búsqueda, puesto que necesitamos respetar las invariantes. Estas operaciones son:

- Inserción, para agregar un nuevo elemento al árbol roji-negro.
- Borrado, para eliminar un elemento miembro del árbol roji-negro.

Estas operaciones requieren de una función auxiliar de balanceo, la cual garantiza la preservación de las invariantes. A continuación discutimos las funciones de balanceo e inserción dadas por [13, 14]. Cabe mencionar que en dichos artículos se omite la operación de borrado, siendo resuelta esta omisión posteriormente por diversos autores. La revisión y explicación detallada de estas soluciones es uno de nuestros propósitos principales, los cuales desarrollaremos en el siguiente capítulo.

2.1. Inserción

La operación de inserción tiene como función agregar nuevos elementos a un árbol roji-negro. A diferencia de las operaciones de inserción para otro tipo de árboles, en este caso se requiere de una función de balanceo, la cual explicaremos más adelante. Esto se debe a que al insertar un nuevo elemento a un árbol roji-negro queremos que el árbol resultante también sea roji-negro, es decir, debe cumplir con las propiedades invariantes antes descritas. Si se inserta un nodo sin hacer un balanceo, puede ser que el nuevo nodo altere la altura negra, más aún, puede tenerse el caso nodo rojo con hijos rojos, violando las invariantes.

La definición de la función de inserción es la siguiente:

```
insert :: Ord a => a -> RB a -> RB a

insert x s = T B a z b where T _ a z b = ins s
             ins E = T R E x E
```

```
ins s@(T B a y b)
   | x<y = balance (ins a) y b
   | x>y = balance a y (ins b)
   | otherwise = s
ins s@(T R a y b)
   | x<y = T R (ins a) y b
   | x>y = T R a y (ins b)
   | otherwise = s
```

observemos que esta definición se basa en dos funciones auxiliares *balance* e *ins*. La primera función es la encargada de balancear adecuadamente el árbol para evitar la violación de invariante nodo rojo con hijos rojos, mientras que la segunda se ocupa de hacer la inserción del elemento en el árbol mediante recursión. Si se quiere hacer la inserción en el árbol vacío, entonces se inserta el elemento como raíz y se pinta de color rojo, es decir, se crea una hoja roja, esto no causa problemas pues al final la definición de *insert* cambiará el color del árbol devuelto por *ins* [1]. Para el caso en que la inserción sea sobre un árbol de raíz negra, entonces se compara dicho elemento `x` con la raíz `y`. Si `x` es menor que `y`, se inserta `x` en el subárbol izquierdo `a` mediante la función *ins* y se llama a la función de balanceo, la cual se encargará de construir adecuadamente un árbol roji-negro. Si `x` es mayor que `y`, se procede análogamente insertando esta vez el elemento en el subárbol derecho. Por último si `x` es igual a `y`, entonces se deja el árbol intacto pues no se quiere que haya elementos repetidos. Para el caso en que la inserción sea sobre un árbol de raíz roja la función *ins* procede como en un árbol binario de búsqueda común manteniendo el color rojo de la raíz. Recalcamos nuevamente que esto no es relevante, pues la función *insert* se encargará de cambiar el resultado de *ins* a un árbol con raíz negra. Observemos también que en el caso en que se hace la inserción a un nodo rojo en el interior del árbol, tampoco presenta un problema, pues en la recursión ya se debio de haber pasado por un nodo negro, por lo que ya existe una llamada a la función *balance* que arregla la posible violación a la invariante rojo-rojo.

En esta implementación se introduce el uso de los llamados constructores inteligentes (en inglés *smart constructor*), término introducido por Adams en [1]. Un constructor inteligente[2] es una función que juega el papel del constructor normal con la diferencia de que devuelve una estructura que preserva la invariante. En nuestro caso la función *balance* es un constructor inteligente que detecta y repara las violaciones a la primera invariante rojo-rojo, es decir, cuando el constructor original produciría un nodo rojo con algún hijo rojo.

[1] Lo que se busca es que la única violación de la invariante se dé en la raíz del árbol, ya que en este punto es muy fácil pintar el nodo raíz de negro y así ya no nos importa si sus subárboles son rojos. Esto lo podremos observar en las implementaciones que analicemos más adelante.

[2] Decidimos mantener el nombre de constructor inteligente, aunque otra posibilidad sería llamarlo un constructor avanzado o ingenioso.

La definición de *balance* es:

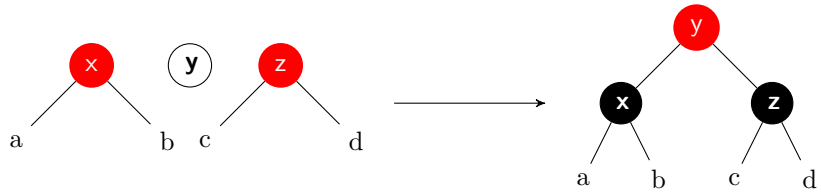
```
balance :: RB a -> a -> RB a -> RB a

balance (T R a x b) y (T R c z d) = T R (T B a x b) y (T B c z d)
balance (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance a x b = T B a x b
```

Explicamos a continuación cada caso de la función *balance*, asumiendo que ambos árboles de entrada son de altura negra n y que al menos uno de los dos es un árbol que viola la invariante rojo-rojo producido por la inserción.

La primera configuración es la siguiente:

```
balance (T R a x b) y (T R c z d) = T R (T B a x b) y (T B c z d)
```

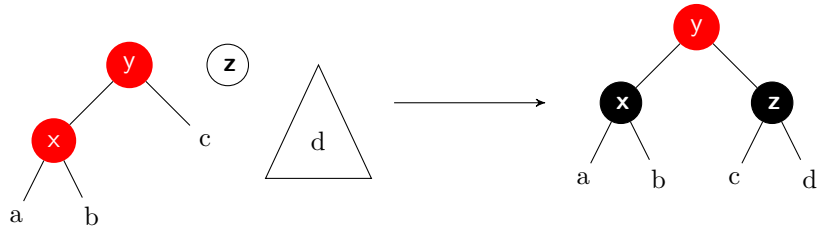
en este caso tenemos de entrada dos árboles con raíz roja en los que puede existir una violación a la invariante rojo-rojo, por lo que la solución es pintar las raíces de dichos árboles, `x` y `z`, de negro y construir un árbol con raíz `y` roja que los tenga de subárboles, tal y como se muestra en la siguiente figura.



La segunda configuración es la siguiente:

```
balance (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
```

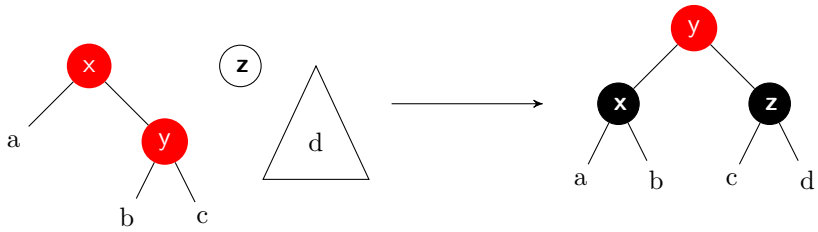
este uno de los casos en el que se muestra la violación de la invariante rojo-rojo de manera explícita, pues hay dos nodos rojos consecutivos. El árbol nuevo se construye tomando a `y` como raíz, cambiando el color de `x` a negro y colocando el nuevo nodo `z` como raíz del subárbol derecho, de modo que los árboles `c` y `d` se encuentran como subárboles de `z`. Obsérvese la siguiente figura.



Obsérvese que este caso corresponde al proceso de rotación a la derecha en una implementación imperativa.
La tercera configuración es:

```
balance (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
```

por lo que estamos ante una violación donde el hijo derecho de la raíz roja es rojo, y se resuelve análogamente al anterior. (Ver la siguiente figura)

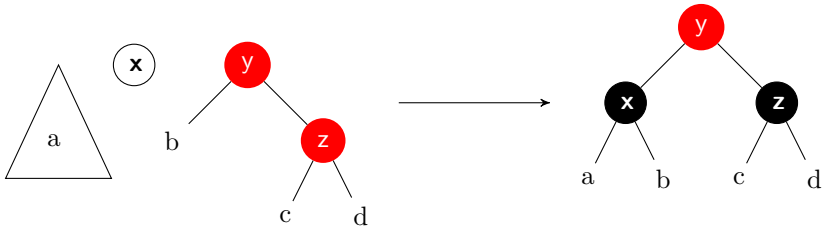


Los siguientes dos casos corresponden a cuando la violación de la invariante se da en el subárbol derecho y se resuelven de manera análoga a los dos anteriores.

La cuarta configuración es:

```
balance a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
```

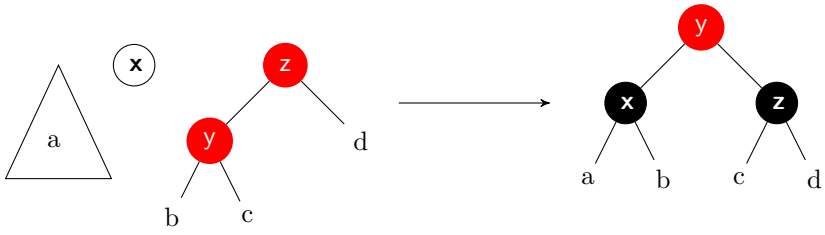
cuya la solución es:



La quinta configuración es:

```
balance a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
```

cuya solución es la siguiente figura:



El último caso de la definición es:

```
balance a x b = T B a x b
```

que corresponde a cuando no hay violación de la invariante, por lo que la función de balanceo llama simplemente al constructor `T`.

Es relevante observar que en cuatro de los cinco casos de la función *balance* la solución es la misma, se construye un árbol con raíz roja y sus dos subárboles tienen sus alturas negras balanceadas. Por lo tanto esta implementación es más simple que las implementaciones típicas de la programación imperativa. La razón es que los cuatro casos peligrosos tratados por nuestra función *balance* se convierten en seis en dichas implementaciones [5], de acuerdo al color del hermano del nodo rojo con hijo rojo, puesto que de esta manera se utilizan menos operaciones de asignación. Al no existir tal operación en el ámbito funcional dicha distinción es innecesaria.

El tipo de árboles roji-negros definidos en este capítulo todavía no es útil para implementar conjuntos, pues si bien ya tenemos definidas las operaciones de pertenencia e inserción de elementos, aún falta definir la operación de borrado, lo cual hacemos en el siguiente capítulo.

Capítulo 3

Operación de borrado

La operación de borrado en árboles roji-negros requiere de mayor cuidado que el que se tiene con las otras operaciones, puesto que debemos mantener la altura negra. Si utilizamos una definición recursiva similiar a la de árboles binarios de búsqueda, el resultado será un árbol que viola la invariante de la altura negra, puesto que la rama que tenía el nodo borrado puede reducir su altura negra. Esto sucede cuando el nodo borrado es negro. En este capítulo desarrollamos cuatro soluciones diferentes al problema de eliminar un elemento de un árbol roji-negro. Las dos primeras utilizan constructores inteligentes y se basan en [11, 9]. La tercera utiliza una aritmética de colores y es una adaptación propia para Haskell de la implementación en Racket dada por [12]. La última utiliza conceptos avanzados de la programación funcional y se basa en [11].

3.1. Constructores inteligentes

De la misma manera en como se implementó la función de inserción, la operación de borrado emplea un constructor inteligente para asegurar que se mantengan las invariantes, es decir, que el resultado de eliminar un elemento de un árbol roji-negro sea también un árbol roji-negro.

Comencemos por la implementación de Stefan Kahrs presentada en [11]. Para esta implementación Kahrs se basa en los constructores y en la función *balance* definidos originalmente por Okasaki, utilizando además tres funciones auxiliares, a saber *balleft*, *balright* y *app*, con que se define a continuación una función *delete* que es la encargada de hacer la eliminación.

```
delete :: Ord a => a -> RB a -> RB a
```

```
delete x t = case del t of {T _ a y b -> T B a y b; _ -> E}
where
del E = E
del (T _ a y b)
     | x<y = delfromLeft a y b
     | x>y = delfromRight a y b
     | otherwise = app a b
delfromLeft a@(T B _ _ _) y b = balleft (del a) y b
delfromLeft a y b = T R (del a) y b

delfromRight a y b@(T B _ _ _) = balright a y (del b)
delfromRight a y b = T R a y (del b)
```

La función *delete* recibe un elemento `x` y un árbol `t`, y devuelve el árbol resultante de eliminar `x` en `t`. Su definición se compone de tres funciones auxiliares locales *del*, $delfromLeft$ y $delfromRight$. La función *del* busca la posición del elemento a eliminar en un árbol dado para posteriormente indicar la función particular que se encargará de hacer la eliminación, la cual puede ser $delfromLeft$ o $delfromRight$. La función $delfromLeft$ elimina un elemento del subárbol izquierdo empleando un constructor inteligente llamado $balleft$. La función $delfromRight$ hace lo propio pero en el caso del subárbol derecho. La función *app* se encarga de pegar los dos subárboles.

Como la función $delfromRight$ es análoga a $delfromLeft$, me enfocaré solo a explicar ésta última. Existen dos casos a analizar. Primero, si el subárbol izquierdo tiene raíz negra se ejecuta la eliminación en éste para posteriormente generar un árbol mediante el constructor inteligente $balleft$. En otro caso, el subárbol tiene raíz roja y simplemente se emplea el constructor `T` generando un árbol rojo, observemos que no es necesario balancear pues sucede lo mismo que en la insercion sobre nodos rojos, en la recursión se llama a la función $balleft$ que es la encargada de arreglar la posible violación a la invariante rojo-rojo.

Explico ahora la función $balleft$, revisando antes la definición de la función auxiliar *red*, cuyo objetivo es cambiar el color de la raíz de un árbol negro a rojo.

```
red :: RB a -> RB a

red (T B a x b) = T R a x b
red _ = error "invariance violation"
```

El mensaje de *error* que emite esta función lo explicaré más adelante.

El constructor inteligente $balleft$ devuelve un árbol balanceado manteniendo de cierta manera el árbol de entrada izquierdo. Su implementación es la siguiente:

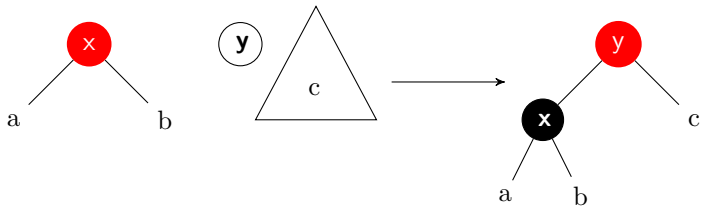
```
balleft :: RB a -> a -> RB a -> RB a

balleft (T R a x b) y c = T R (T B a x b) y c
balleft bl x (T B a y b) = balance bl x (T R a y b)
balleft bl x (T R (T B a y b) z c) =
   T R (T B bl x a) y (balance b z (red c))
```

Analicemos cada caso de la función *balleft*.

En el primer caso

```
balleft (T R a x b) y c = T R (T B a x b) y c
```

se recibe un elemento `y`, y dos árboles de los cuales el primero tiene raíz roja. Se devuelve un árbol rojo, cuya raíz es el elemento de entrada `y`, el subárbol izquierdo es el primer árbol de entrada pero con la raíz recoloreada a negro, y el subárbol derecho es el segundo árbol de entrada. Graficamente este caso de la función se puede observar como en la siguiente figura.

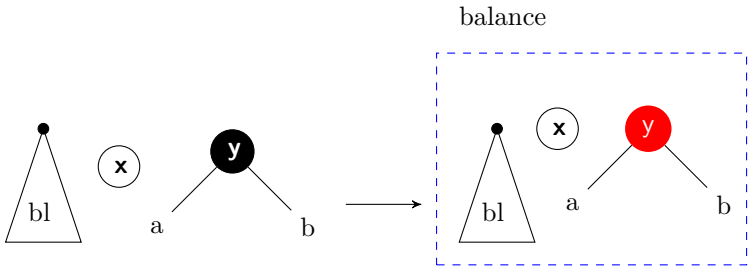


Como se está suponiendo que el árbol izquierdo está desbalanceado porque se elimino un nodo negro, su raíz roja se cambia a negra para recuperar el balanceo, pero además el nuevo árbol debe tener raíz roja `y`, para garantizar que no hay un nuevo desbalanceo con el subárbol derecho `c`.

En el segundo caso se tiene

```
balleft bl x (T B a y b) = balance bl x (T R a y b)
```

La siguiente figura nos muestra graficamente este caso.



El cambio de color en la raíz del segundo árbol reduce su altura negra en uno; haciendo que el primer árbol quede desbalanceado con respecto al segundo árbol, por lo que la función *balance* es necesaria para obtener un nuevo árbol roji-negro.

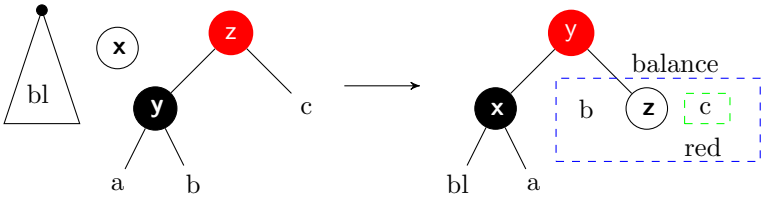
En el último caso

```
balleft bl x (T R (T B a y b) z c) =
   T R (T B bl x a) y (balance b z (red c))
```

la función recibe como argumentos un árbol negro `bl`, un elemento y un árbol rojo con subárboles negros, y se devuelve un árbol rojo tal que:

- la nueva raíz es la raíz del subárbol izquierdo del segundo árbol de entrada `y`.
- El subárbol izquierdo es un árbol negro cuya raíz es el elemento de entrada `x`, su subárbol izquierdo es `bl` y su subárbol derecho es `a`. A la derecha de `y` se encuentra el resultado de balancear un árbol con raíz `z` (la raíz del segundo árbol de entrada), el subárbol derecho `b` que era el subárbol derecho del subárbol izquierdo del segundo árbol de entrada y el subárbol derecho `c` que era el subárbol derecho del segundo árbol de entrada, pero cambiando su raíz a roja.

Todo lo anterior se puede visualizar en la siguiente figura con más claridad.



Observemos que el segundo árbol de entrada es un árbol válido, por lo que la función *red* hace el cambio de color correspondiente. Si el subárbol `c` fuera rojo, claramente el árbol de entrada estaría violando la invariante rojo-rojo, por lo que la función *red* devolvería el error por violación de invariante.

El constructor inteligente *balright* es análogo a *balleft* y se define de la siguiente manera:

```
balright :: RB a -> a -> RB a -> RB a

balright a x (T R b y c) = T R a x (T B b y c)
balright (T B a x b) y bl = balance (T R a x b) y bl
balright (T R a x (T B b y c)) z bl =
   T R (balance (red a) x b) y (T B c z bl)
```

Finalmente, tenemos la función auxiliar *app* que se encarga de pegar dos árboles de manera adecuada y se utiliza en el caso en que el elemento a borrar sea la raíz. En HASKELL su implementación es la siguiente:

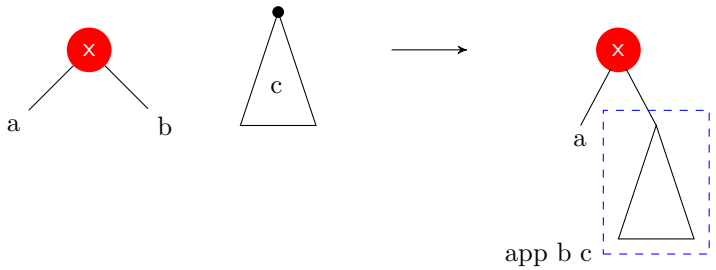
```
app :: RB a -> RB a -> RB a

app E x = x
app x E = x
app (T R a x b) (T R c y d) =
case app b c of
    T R b' z c' -> T R(T R a x b') z (T R c' y d)
    bc -> T R a x (T R bc y d)
app (T B a x b) (T B c y d) =
case app b c of
        T R b' z c' -> T R(T B a x b') z (T B c' y d)
        bc -> balleft a x (T B bc y d)
app a (T R b x c) = T R (app a b) x c
app (T R a x b) c = T R a x (app b c)
```

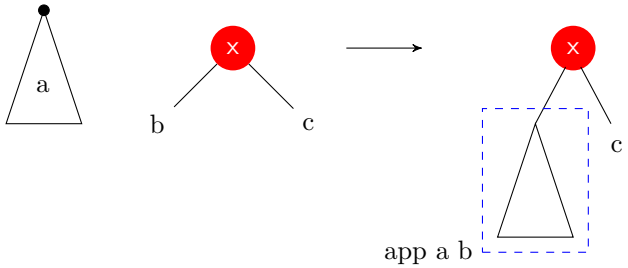
Los primeros dos casos resuelven de la manera obvia el pegado cuando uno de los árboles es vacío.

Los últimos dos casos resuelven el problema de pegar un árbol rojo con uno negro y son:

```
app (T R a x b) c = T R a x (app b c)
app a (T R b x c) = T R (app a b) x c
```

por ejemplo, en el primer caso solo hacemos la llamada recursiva pegando el subárbol derecho del primer árbol de entrada `b` con el segundo árbol de entrada; y análogamente para el segundo caso. Esto se muestra en las siguientes figuras:



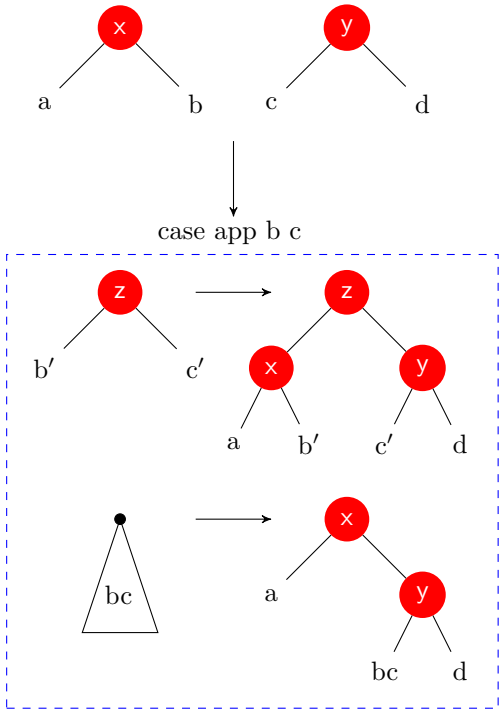


Analicemos ahora los casos más complicados que corresponden a pegar dos árboles con raíces del mismo color. En el caso en que estos árboles sean rojos la definición es:

```
app (T R a x b) (T R c y d) =
case app b c of
    T R b' z c' -> T R(T R a x b') z (T R c' y d)
    bc -> T R a x (T R bc y d)
```

donde se analiza el resultado de pegar el subárbol derecho del primer árbol con el subárbol izquierdo del segundo; si el resultado es un árbol rojo entonces la función devuelve un árbol rojo con subárboles rojos, pegando los subárboles de manera adecuada tal y como se ve en la figura más abajo. Aquí se esta violando explícitamente la invariante rojo-rojo, este problema es resuelto al final por la función *delete* la cual cambia el color de la raíz del árbol devuelto por la función *del* a negro.

Si el resultado de la recursión `app b c` es un árbol rojo, se devuelve un árbol rojo tal que: sus subárboles izquierdo y derecho son los árboles izquierdo y derecho de entrada respectivamente, con la diferencia de que los subárboles `b` y `c` son los subárboles del árbol resultado de la recursión. Si el resultado de la recursión es un árbol negro entonces se devuelve un árbol rojo tal que: el subárbol izquierdo es el subárbol izquierdo del primer árbol de entrada. Mientras que el subárbol derecho se obtiene al sustituir el subárbol izquierdo del segundo árbol de entrada por el resultado de la llamada recursiva correspondiente tal y como se muestra en la figura de abajo.

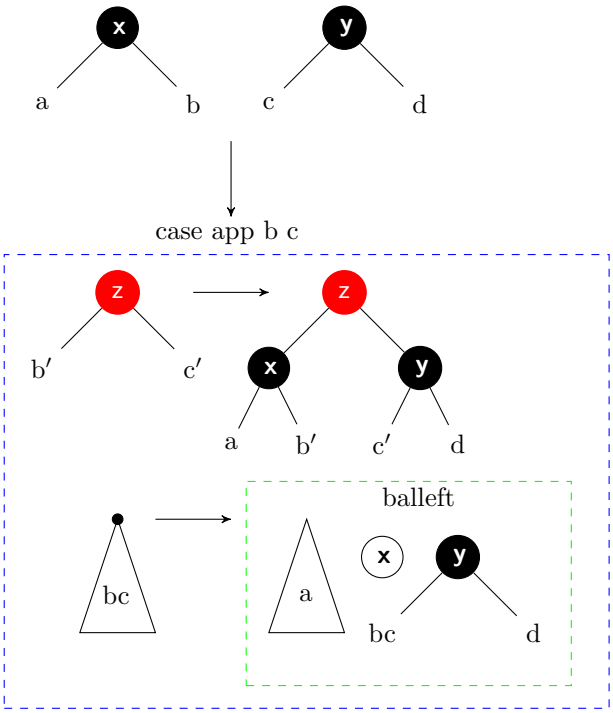


Para el caso en el que pegamos dos árboles negros la definición es la siguiente:

```
app (T B a x b) (T B c y d) =
case app b c of
        T R b' z c' -> T R(T B a x b') z (T B c' y d)
        bc -> balleft a x (T B bc y d)
```

el primer caso es idéntico a cuando los árboles eran rojos produciendo un árbol que no viola la invariante. Mientras que para el segundo caso se requiere una llamada al constructor inteligente $balleft$ puesto que no se conocen los colores de los subárboles `a` y `d` por lo que $balleft$ decidirá como construir el árbol de manera inteligente.

La siguiente figura muestra este caso:



Con esto queda completa la primera implementación del método de borrado, la cual se sirvió del uso de constructores inteligentes, los cuales constuyen un árbol roji-negro válido.

3.2. Constructores Inteligentes II

La siguiente implementación a revisar es la que corresponde al artículo [9] de Ralf Hinze. Esta implementación consiste en agregar un constructor a la definición de árbol roji-negro llamado `Blacken`. La definición es la siguiente:

```
data Color = R | B deriving (Show, Eq)
data RB a = E | Node Color (RB a) a (RB a)
              | Blacken (RB a) deriving (Show)
```

La idea de introducir el nuevo constructor `Blacken` viene del problema mencionado al principio del capítulo, el cual consiste en la posible reducción de la altura negra en alguna rama del árbol al borrar un elemento.

Para solucionar este problema Hinze propone compensar la disminución de la altura negra agregando una capa negra adicional mediante el constructor `Blacken`, es decir, si `t` es un árbol roji-negro entonces `Blacken(t)` es un árbol roji-negro

idéntico a `t` excepto que su raíz se piensa como doblemente negra. Las funciones de balanceo utilizan este constructor hasta llegar al caso en que el nodo doblemente negro, obtenido de borrar un elemento, es el nodo raíz del árbol, por lo que `Blacken` puede eliminarse, obteniéndose un árbol roji-negro convencional. Esta idea se generaliza en la siguiente sección mediante una aritmética de colores.

Empezaremos definiendo algunas funciones auxiliares de utilidad. La función *color* tiene como finalidad obtener el color del árbol de entrada, para el caso del vacío se devuelve negro por la convención de que estos árboles son de dicho color. La función en HASKELL es:

```
color :: RB a -> Color

color E = B
color (Node c _ _ _) = c
```

La siguiente función es *make* que devuelve un árbol con la raíz del color que recibe como entrada. Su implementación funcional es la siguiente:

```
make :: Color -> RB a -> RB a

make c E = E
make c (Node _ l a r) = Node c l a r
```

Continuamos con la función *blacken*, la cual se encarga de colorear de negro la raíz del árbol de entrada. En el caso en que dicho árbol ya era negro se utiliza el constructor `Blacken` para encapsularlo y considerarlo como un árbol de raíz doblemente negra. La implementación en HASKELL se muestra a continuación:

```
blacken :: RB a -> RB a

blacken t | color t == R = make B t
          | otherwise = Blacken t
```

La siguiente función a definir es *paint*, cuya implementación funcional es la que se muestra a continuación:

```
paint :: Color -> RB a -> RB a

paint R t = t
paint B t = blacken t
```

esta función es la que se usará si un nodo de color negro se elimina y tiene el objetivo de agregar una capa negra adicional al árbol de entrada apoyándose de *blacken*. Si el color que se recibe de entrada es rojo entonces el árbol permanece igual. Mientras que si es negro se llama a la función *blacken* la cual encapsula

el árbol para considerarlo como doble negro.

Por último la función *unBlack*, cuya implementación es la que sigue

```
unBlack :: RB a -> RB a

unBlack (Blacken t) = t
unBlack t = t
```

se encarga de eliminar la presencia del constructor `Blacken` delos árboles doblemente negros.

La función *delete* se encarga de hacer la operación de borrado. La implementación es la siguiente:

```
delete :: Ord a => a -> RB a -> RB a

delete a t = unBlack (del t)
             where
             del E = E
             del (Node c l b r)
                                 | a < b = lbal c (del l) b r
                                 | a > b = rbal c l b (del r)
                                 | otherwise = join c l r
```

Cabe mencionar que, por claridad, hemos hecho una ligera modificación a la implementación original de Hinze, sustiyuendo el uso del tipo *Ordery* por los operadores de comparación usuales menor y mayor.

Se observa que esta implementación es similar a la anterior aunque un poco más compacta y elegante. La función *delete* simplemente llama a la función *del*, la cual se encarga del proceso de borrado comparando el nodo a borrar con la raíz del árbol, borrando recursivamente en el subárbol correspondiente y llamando a los constructores inteligentes *lbal*, *rbal* o *join* para construir un árbol roji-negro válido.

Discutimos ahora a detalle la definición del constructor inteligente *lbal*, el cual toma como entrada un color, dos árboles roji-negros y un elemento, y construye inteligentemente un árbol roji-negro nuevo sin la capa doble negra. En esta función estamos suponiendo que uno de los árboles de entrada, el izquierdo, es un árbol encapsulado por *Blacken*, es decir, es un árbol doblemente negro. La implementación es la siguiente:

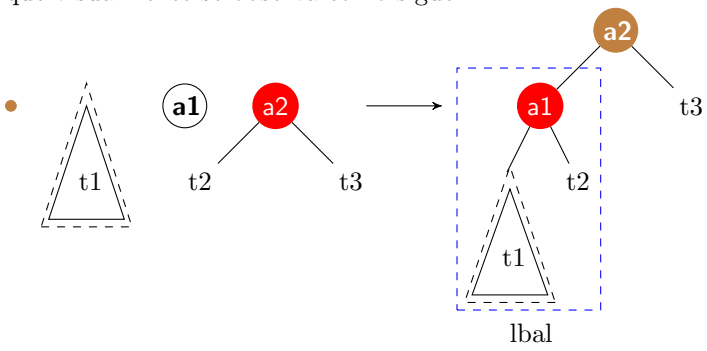
```
lbal :: Color -> RB a -> a -> RB a -> RB a

lbal c (Blacken t1) a1 (Node R t2 a2 t3) =
Node c (lbal R (Blacken t1) a1 t2) a2 t3
```

```
lbal c (Blacken t1) a1 (Node B (Node R t2 a2 t3) a3 t4) =
Node c (Node B t1 a1 t2) a2 (Node B t3 a3 t4)

lbal c (Blacken t1) a1 (Node B t2 a2 (Node R t3 a3 t4)) =
Node c (Node B t1 a1 t2) a2 (Node B t3 a3 t4)

lbal c (Blacken t1) a1 (Node B t2 a2 t3) =
blacken (Node c t1 a1 (Node R t2 a2 t3))

lbal c t1 a1 t2 = Node c t1 a1 t2
```

El primer caso de *lbal* es

```
lbal c (Blacken t1) a1 (Node R t2 a2 t3) =
Node c (lbal R (Blacken t1) a1 t2) a2 t3
```

que visualmente se observa como sigue



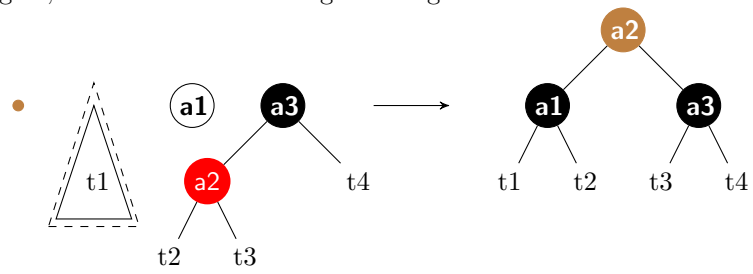
- Color cualquiera dentro de los definidos

aquí se recibe como entrada un color arbitrario representado en la figura con café, un árbol con raíz doble negra, el elemento `a1` y un árbol rojo. Se devuelve un árbol cuya raíz es del color de entrada, su subárbol derecho de éste es el subárbol derecho del segundo árbol de entrada y el subárbol izquierdo de éste es el resultado es la llamada recursiva. Hay que observar que el orden en los argumentos de entrada es significativo, pues en la construcción se supone que los elementos en `t1` son menores que `a1` y que `a1` es menor que los elementos de `t2`, por lo tanto menor a los elementos del segundo árbol de entrada. Además hay que decir que *lbal* es un constructor inteligente que da por hecho el cumplimiento de la invariante de nodos rojos (pues los árboles de entrada no tienen nodos rojos adyacentes) y solo se encarga de solucionar violaciones respecto a la nueva capa negra (nos encargamos de que la capa doble negra quede en la raíz).

El segundo caso es

```
lbal c (Blacken t1) a1 (Node B (Node R t2 a2 t3) a3 t4) =
Node c (Node B t1 a1 t2) a2 (Node B t3 a3 t4)
```

aquí la función *lbal* arma un árbol con raíz del color de entrada y con subárboles negros, tal como se ve en la siguiente figura.

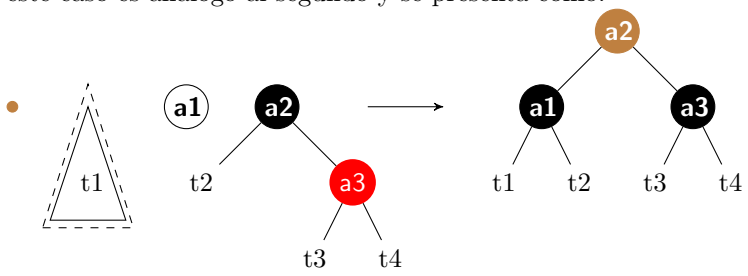


- Color cualquiera dentro de los definidos

Observemos que en este caso lo que se está buscando es de alguna manera compensar la altura negra que le estamos quitando al árbol doblemente negro cuando desaparecemos el constructor `Blacken`, es por esta razón que a `a1` lo pintamos de negro.
El tercer caso es análogo al anterior.

```
lbal c (Blacken t1) a1 (Node B t2 a2 (Node R t3 a3 t4)) =
Node c (Node B t1 a1 t2) a2 (Node B t3 a3 t4)
```

éste caso es análogo al segundo y se presenta como:



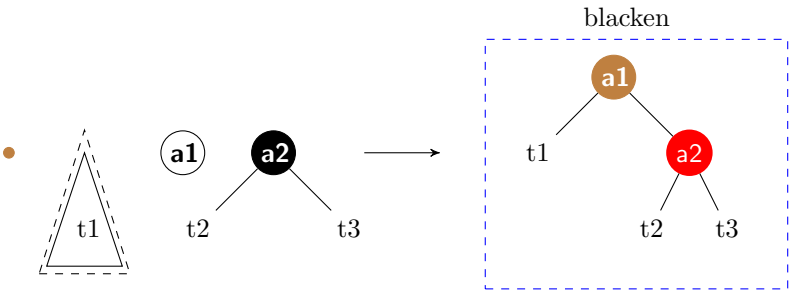
- Color cualquiera dentro de los definidos

El cuarto caso es

```
lbal c (Blacken t1) a1 (Node B t2 a2 t3) =
blacken (Node c t1 a1 (Node R t2 a2 t3))
```

donde los árboles `t2` y `t3` son necesariamente negros debido al orden de las ecuaciones.

Gráficamente este caso se visualiza de la siguiente manera

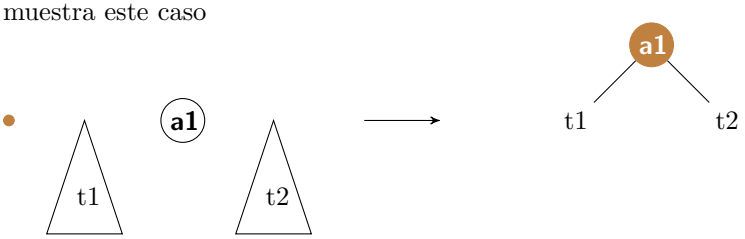


- Color cualquiera dentro de los definidos

El último caso es

```
lbal c t1 a1 t2 = Node c t1 a1 t2
```

y corresponde a cuando el árbol izquierdo no es doble negro, por lo que la función *lbal* simplemente llama al constructor original *Node*. La siguiente figura muestra este caso



- Color cualquiera dentro de los definidos

Pasemos a la función *rbal*, la cual es similar a la función *lbal* y construye un árbol roji-negro a partir de un árbol (desarmado), cuyo subárbol derecho era un árbol doble negro. Para esta función solo dare el código en HASKELL, pues la idea es la misma que para *lbal*. La implementación es la que se muestra a continuación:

```
rbal :: Color -> RB a -> a -> RB a -> RB a

rbal c (Node R t1 a1 t2) a2 (Blacken t3) =
Node c t1 a1 (rbal R t2 a2 (Blacken t3))

rbal c (Node B t1 a1 (Node R t2 a2 t3)) a3 (Blacken t4) =
Node c (Node B t1 a1 t2) a2 (Node B t3 a3 t4)

rbal c (Node B (Node R t1 a1 t2) a2 t3) a3 (Blacken t4) =
Node c (Node B t1 a1 t2) a2 (Node B t3 a3 t4)
```

```
rbal c (Node B t1 a1 t2) a2 (Blacken t3) =
    blacken (Node c (Node R t1 a1 t2) a2 t3)

rbal c t1 a1 t2 = Node c t1 a1 t2
```

En el caso en que el elemento a borrar es la raíz del árbol dado, la función *del* requiere pegar los subárboles, lo cual se hace mediante la función *join* que explicamos a continuación.

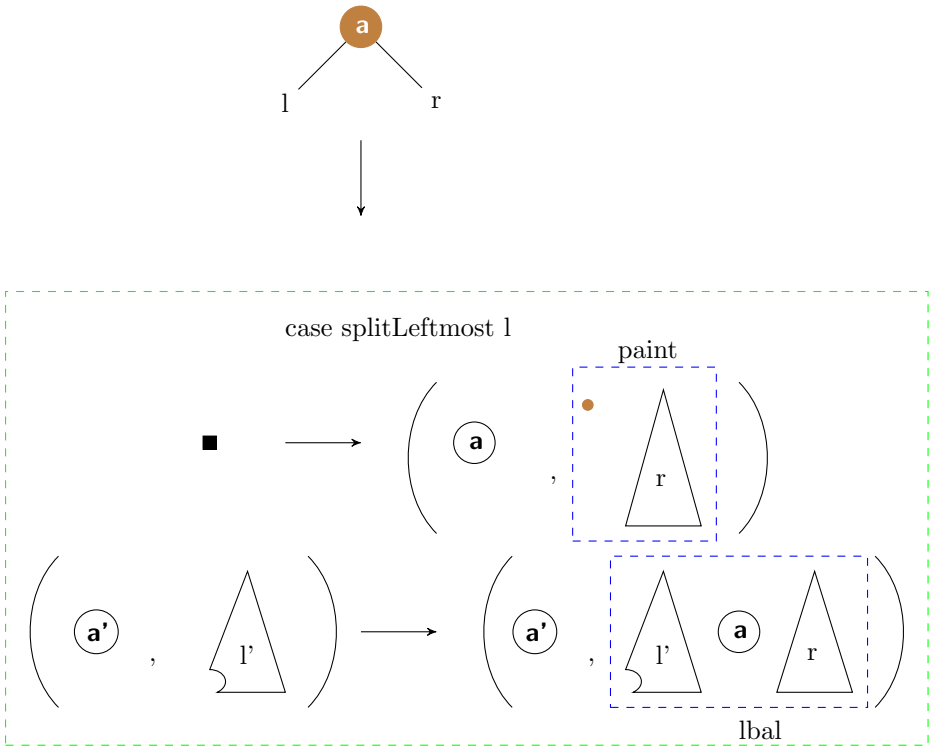
```
join :: Color -> RB a -> RB a -> RB a

join c l r = case splitLeftmost r of
             Nothing -> paint c l
             Just (a, r') -> rbal c l a r'
```

La función *join* recibe como entrada el color y los subárboles del árbol al que le borramos la raíz y a partir de éstos construye inteligentemente un nuevo árbol roji-negro. Esta función se sirve de la función $splitLeftmost$, la cual separa un árbol roji-negro en su elemento más a la izquierda y el árbol que resulta de borrar dicho elemento. En el caso en que el subárbol derecho r sea vacío, (es decir, cuando $splitLeftmost$ devuelve `Nothing`) la función *join* simplemente devuelve el árbol izquierdo pero pintado con el color del árbol original. En otro caso *join* devuelve un árbol roji-negro construido por *rbal* en donde la raíz y el subárbol derecho se obtuvieron mediante la llamada a la función $splitLeftmost$ en el subárbol derecho original.

La función $splitLeftmost$ se implementa como sigue:

```
splitLeftmost :: RB a -> Maybe (a ,(RB a))

splitLeftmost E = Nothing
splitLeftmost (Node c l a r) = case splitLeftmost l of
                               Nothing -> Just (a, (paint c r))
                               Just (a', l') ->
                                        Just (a', (lbal c l' a r))
```

En el caso en que el árbol de entrada sea vacío devolvemos `Nothing` puesto que el árbol vacío no tiene nodos más a la izquierda. Si el subárbol izquierdo del árbol de entrada es vacío, entonces devolvemos la raíz como el nodo más a la izquierda junto con el subárbol derecho pero pintado del color del árbol de entrada. Si el subárbol izquierdo del árbol de entrada es no vacío, entonces se devuelve el elemento más a la izquierda de dicho subárbol, junto con el árbol resultado de hacer el balanceo izquierdo pero eliminando el elemento mencionado. Todo lo anterior se muestra en la siguiente imagen:



El punto café representa el color que se recibe como parámetro en la función *paint*.

Con esto se concluye la implementación de la operación de borrado mediante constructores inteligentes dada por Ralf Hinze, y también con esto concluimos esta sección.

3.3. Aritmética de colores

Las implementaciones funcionales de la operación de borrado, mediante el uso de constructores inteligentes desarrolladas en las secciones anteriores, resultan complejas debido principalmente al uso de distintas funciones de balanceo. En esta sección presentamos una implementación de la operación de borrado que sólo utiliza la operación de balanceo original de Okasaki, y que en nuestra opinión resulta más intuitiva y fácil de entender. Esta implementación, desarrollada originalmente en Racket y discutida en [12], utiliza temporalmente dos nuevos colores (doble negro y negro negativo) y se divide en tres fases: eliminación, burbujeo y balanceo. Nuestra aportación, además de la adaptación a Haskell, consta de ciertas mejoras, como la modularización de la fase del burbujeo.

Empezamos dando la definición para el color de los nodos de un árbol, R para rojo, B para negro, BB para doble negro y NB para negro negativo. El tipo de datos es el siguiente:

```
data Color = R | B | BB | NB deriving Show
```

Como cada implementación presentada en esta sección es independiente una de otra, entonces a la definición original que ya se tenía de árboles roji-negros se agrega un nuevo constructor EBB que representa el árbol vacío de color doble negro.

```
data RB a = E | EBB | T Color (RB a) a (RB a) deriving Show
```

La función *redden* es una función auxiliar que se usará para la definición de la función *balance* más adelante; *redden* devuelve un árbol pero cambiando el color de su raíz a rojo, es decir, devuelve el mismo árbol que toma de argumento pero con raíz roja. Si el árbol que recibe es vacío devuelve un error pues al árbol vacío no tiene raíz roja. Además en el coloreado de árboles se pueden violar las invariantes. La función en HASKELL es como sigue:

```
redden:: RB a -> RB a

redden (T c a x b) = T R a x b
redden E = error "invariance violation"
```

Cabe mencionar que en la función no está definida para cuando recibimos de entrada un vacío doble negro, esto por que nunca se cae en este caso.

La función *blacken* es también una función auxiliar que se utilizará por la función principal de borrado y es muy parecida a la función *redden* pues lo que devuelve es el árbol que se pasa de entrada pero con raíz negra. Si el árbol que recibe es vacío, lo deja igual pues éste ya es de color negro; y si recibe el árbol vacío doble negro, devuelve el árbol vacío negro.

```
blacken:: RB a -> RB a

blacken (T c a x b) = T B a x b
blacken EBB = E
blacken E = E
```

A continuación definimos operaciones aritméticas de suma y resta entre algunos de los colores, las cuales servirán de apoyo en distintas fases de la implementación. Estas operaciones las denominamos *aritmética de colores* y son las siguientes.

- $B + B = BB$

- $R + B = B$
- $NB + B = R$
- BB - $B = B$
- B - $B = R$
- R - $B = NB$

Como se observa se trata de operaciones que suman o restan el color negro a otros colores. Como sólo nos interesa sumar o restar el color negro, en vez de las operaciones de suma o resta podemos hablar de las funciones de predecesor y sucesor con respecto al negro. Con esto en mente podemos ordenar los colores de la siguiente forma $NB < R < B < BB$ y definir funciones sucesor y predecesor de acuerdo a las reglas anteriores, por ejemplo como $NB + B = R$ entonces $suc(NB) = R$. Esto facilita la implementación de funciones que operan con los colores y simplifica los cambios de color en el proceso de burbujeo en el árbol. Las funciones sucesor y predecesor son:

```
sucBlack:: Color -> Color

sucBlack NB = R
sucBlack R = B
sucBlack B = BB

predBlack:: Color -> Color

predBlack R = NB
predBlack B = R
predBlack BB = B
```

El siguiente paso es generalizar las funciones sucesor y predecesor a árboles. La función *sucBlackTree* recibe un árbol con raíz de cualquier color y devuelve el árbol que resulta de cambiar el color original de la raíz por su sucesor. Por ejemplo, si recibe un árbol con raíz roja la función nos devolverá el mismo árbol pero con raíz negra. Si la función recibe el árbol vacío E la función nos devuelve el árbol vacío doble negro pues E es negro.

```
sucBlackTree :: RB a -> RB a

sucBlackTree (T c a x b) = (T (sucBlack c) a x b)
sucBlackTree E = EBB
```

Observemos que *sucBlackTree* de `EBB` no está definido pues nunca caemos en ese caso, lo mismo sucede con *sucBlackTree* de `T BB a x b`.

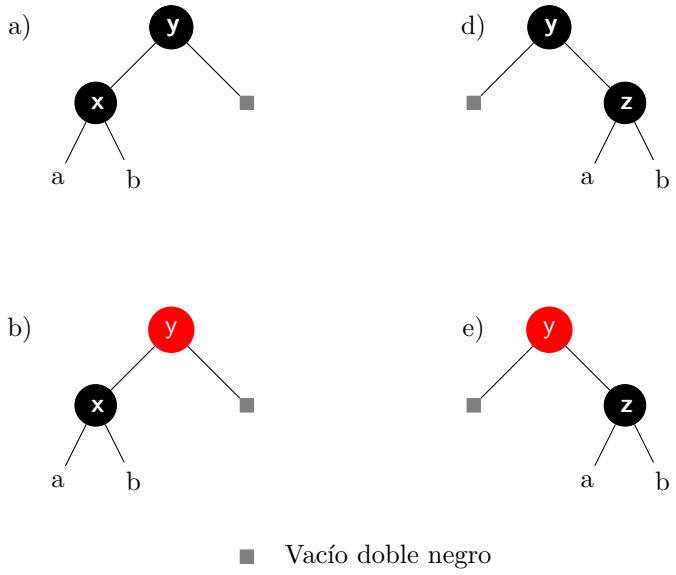
La función *predBlackTree* es análoga a la función *sucBlackTree* pero se encarga de cambiar el color del árbol por su predecesor. Por ejemplo, si recibe un árbol con raíz roja la función nos devolverá el mismo árbol pero con raíz negra negativa.

```
predBlackTree :: RB a -> RB a

predBlackTree (T c a x b) = (T (predBlack c) a x b)
predBlackTree EBB = E
```

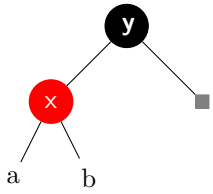
Al igua que sucede con la función *sucBlackTree*, *predBlackTree* tiene casos que no son necesario definir, pues nunca caemos en ellos. Estos casos son *predBlackTree* de `E` y *predBlackTree* de `T NB a x b`.

El caso más simple de la operación de borrado consiste en borrar el elemento `x` de un árbol que sólo contiene a dicho elemento, es decir, de una hoja etiquetada con `x`. Si la hoja es roja, simplemente se devuelve el árbol vacío. Pero si la hoja es negra debemos ser cuidadosos pues borrar el nodo `x` afectará la altura negra. Para solucionar este problema se introduce el árbol vacío doble negro, el cual debe eliminarse en el resultado final. La operación de burbujeo es un proceso que busca mover nodos de color doble negro de hijos a padres y si es posible, eliminarlos totalmente. Resulta conveniente explicar en este momento la función de burbujeo antes de la operación de borrado.

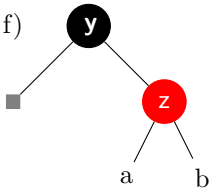
Las siguientes figuras muestran las configuraciones posibles en donde aparece un nodo de color doble negro. Primero mostramos los casos correspondientes al árbol vacío doble negro.



c)



f)

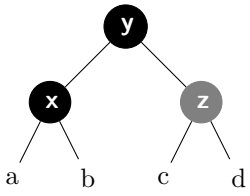


■ Vacío doble negro

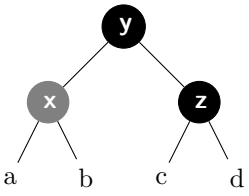
La columna de la derecha, muestra los casos en que el árbol vacío doble negro figura a la izquierda, y la columna de la izquierda el caso contrario.

El proceso de burbujeo se encargará de eliminar el árbol vacío doble negro, recoloreando un nodo superior con el color doble negro, dando como resultado los siguiente casos para continuar el burbujeo:

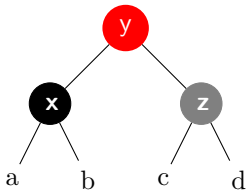
i)



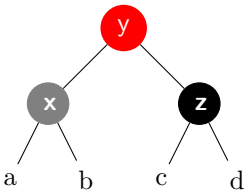
iv)



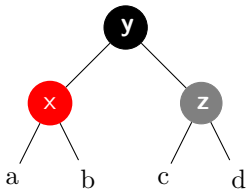
ii)



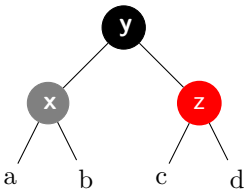
v)



iii)



vi)

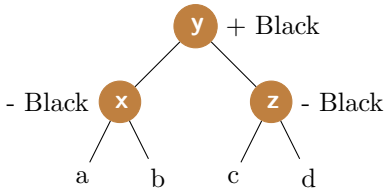


● Nodo doble negro

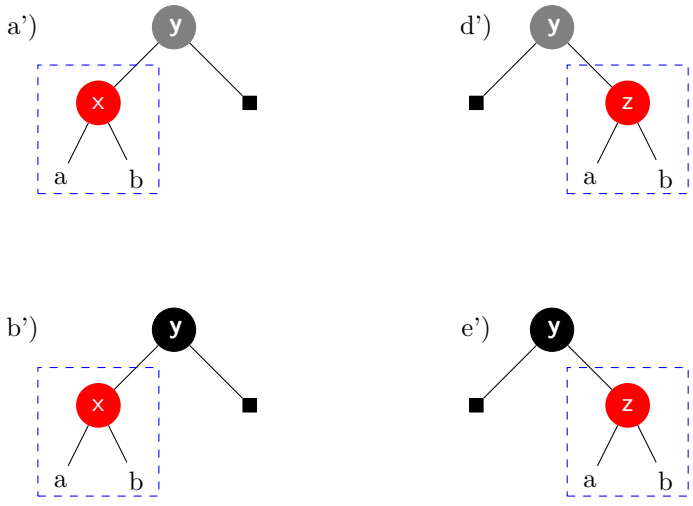
La columna de la derecha muestra el caso donde el nodo doble negro figura en

el subárbol izquierdo, mientras que en la columna de la izquierda se muestra el caso contrario.

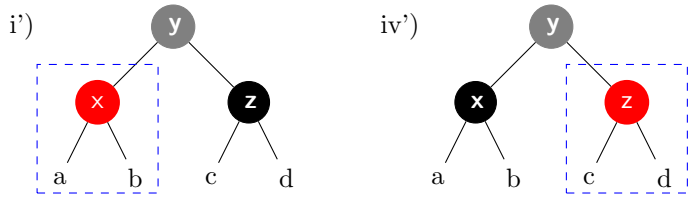
La solución a la aparición del nodo doble negro consiste en aplicar la aritmética de colores de modo tal que al nodo padre se le sumará un color negro, mientras que a los nodos hijos se les restará el mismo color, como se muestra a continuación.

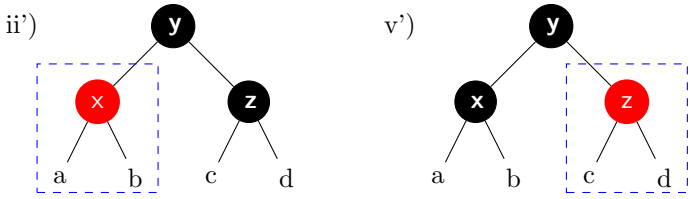


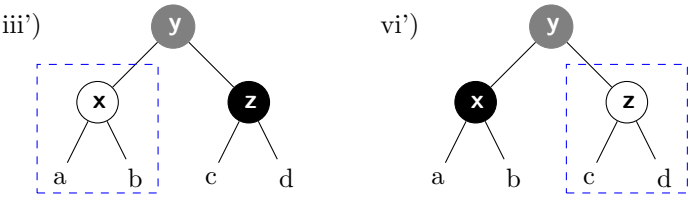
Dando como resultado los casos que muestran en las figuras de abajo. Primero para los casos que involucran al árbol vacío doble negro y después para un nodo de color doble negro. En cualquier caso el árbol corresponde a la transformación aritmética del árbol que está en la misma posición que en las figuras anteriores. La línea punteada indica los casos en que es necesario aplicar la función de balanceo para eliminar cualquier violación de invariantes. Hay que recordar que en esta implementación se pueden tener violaciones del estilo rojo-rojo (que son las originales) y NB-B (negro negativo-negro).



- Nodo doble negro
- Nodo negro negativo







- Nodo doble negro
- Nodo negro negativo

Observemos que hay dos casos en los que al aplicar el burbujeo el nodo doble negro desaparece por completo.

La implementación en HASKELL se observa como sigue:

```
bubble:: RB a -> RB a

bubble E = E
bubble EBB = EBB
bubble (T B (T B a x b) y EBB) = balance BB (T R a x b) y E
bubble (T R (T B a x b) y EBB) = balance B (T R a x b) y E
bubble (T B (T R a x b) y EBB) = balance BB (T NB a x b) y E
bubble (T B EBB y (T B a z b)) = balance BB E y  (T R a z b)
bubble (T R EBB y (T B a z b)) = balance B E y  (T R a z b)
bubble (T B EBB y (T R a z b)) = balance BB E y  (T NB a z b)
bubble (T c (T BB a y b) x r) =
balance (sucBlack c) (predBlackTree (T BB a y b)) x (predBlackTree r)
bubble (T c l x (T BB a y b)) =
balance (sucBlack c) (predBlackTree l) x (predBlackTree (T BB a y b))
bubble (T c a x b) = (T c (bubble a) x (bubble b))
```

El primer caso corresponde a burbujear el árbol vacío, el segundo a burbujear el árbol vacío doble negro. Los siguientes seis corresponden a las figuras de las páginas 34 y 36. Las figuras de las páginas 35 y 37 corresponden a los dos siguientes casos (se reducen a dos debido al uso de las funciones para la aritmética de colores). El último caso se encarga de hacer la recursión. Observemos que la función *bubble* depende de la función *balance*, por lo que al burbujear ya estamos cumpliendo también con la fase de balanceo.

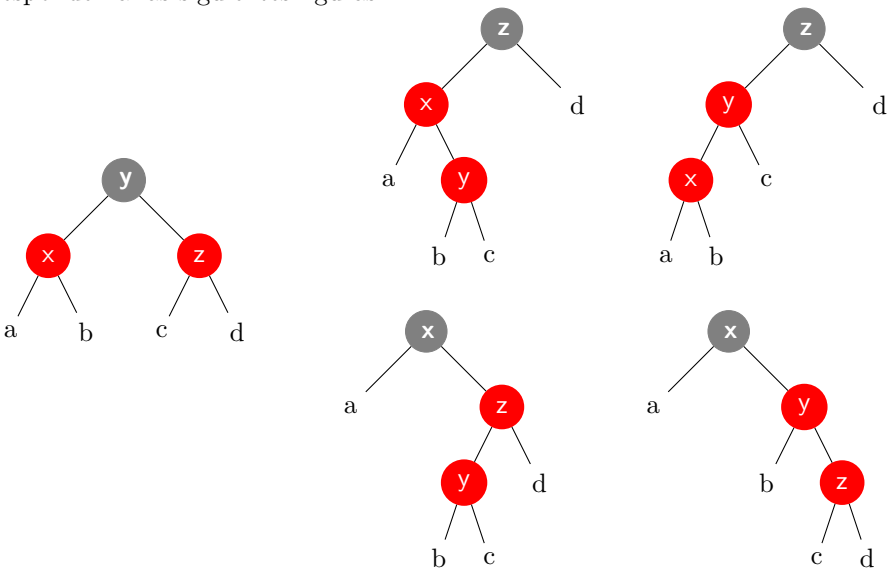
Hay que mencionar que el procedimiento original de burbujeo implementado en Racket no es recursivo, razón por la cual se llama a esta función en tres procedimientos: la operación de borrado, la operación que remueve la raíz y la operación que remueve el elemento máximo en un árbol. Como aquí nos interesa hacer el burbujeo una sola vez, con el fin de modularizar la implementación, nuestra definición de la operación de burbujeo es recursiva.

La función *balance* extiende a la función original de Okasaki (definida en la página 14) agregando los casos para cuando se tiene un nodo de color doble negro y para desaparecer los nodos negros negativos que, de igual forma que los nodos rojos, violan las invariantes. La implementación es como sigue:

```
balance :: Color -> RB a -> a -> RB a -> RB a

balance B (T R a x b) y (T R c z d) =
     T R (T B a x b) y (T B c z d)
balance B (T R (T R a x b) y c) z d =
     T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d =
     T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) =
```

```
     T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) =
     T R (T B a x b) y (T B c z d)
balance BB (T R a x b) y (T R c z d) =
     T B (T B a x b) y (T B c z d)
balance BB (T R (T R a x b) y c) z d =
     T B (T B a x b) y (T B c z d)
balance BB (T R a x (T R b y c)) z d =
     T B (T B a x b) y (T B c z d)
balance BB a x (T R b y (T R c z d)) =
     T B (T B a x b) y (T B c z d)
balance BB a x (T R (T R b y c) z d) =
     T B (T B a x b) y (T B c z d)
balance BB a x (T NB (T B b y c) z (T B d v e)) =
     T B (T B a x b) y (balance B c z (redden (T B d v e)))
balance BB (T NB (T B d v e) x (T B b y c)) z a =
     T B (balance B (redden (T B d v e)) x b) y (T B c z a)
balance c a x b = T c a x b
```

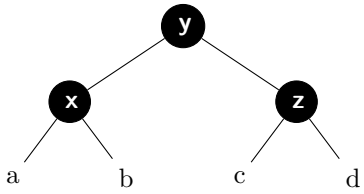
A continuación analizamos esta definición mostrando nuevamente dibujos ilustrativos. Los primeros cinco casos corresponden a la definición de Okasaki discutidos previamente. Los siguientes cinco casos definen cómo restaurar la primera invariante ante una violación rojo-rojo donde el nodo padre es doble negro y corresponden a las siguientes figuras:



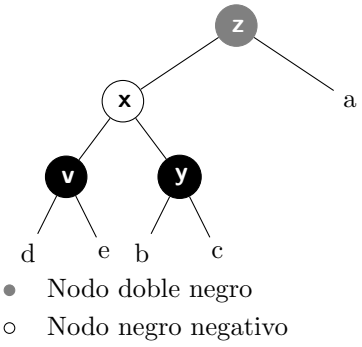
- Nodo doble negro

En todos los casos de la figura anterior la solución es la misma y se muestra en

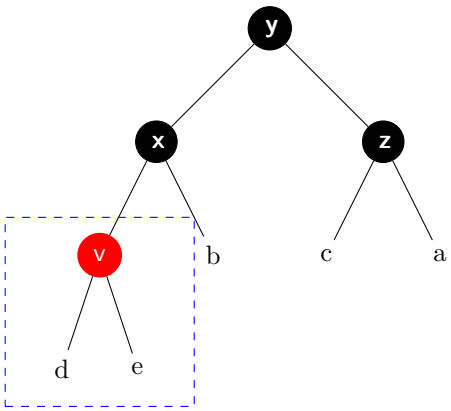
la siguiente figura:



Si un nodo negro negativo aparece como resultado del burbujeo, entonces necesariamente su nodo padre es doble negro y sus hijos son negros (esto se debe a la aritmética de colores). Existen dos posibilidades, que en la función *balance* corresponden al antepenúltimo y último caso, ilustramos aquí uno de ellos, a saber cuando el nodo negro negativo está en el subárbol izquierdo. Esta situación se muestra en la siguiente figura:



La solución dada por la función de balanceo es la siguiente:

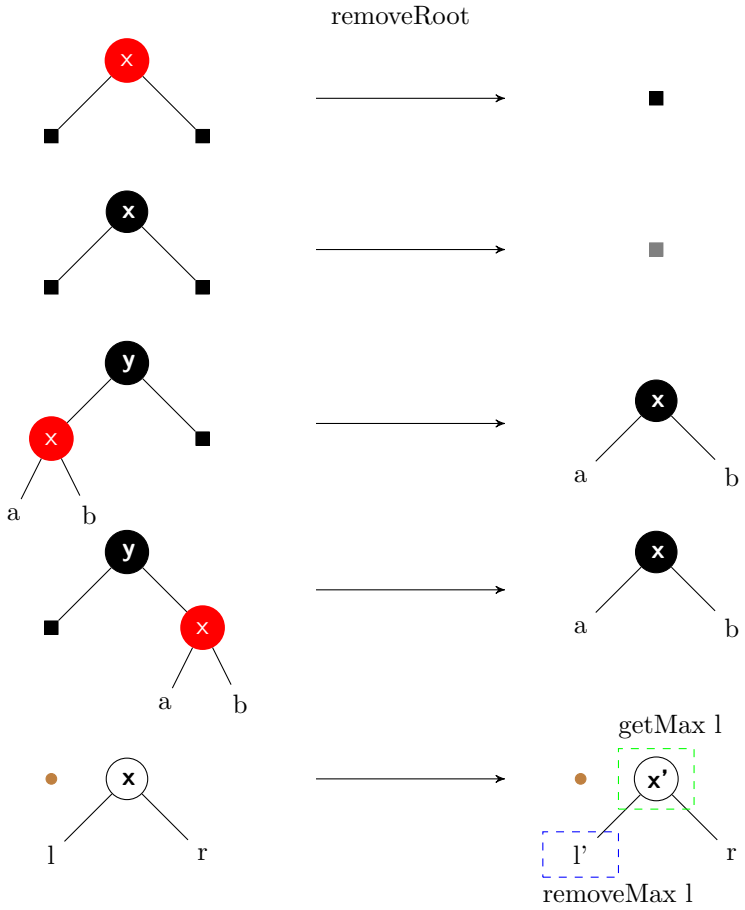


Observemos que en esta configuración es posible crear violaciones rojo-rojo, las cuales deben ser corregidas por la misma función de balanceo, con lo cual ésta se convierte en una definición recursiva.

La siguiente función a discutir es la función $removeMax$, que tiene como propósito eliminar el elemento máximo de un árbol. La implementación es como sigue:

```
removeMax :: RB a -> RB a

removeMax E = E
removeMax (T c l x E) = removeRoot (T c l x E)
removeMax (T c l x r) = (T c l x (removeMax r))
```

La función *removeRoot*, se encarga de eliminar la raíz de un árbol, uniendo los subárboles de manera adecuada. Los casos sencillos corresponden a cuando uno o ambos subárboles son vacíos, y son los primeros cuatro que se muestran en la siguiente figura, nótese que en *removeMax* sólo se usa el tercer caso, sin embargo a continuación presentamos la definición completa:



■ Vacío doble negro

● Color cualquiera dentro de los definidos

En el quinto caso, en que los dos subárboles son no vacíos, se busca y elimina el nodo con valor máximo del subárbol izquierdo, mediante las funciones $getMax$ y $removeMax$, respectivamente. Este valor máximo se utiliza como nodo raíz, dejando intacto al subárbol derecho.

En el procedimiento original en RACKET, se hace un tipo de burbujeo, pero consideramos que el burbujeo debe hacerse después de hacer la eliminación del nodo que deseamos, pues en este momento puede aparecer el color doble negro como hoja y con esto burbujeamos una sola vez, obteniendo la modularización de la implementación original. En HASKELL la implementación es la siguiente:

```
removeRoot:: RB a -> RB a

removeRoot (T R E x E) = E
removeRoot (T B E x E) = EBB
removeRoot (T B (T R a x b) y E) = (T B a x b)
removeRoot (T B E y (T R a x b)) = (T B a x b)
removeRoot (T c l x r) = (T c (removeMax l) (getMax l) r)

getMax :: RB a -> a

getMax (T c l x E) = x
getMax (T c l x r) = getMax r
```

La siguiente función auxiliar llamada *delete* es la encargada de eliminar el nodo deseado, y corresponde esencialmente a la función de borrado en un árbol binario de búsqueda. La implementación funcional es la siguiente:

```
delete :: Ord a => a -> RB a -> RB a

delete x E = E
delete x (T c a y b)
  | member x (T c a y b) = if x < y then (T c (delete x a) y b)
                  else if x > y then (T c a y (delete x b))
                  else removeRoot (T c a y b)
  | otherwise = (T c a y b)
```

Esta función recibe dos argumentos, el elemento que deseamos eliminar y el árbol del cual queremos eliminarlo. Si el elemento dado pertenece al árbol entonces lo busca para eliminarlo, en caso contrario no le hace nada al árbol de entrada. La operación de borrado incluye una búsqueda recursiva. Si el elemento que deseamos eliminar es menor que el elemento raíz del árbol, entonces se hace la llamada recursiva en el subárbol izquierdo; si es mayor, entonces se hace la llamada recursiva en el subárbol derecho; y si es igual, se realiza la operación de borrado mediante la función *removeRoot*. Obsérvese que esta operación *delete* puede generar un árbol que viole las invariantes, debido a que *removeRoot* puede introducir nodos doble negro, específicamente, esta función introduce árboles

vacíos doble negro.

La operación de borrado definitiva combina a todas las operaciones anteriores de manera modular y su definición es:

```
deleteRB :: Ord a => a -> RB a -> RB a

deleteRB x t = blacken (bubble (delete x t))
```

De acuerdo a este código la operación de borrado se realiza secuencialmente como sigue: Primero se elimina el nodo deseado del árbol; como ya se dijo, esta operación puede producir nodos de color doble negro, por lo que debe de aplicarse la operación de burbujeo para eliminarlos (hay que recordar que al burbujear también se balancea el árbol); por último, nos aseguramos de que la raíz del árbol resultante sea negra aplicando la función *blacken*.

Con esto queda terminada la implementación de la operación de borrado mediante el uso de la aritmética de colores.

3.4. Captura de invariantes con tipos anidados

Las tres implementaciones ya presentadas permiten construir árboles roji-negros inválidos, es decir, podemos definir un árbol `t` de tipo `RB a` de tal forma que `t` viole las invariantes. Por ejemplo: `T R (T R E y E) x E`.

Esto implica que en diversas aplicaciones se tenga que implementar una función auxiliar que verifique si un árbol `t` de tipo `RB a` es en realidad un árbol roji-negro válido. En esta sección presentamos una implementación de árboles roji-negros utilizando conceptos sofisticados de la programación funcional, los cuales garantizan el cumplimiento de las invariantes de manera estática, es decir, en tiempo de compilación. Esto se logrará definiendo un tipo de árboles roji-negros (`RB a`) mediante la técnica de tipos anidados o heterogéneos, lo cual nos permitirá garantizar que si `t` es un elemento de tipo `RB a`, entonces `t` es un árbol roji-negro válido.

Los tipos anidados o heterogéneos son un tipo de dato recursivo parametrizado en cuya definición el parámetro cambia en la llamada recursiva. Por ejemplo, el siguiente es un tipo de dato anidado que define árboles perfectos, es decir, árboles binarios completos cuyos subárboles tienen la misma altura:

```
data Perfect a = Zero a | Succ (Perfect (a,a))
```

Hay que notar que esta definición genera una familia infinita de tipos parametrizados y que, a diferencia de un tipo homogéneo o no anidado, como el tipo de listas comunes, resulta imposible aislar a un elemento de esta familia, por ejemplo *Perfect a* requiere de *Perfect (a,a)*. Para profundizar más sobre esta

clase de tipos de datos se recomienda revisar el capítulo 10 de [13].

La implementación que desarrollamos a continuación es una versión detallada de la original discutida por Kahrs en el artículo [11]. Empecemos definiendo el siguiente constructor de tipo

```
data EmptyT a = E deriving Show
```

los constructores de tipo son declaraciones *data* que definen tipos dependientes de uno o más parámetros y por lo tanto pueden considerarse como operadores que transforman tipos en tipos. En este caso $EmptyT$ es un constructor de tipo, puesto que dado un tipo cualquiera a, $EmptyT\ a$ es nuevamente un tipo, con esto en mente podemos decir que $EmptyT$ es una función que transforma al tipo a en el tipo de árboles vacíos de elementos de tipo a, el cual por supuesto, tiene un único elemento. A continuación se definen los siguientes constructores:

```
type Tr t a = (t a,a,t a)
data PRed t a = C (t a) | R (Tr t a)
```

lo primero y lo más importante que hay que notar en estas declaraciones es la expresión `t`, que introducimos para denotar un constructor arbitrario de tipos de árbol, es decir, `t a` es un tipo de árboles con elementos en `a`, observemos que en anteriores implementaciones se utilizaban constructores particulares mientras que con esta implementación se busca la parametrización de constructores. El tipo `Tr t a` tiene como función construir árboles *desarmados* (una terna donde los extremos son árboles de tipo `t a` y el centro es la raíz de tipo `a`), a partir de un constructor de árboles `t` y un tipo de elementos `a`. Por otra parte, un elemento del tipo `PRed t a` es un árbol, al que denominamos *potencialmente rojo*, construido a partir de un árbol desarmado del tipo `Tr t a` o a partir del constructor de árboles `t`. Con potencialmente rojo nos referimos a que estos árboles pueden tener raíz roja (aquellos de la forma `R (t1 , x, t2 )` donde `t1 ,t2 :: t a` y `x:: a`), o bien pueden ser árboles de tipo `t a` encapsulados mediante el constructor `C`. Recordemos que un árbol de color negro, (es decir, de raíz negra) puede tener subárboles de color rojo o negro, es por eso que los dos constructores del tipo `PRed t a` son `C` y `R`. En particular el constructor `C` tiene como propósito incrustar árboles de color negro en el tipo de árboles de color potencialmente rojo, que es exactamente el tipo de árboles permitidos como subárboles de nodos rojos.

El siguiente tipo que se define es

```
data AddBLayer t a = B(Tr(PRed t) a) deriving Show
```

un elemento de este tipo se construye tomando un árbol desarmado con subárboles potencialmente rojos unidos mediante una raíz negra.

Observemos que tanto en `PRed` como en `AddBLayer`, el nombre del constructor denota el color del nuevo árbol.

Después de todas las definiciones anteriores, se define un tipo de datos que se encarga de hacer la construcción de un árbol roji-negro a partir del constructor de árboles arbitrario `t`.

```
data RBT t a = Zero (t a) | Suc (RBT (AddBLayer t) a)
```

el nombre de los constructores `Zero` y `Suc` (Sucesor), se eligieron para codificar la altura negra de un árbol, ésta es una aportación nuestra.

Observemos que `RBT t a` define a una familia de tipos anidados, llamados de *orden superior*, puesto que el parámetro `t`, que cambia en la definición, no es un tipo sino un constructor de tipos.

Veamos ahora cómo se ven los tipos recién definidos cuando el constructor arbitrario `t` se sustituye por el constructor de árboles vacíos `EmptyT`.

```
type Tr EmptyT a = (EmptyT a,a,EmptyT a)
```

por lo que un elemento de este tipo es una tupla de la forma `(E, x, E)`,es decir, es un árbol desarmado con subárboles vacíos y raíz `x`, dicho de otra forma es una hoja desarmada. Por otro lado,

```
data PRed EmptyT a = C (EmptyT a) | R (Tr EmptyT a)
```

un elemento de este tipo es un árbol vacío encapsulado `C(E)`, o bien, un árbol rojo de la forma `R(E,x,E)`, es decir, es una hoja roja.

```
data AddBLayer EmptyT a = B(Tr(PRed EmptyT) a)
```

un elemento de este tipo es un árbol negro construido a partir de una tupla de árboles desarmados potencialmente rojos, aquí se tienen cuatro casos posibles de árboles negros de altura negra 1 que son:

- `B(R(E,a,E),a,R(E,a,E))`
- `B(C(E),a,C(E))`
- `B(C(E),a,R(E,a,E))`
- `B(R(E,a,E),a,C(E))`

Finalmente, el constructor de árboles roji-negros a partir de `EmptyT` es:

```
data RBT EmptyT a = Zero (EmptyT a) | Suc (RBT (AddBLayer EmptyT) a)
```

los elementos de este tipo son, o bien, un árbol vacío de altura negra cero, `Zero(E)`; o bien, alguno de los siguientes casos

- `Suc(Zero(B(R(E,a,E),a,R(E,a,E))))`
- `Suc(Zero(B(C(E),a,C(E))))`
- `Suc(Zero(B(C(E),a,R(E,a,E))))`
- `Suc(Zero(B(R(E,a,E),a,C(E))))`

Notemos que la aplicación de los constructores `Suc` o `Zero` es necesaria para obtener elementos del tipo `RBT (EmptyT a)`, y que la altura negra de estos árboles es 1, la cual está explícita en la representación del árbol (mediante `Suc (Zero)`). La aplicación del constructor `Suc` requiere de un parámetro de tipo `RBT (AddBLayer EmptyT) a`, el cual de acuerdo a la definición es el siguiente:

```
data RBT (AddBLayer EmptyT) a = Zero ((AddBLayer EmptyT) a)
                      | Suc (RBT (AddBLayer (AddBLayer EmptyT)) a)
```

donde justo los casos

- `Zero(B(R(E,a,E),a,R(E,a,E)))`
- `Zero(B(C(E),a,C(E)))`
- `Zero(B(C(E),a,R(E,a,E)))`
- `Zero(B(R(E,a,E),a,C(E)))`

son los parámetros del constructor `Suc` en los ejemplos anteriores.

Con esto se garantiza que un nodo rojo no tenga hijos rojos, además por el tipo de construcción también se garantiza que no se viola la invariante sobre las alturas negras.

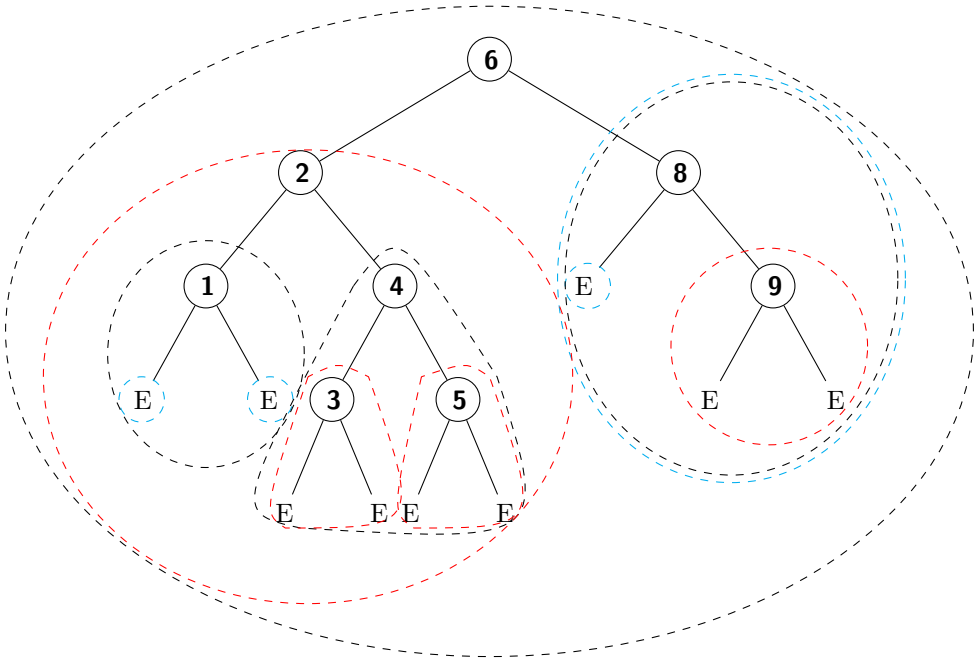
De la misma manera que en los ejemplos anteriores, podemos construir cualquier árbol roji-negro definimos, por lo tanto, el tipo de árboles roji-negros como

```
type RBTree a = RBT EmptyT a
```

por ejemplo, el árbol de altura negra 2 de la página 10 se implementa como

```
Suc(Suc(Zero(B(R(B(C(E),1,C(E)),2,B(R(E,3,E),4,R(E,5,E))),
             6,
             C(B(C(E),8,R(E,9,E)))))))
```

y corresponde a la siguiente figura



donde las líneas punteadas en negro, rojo y azul corresponden a los constructores `B`, `R` y `C` respectivamente. Debido al uso de tipos anidados, esta definición de árboles roji-negros captura las invariantes de forma estática, es decir, si `t` es un elemento del tipo `RBT t a`, entonces `t` es un árbol roji-negro válido. El precio a pagar por el uso de esta clase de tipos es que las funciones que los involucran requieren de razonamientos y técnicas sofisticadas, como la llamada recursión polimórfica. Mostramos esto mediante la definición de la función de pertenencia en un árbol roji-negro.

```
member :: Ord a => a -> RBTree a -> Bool

member x t = rbmember x t (\ _ -> False)

rbmember :: Ord a => a -> RBT t a -> (t a->Bool) -> Bool

rbmember x (Zero t) m = m t
rbmember x (Suc u) m = rbmember x u (ablmem x m)

ablmem :: Ord a => a -> (t a->Bool) -> AddBLayer t a -> Bool

ablmem x m (B(l,y,r))
| x<y = prmem x m l
| x>y = prmem x m r
```

```
| otherwise = True

prmem :: Ord a => a -> (t a->Bool) -> PRed t a->Bool

prmem x m (C t) = m t
prmem x m (R(l,y,r))
                    | x<y = m l
                    | x>y = m r
                    | otherwise = True
```

Lo complicado de esta definición radica en el hecho de que el tipo `RBTree a` es un caso particular del tipo `RBT t a`, el cual requiere para su definición de los tipos `AddBLayer t a` y `PRed t a`. De manera que es necesario definir una función de pertenencia para cada uno de estos tipos. Así, la función *member* resulta ser un caso particular de la función *rbmember*, la cual decide la pertenencia para árboles del tipo `RBT t a`. Puesto que este tipo depende de un constructor de árboles arbitrario `t`, la función *rbmember* requiere de un parámetro adicional `m` del tipo `t a -> Bool`, el cual es una función de pertenencia para árboles del tipo `t a`, que en el caso en que `t` es `EmptyT` y ya que `RBTree a` se define como `RBT EmptyT a`, la función member simplemente llama a *rbmember* utilizando la función de pertenencia para `EmptyT` corresponde a la función constante `False`.

Revisemos una a una estas funciones de pertenencia.

La definición de *rbmember* es:

```
rbmember :: Ord a => a -> RBT t a -> (t a->Bool) -> Bool

rbmember x (Zero t) m = m t
rbmember x (Suc u) m = rbmember x u (ablmem x m)
```

en el primer caso la pertenencia se decide desencapsulando el árbol `t` e invocando a la función de pertenencia `m`. En el segundo caso se hace una llamada recursiva[1] buscando el elemento en el árbol `u` pero utilizando la función de pertenencia auxiliar *ablmem*, cuya implementación es la siguiente:

```
ablmem :: Ord a => a -> (t a->Bool) -> AddBLayer t a -> Bool

ablmem x m (B(l,y,r))
 | x<y = prmem x m l
 | x>y = prmem x m r
 | otherwise = True
```

esta es la función de pertenencia para árboles del tipo `AddBLayer t a` y lo que hace es comparar el elemento buscado con la raíz del árbol dado, llamando a

[1] Se trata de una recursión polimórfica puesto que en la llamada recursiva el tipo de la función *rbmember* cambia.

la función de pertenencia en el subárbol izquierdo o derecho según sea el caso. Puesto que los subárboles de un árbol del tipo `AddBLayer t a` son de tipo `PRed t a`, la función *ablmem* no es recursiva sino que llama a la función de pertenencia *prmem* para árboles potencialmente rojos, cuya definición aparece a continuación tomando en cuenta las mismas ideas de las funciones anteriores.

```
prmem :: Ord a => a -> (t a->Bool) -> PRed t a->Bool

prmem x m (C t) = m t
prmem x m (R(l,y,r))
 | x<y = m l
 | x>y = m r
 | otherwise = True
```

Lo siguiente a revisar en la implementación es la función de balanceo a ser usada en las de inserción y remoción. Aquí tenemos que considerar la implementación de otros dos tipos, que son:

```
type RR t a = PRed (PRed t) a
type RL t a = PRed (AddBLayer t) a
```

estos tipos construyen un árbol potencialmente rojo a partir de árboles potencialmente rojos o negros respectivamente. Hay que observar que los árboles de tipo `RR t a` son árboles que presentan una violación a la invariante de nodos rojos.

La función *balance* dada en esta implementación se origina exactamente en la misma idea que maneja Okasaki en su implementación, con la diferencia de que como estamos hablando de familias infinitas de tipos se tienen que realizar unos cambios. La nueva función *balance* es la siguiente

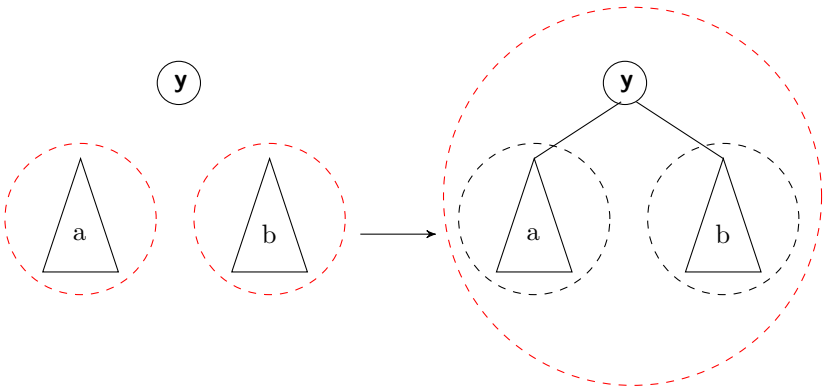
```
balance :: RR t a -> a -> RR t a -> RL t a

balance (R a) y (R b) = R(B a,y,B b)
balance (C a) x b = balanceR a x b
balance a x (C b) = balanceL a x b
```

esta función recibe dos árboles potencialmente rojos de tipo `RR t a`, y un elemento de tipo `a` y devuelve un árbol de tipo `RL t a`. Recordemos que los dos árboles de entrada son árboles con violación de invariante y en caso de que tengan constructor `R`, notemos que el árbol que devuelve es un árbol sin violación alguna. El primer caso

```
balance (R a) y (R b) = R(B a,y,B b)
```

es exactamente el primer caso de la función *balance* de Okasaki de la página 14. Se reciben dos árboles rojos y un elemento de entrada, y se devuelve un árbol rojo cuya raíz es el elemento de entrada y los subárboles son los árboles

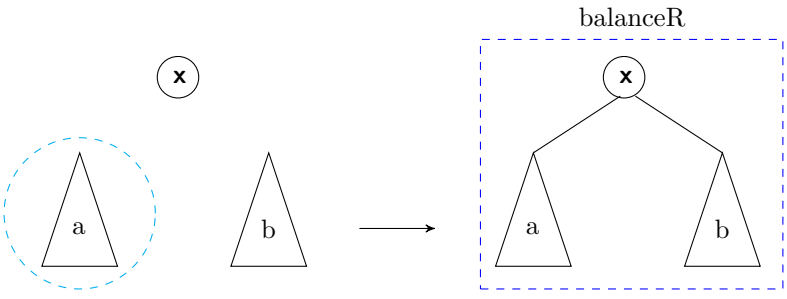
de entrada pero con su raíz ahora negra. Recordemos que las circunferencias punteadas denotan la aplicación de los constructores `R` o `B`, la definición anterior corresponde a la siguiente figura

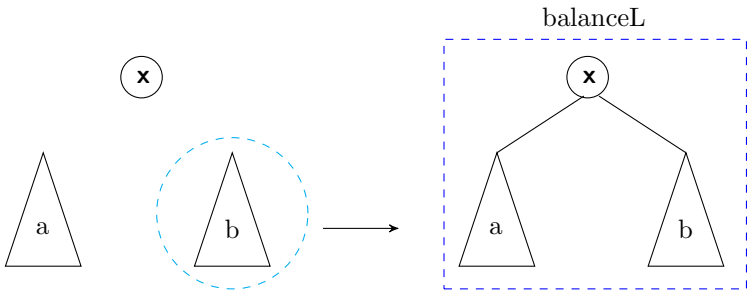


Los siguientes dos casos

```
balance (C a) x b = balanceR a x b
balance a x (C b) = balanceL a x b
```

son los correspondientes a cuando uno de los dos árboles de entrada es negro, pues recordemos que `C` encapsula árboles negros sólo para poder encajarlos en el tipo potencialmente rojo que necesitamos. Observemos que si el primer árbol de entrada es negro, entonces devolvemos el resultado de hacer *balanceR* a los argumentos pero sin poner el constructor `C` al primer árbol de entrada. Si el segundo árbol de entrada es negro, entonces devolvemos el resultado de hacer *balanceL* a los argumentos pero sin poner el constructor `C` al segundo árbol de entrada. Las funciones *balanceR* y *balanceL* corresponden a las definiciones que cubren los casos respecto a los árboles de entrada de *balance*, de los cuales no conocemos su color. Estos dos casos los podemos ver más claramente en las siguientes imágenes



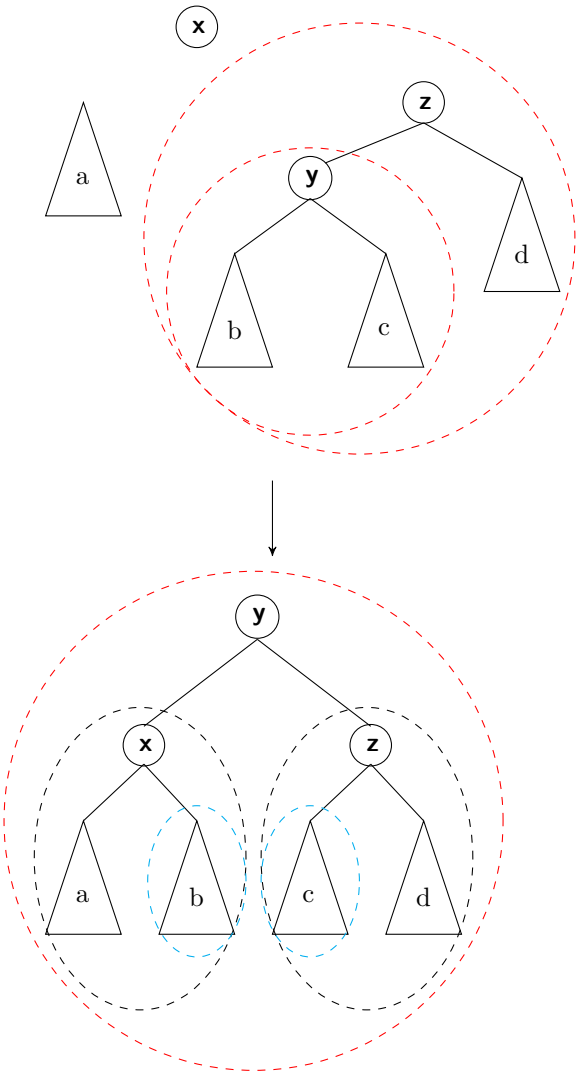


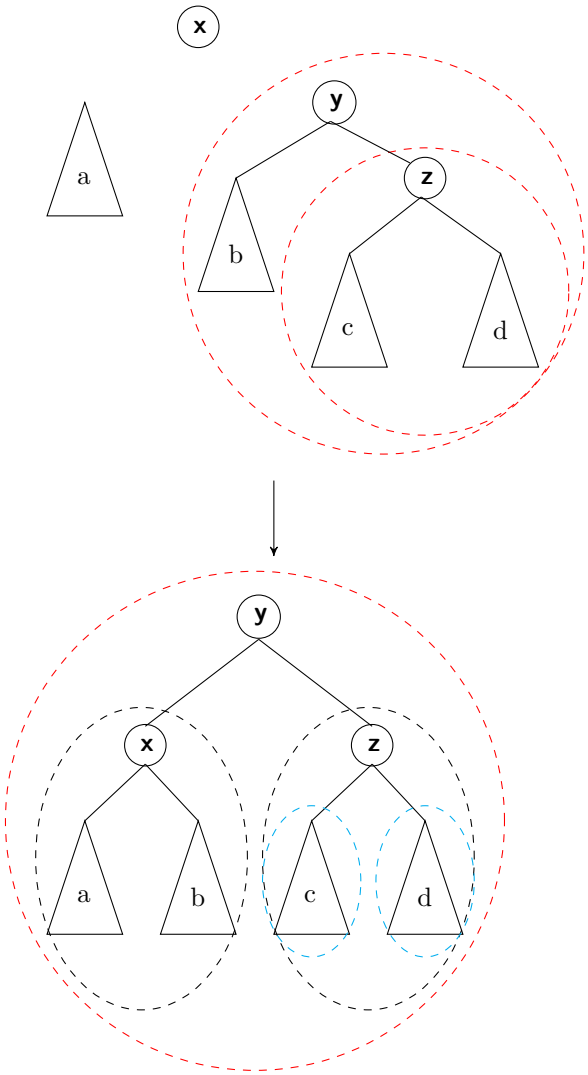
aquí, la circunferencia azul punteada, indica que el árbol está encapsulado con `C`.

Debido a que *balanceR* es análogo a *balanceL*, explicaremos sólo *balanceR*. La función *balanceR* se encarga de eliminar la violación de la invariante rojo en un árbol derecho. Esta función toma dos árboles potencialmente rojos, uno de tipo `PRed t a`, otro de tipo `RR t a` y un elemento de tipo `a`, de manera que construye un árbol de tipo `RL t a`, resultado de restaurar la invariante violado por el segundo árbol de entrada, la implementación es la siguiente

```
balanceR :: PRed t a -> a -> RR t a -> RL t a

balanceR a x (R(R(b,y,c),z,d)) = R(B(a,x,C b),y,B(C c,z,d))
balanceR a x (R(b,y,R(c,z,d))) = R(B(a,x,b),y,B(C c,z,C d))
balanceR a x (R(C b,y,C c)) = C(B(a,x,R(b,y,c)))
balanceR a x (C b) = C(B(a,x,b))
```

los dos primeros casos corresponden a los casos cuatro y cinco de la implementación inicial dada por Okasaki y de hecho la solución es exactamente la misma, con la diferencia de que para evitar errores de tipado, algunos árboles negros van encapsulados en `C`. Las siguientes figuras muestran dichos casos


x
z
y
a
b
c
d
y
x
z
a
b
c
d

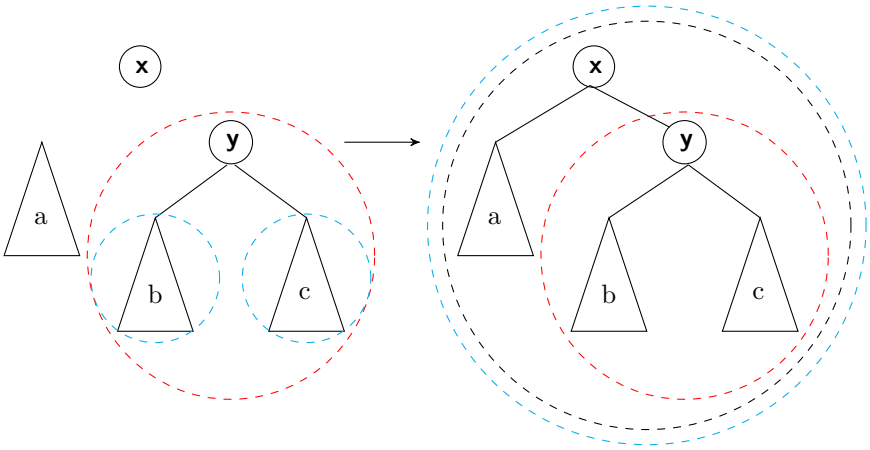


observemos que los dos casos anteriores son cuando tenemos dos nodos rojos adyacentes, es decir, cuando existe una violación de invariante.

El tercer caso es

```
balanceR a x (R(C b,y,C c)) = C(B(a,x,R(b,y,c)))
```

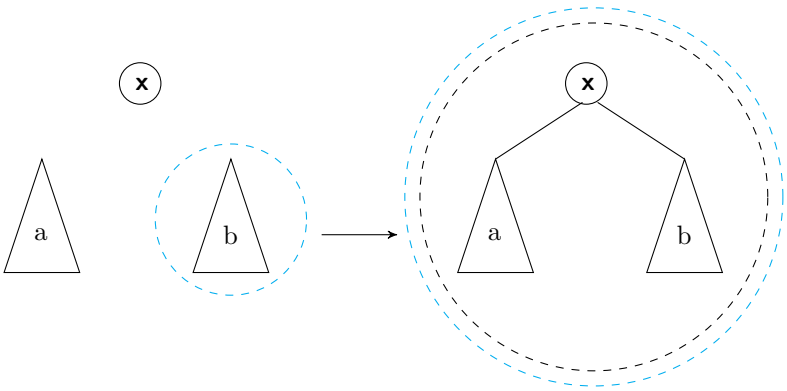
que corresponde al caso en que el segundo árbol de entrada es rojo con subárboles negros. Aquí la solución es armar un árbol negro cuya raíz es el elemento de entrada, como subárbol izquierdo tiene al primer árbol de entrada y como subárbol derecho un árbol rojo, con sus subárboles sin encapsular puesto que se sabe son negros. Lo anterior se puede observar en la siguiente figura



El cuarto caso es

```
balanceR a x (C b) = C(B(a,x,b))
```

y corresponde al último caso de la implementación de Okasaki, es cuando el segundo árbol de entrada es negro. De modo que se devuelve un árbol negro encapsulado, para evitar errores de tipado con el tipo `RL t a`, que se construye a partir de los árboles de entrada. Este proceso se visualiza de la siguiente manera:



Sigamos con la definición de la operación de inserción en árboles roji-negros con tipos anidados. Siguiendo la idea de la función *member* tendríamos que pasar un parámetro extra a la función de inserción en todos los casos, la cual es la función de inserción del elemento anterior en la familia de tipos parametrizados. Para simplificar esto se usará el mecanismo de clases de Haskell, de tal forma que los constructores de árbol `t` que tengan definida una función de inserción serán instancias de una misma clase.

Antes de continuar conviene recordar qué es una clase en Haskell. Tal y como un tipo es una colección de valores relacionados de alguna manera, una clase es

una colección de tipos o constructores de tipo que soportan ciertas operaciones que podemos llamar métodos, aunque no tienen las características de los métodos en la programación orientada a objetos, si no que más bien corresponden al concepto de sobrecarga de operadores. HASKELL proporciona algunas clases básicas como son `Eq a` que colecta aquellos tipos que tienen definida una función de igualdad; `Ord a` cuyos elementos son tipos ordenables mediante una función `<`; `Num a` que incluye a todos los tipos númericos; y `Show a` cuyos elementos son todos aquellos tipos que pueden convertirse en cadenas y por lo tanto pueden verse en la pantalla.

Necesitamos ahora una clase que colecte a todos aquellos constructores de tipo (en realidad constructores de árboles) que tengan definida una función de inserción. Dicha clase se define como sigue:

```
class Insertion t where ins :: Ord a => a -> t a -> PRed t a
```

Como podemos observar, la clase `Insertion` cuenta con la función *ins*, la cual toma un elemento de tipo `a` y un árbol de tipo `t a`, y devuelve un árbol potencialmente rojo. Adicionalmente, se requiere que el tipo de elementos `a` sea un tipo de la clase `Ord`, es decir, un tipo ordenado. La idea es la misma que en la primera implementación de este trabajo, devolver un árbol rojo para al final pintar la raíz de negro.

A continuación se explicarán las instancias de la clase de inserción.

Primero tenemos la declaración de una instancia de la clase cuando el árbol sobre el que vamos a insertar es vacío, aquí la función *ins* simplemente devuelve una hoja roja.

```
instance Insertion EmptyT where ins x E = R(E,x,E)
```

Las siguientes dos declaraciones son una instancia de clase para cuando el árbol sobre el que vamos a insertar es distinto del vacío, y aquí tenemos dos casos, cuando el árbol sobre el que vamos a hacer la inserción es de tipo `AddBLayer t` y cuando es de tipo `PRed t`.

En el primer caso la definición es

```
instance Insertion t => Insertion (AddBLayer t) where
ins x t@(B(l,y,r))
    | x<y = balance(ins x l) y (C r)
    | x>y = balance(C l) y (ins x r)
    | otherwise = C t
```

la idea es la misma que en implementaciones anteriores, comparar [2] el nodo a insertar `x` con el nodo raíz `y` e insertar en el subárbol izquierdo o derecho

[2] Para esto es que se requiere que el tipo de elementos `a` pertenezca a la clase `Ord`

según sea el caso, construyendo el nuevo árbol de manera inteligente mediante la función *balance*. Lo importante a notar aquí es la llamada recursiva a la función *ins*. Estamos nuevamente ante un ejemplo de recursión polimórfica, dado que las llamadas a *ins* en la función *balance* son para el tipo `PRed t a`, por lo que resulta imprescindible instanciar este tipo a la clase de inserción. Esto se hace de la siguiente manera:

```
instance Insertion t => Insertion (PRed t) where
ins x (C t) = C(ins x t)
ins x t@(R(a,y,b))
    | x<y = R(ins x a,y,C b)
    | x>y = R(C a,y,ins x b)
    | otherwise = C t
```

aquí la función *ins* hace llamadas recursivas a la función *ins* para el constructor `t` por lo tanto la declaración de instancia debe garantizar que `t` tenga una función de inserción, lo cual se logra mediante la declaración `Insertion t`.

Lo que sigue es definir la función de inserción para árboles del tipo `RBTree a`. Ésta resulta ser un caso particular de la función *rbinsert* por lo que su definición es la siguiente

```
insert :: Ord a => a -> RBTree a -> RBTree a

insert = rbinsert

rbinsert :: (Ord a,Insertion t) => a -> RBT t a -> RBT t a

rbinsert x (Zero t) = blackenIns(ins x t)
rbinsert x (Suc t) = Suc (rbinsert x t)
```

La función *rbinsert* se comporta como sigue: si el árbol de entrada es de la forma `Zero t`, entonces simplemente hacemos la inserción en el árbol `t` y pintamos de negro la raíz del árbol resultado empleando la función *blackenIns* que explicaremos en un momento más. Si el árbol de entrada es de la forma `Suc t`, se hace una recursión polimórfica llamando a la misma función pero con tipo `RBT (AddBLayer t) a`, encapsulando el resultado de esta llamada nuevamente con el constructor `Suc`.

La función *blackenIns* se implementa en Haskell como

```
blackenIns :: PRed t a -> RBT t a

blackenIns (C u) = Zero u
blackenIns (R(a,x,b)) = Suc(Zero(B(C a,x,C b)))
```

es la encargada de pintar de negro la raíz de un árbol de tipo `PRed t a` resultado de la inserción. Esta función recibe un árbol potencialmente rojo de

entrada, y devuelve un árbol roji-negro de tipo `RBT t a`. Debido a que el árbol de entrada es potencialmente rojo, debemos revisar dos casos. Si el árbol que se recibe está encapsulado en `C`, entonces devolvemos el árbol `u` pero encapsulado con `Zero`, esto porque ya sabemos que `u` es negro y no hay nada que hacer. Si el árbol de entrada es rojo, entonces se devuelve un árbol negro cuyos subárboles son los mismos (pues sólo queremos pintar el árbol) pero encapsulados en `C` para respetar los tipos. Debido a que este árbol proviene de uno rojo, debemos encapsularlo en los constructores `Zero` y `Suc`, pues estamos añadiendo un nodo negro, y por consiguiente debemos incrementar la altura negra en uno.

Una vez que ya tenemos la inserción podemos empezar a revisar la implementación para la operación de borrado. De la misma forma que con *insert*, para la función *delete* principal se necesitarán algunas clases y varias funciones auxiliares. Empecemos por definir la clase `Append`, cuyas instancias son constructores de árbol que tienen definida una función `app`, la cual se encarga de unir dos árboles construidos por `t`, devolviendo un árbol potencialmente rojo. La implementación es la siguiente

```
class Append t where app :: t a -> t a -> PRed t a
```

A continuación definimos las instancias de esta clase necesarias para definir la operación de borrado. Para el caso del constructor de árboles vacíos `EmptyT` la definición es

```
instance Append EmptyT where app _ _ = C E
```

En este caso devolvemos `C E` puesto que el resultado de pegar dos árboles vacíos es nuevamente un árbol vacío, el cual se inyecta en el tipo de árboles potencialmente rojos mediante el constructor `C`.

La segunda instancia es la encargada de pegar dos árboles potencialmente rojos.

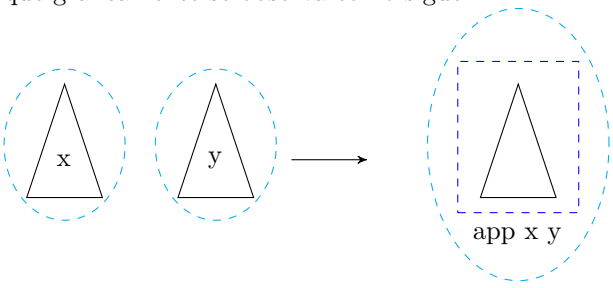
```
instance Append t => Append (PRed t) where
    app (C x) (C y) = C(app x y)
    app (C t) (R(a,x,b)) = R(app t a,x,C b)
    app (R(a,x,b)) (C t) = R(C a,x,app b t)
    app (R(a,x,b))(R(c,y,d)) = threeformR a x (app b c) y d
```

esta instancia es una variación nuestra del código original, hecha con el propósito de obtener una presentación más uniforme y elegante del código. Analizamos ahora los diferentes casos de la definición.

El primer caso es

```
app (C x) (C y) = C(app x y)
```

que gráficamente se observa como sigue

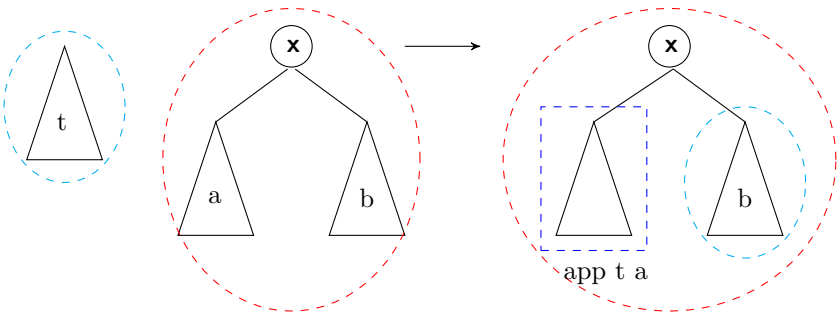


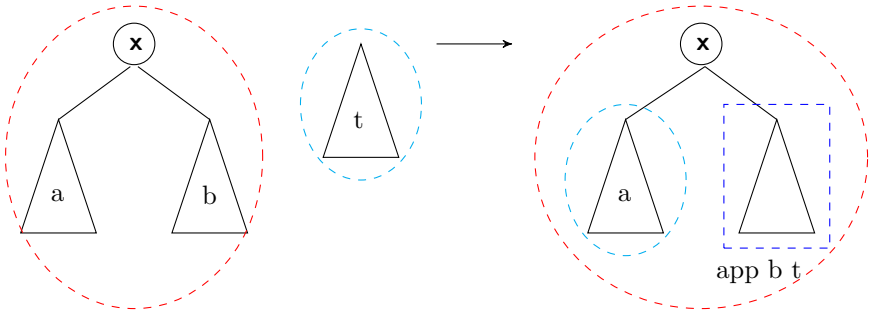
Este caso corresponde a cuando los dos árboles potencialmente rojos que vamos a unir están encapsulados con `C`, es decir, son negros, en tal situación se devuelve el árbol resultante de pegar los árboles `x` y `y`, encapsulado nuevamente el resultado en `C`. Aquí la llamada a la función `app` del lado derecho no es una llamada recursiva, puesto que corresponde a la función `app` en el constructor de árbol `t` (debido a esto es que en la definición de la instancia se requiere que `t` también pertenezca a la clase `Append`).

Los siguientes dos casos son

```
app (C t) (R(a,x,b)) = R(app t a,x,C b)
app (R(a,x,b)) (C t) = R(C a,x,app b t)
```

y corresponden a cuando queremos unir un árbol rojo con uno negro (encapsuládo con `C`). Aquí devolvemos un árbol rojo, donde el subárbol izquierdo o derecho se obtiene al pegar el árbol negro con el subárbol izquierdo o derecho del árbol rojo de entrada. Estos casos están representados por las siguiente figuras.

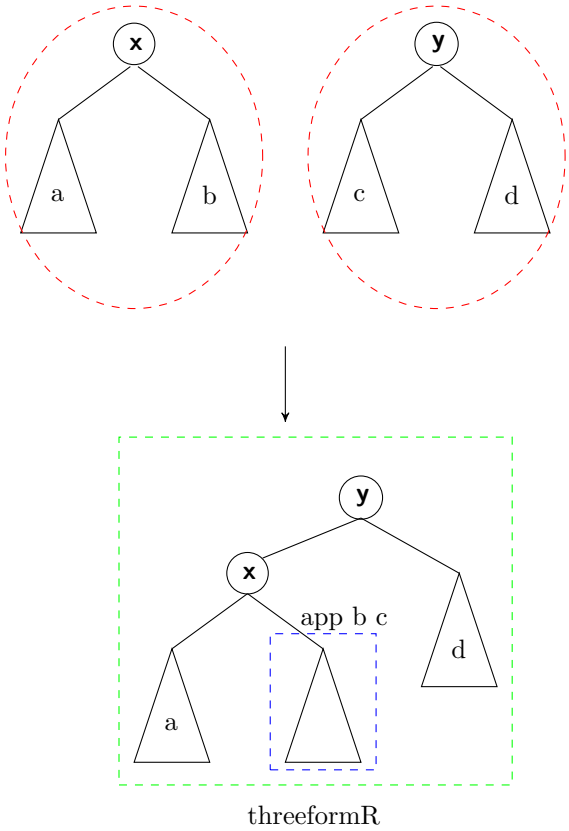




El último caso es

```
app (R(a,x,b))(R(c,y,d)) = threeformR a x (app b c) y d
```

y corresponde a cuando se desea unir dos árboles rojos. Aquí se devuelve un árbol de tipo `RR t a` (el cual es un caso particular de `PRed`) resultado de llamar a la función $threeformR$, que se explicará más adelante y que se encarga de construir un nuevo árbol rojo a partir de tres árboles y dos elementos. La representación gráfica es la siguiente

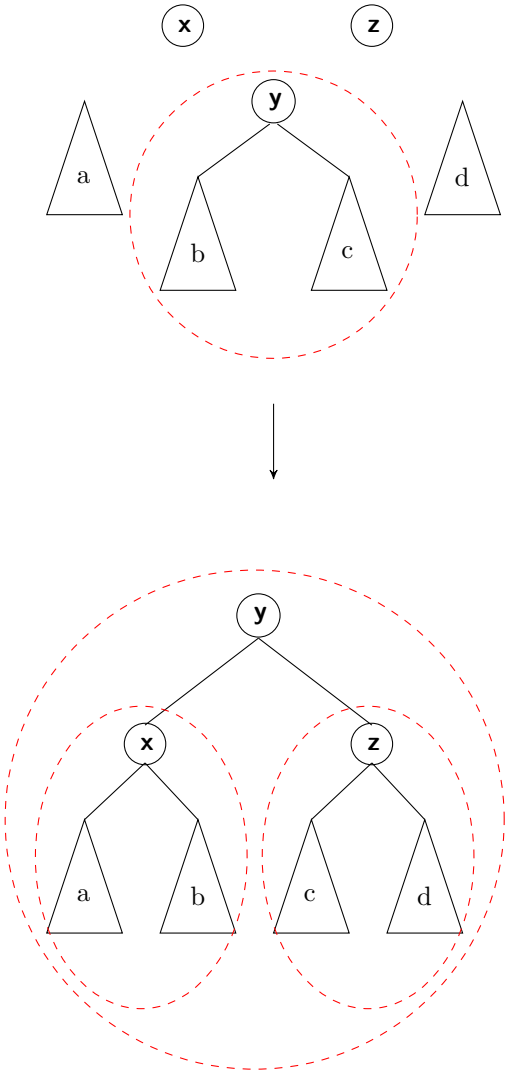


La función *threeformR* tiene la siguiente implementación

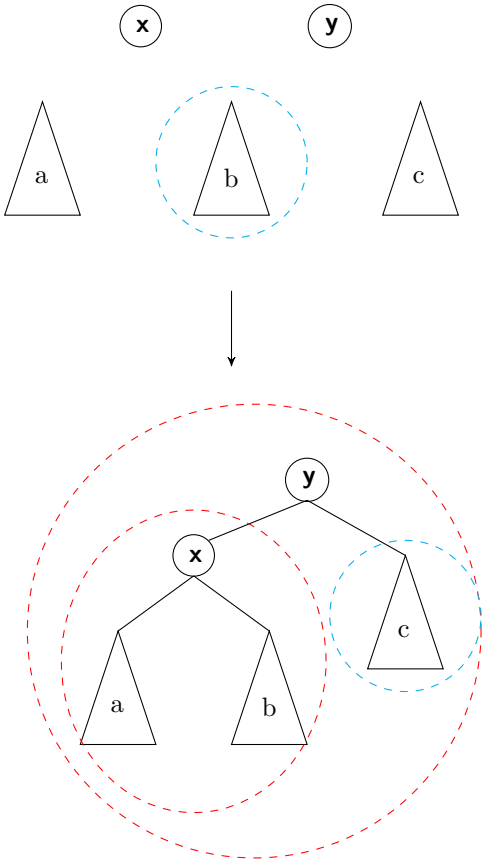
```
threeformR:: t a -> a -> PRed t a -> a -> t a -> RR t a

threeformR a x (R(b,y,c)) z d = R(R(a,x,b),y,R(c,z,d))
threeformR a x (C b) y c = R(R(a,x,b),y,C c)
```

tiene como objetivo construir un árbol de tipo `RR t a` a partir de dos elementos y tres árboles, dos de tipo `t a` y uno de tipo `PRed t a`. La representación gráfica es:

Primer caso



Segundo caso

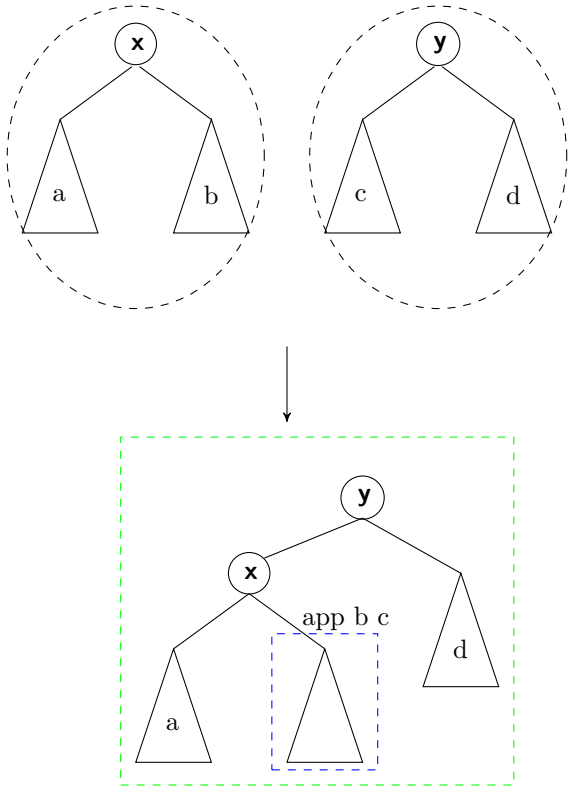


La definición muestra una clara violación a la invariante rojo-rojo, que se solucionará más adelante.

La última instancia de la clase que necesitamos es para el constructor `AddBLayer t`. Su definición es:

```
instance Append t => Append (AddBLayer t) where
    app (B(a,x,b)) (B(c,y,d)) = threeformB a x (app b c) y d
```

Aquí la función `app` pega dos árboles negros devolviendo un árbol potencialmente rojo, resultado de llamar a la función $threeformB$, función análoga a $threeformR$. La representación gráfica es



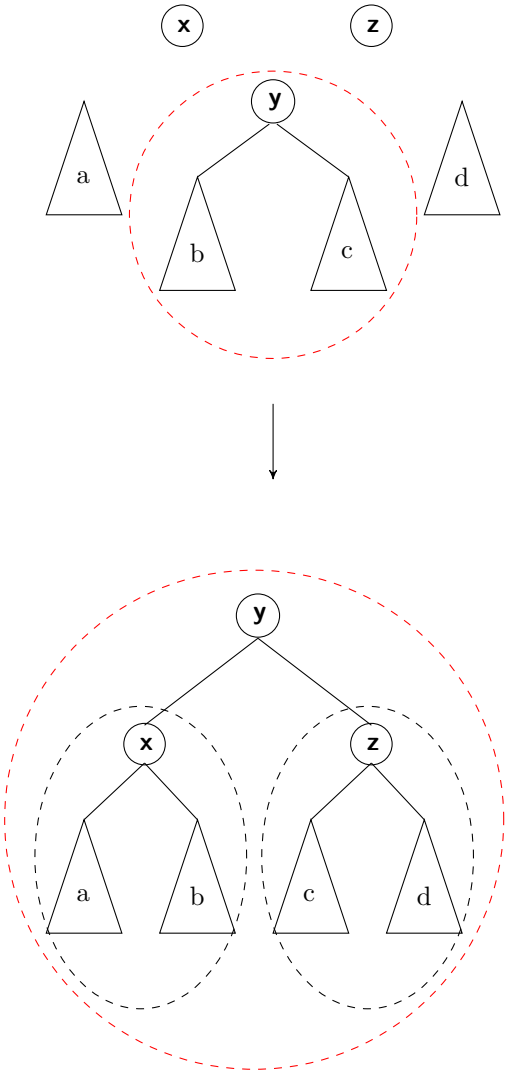
threeformB

La función *threeformB*, que en HASKELL se observa como sigue

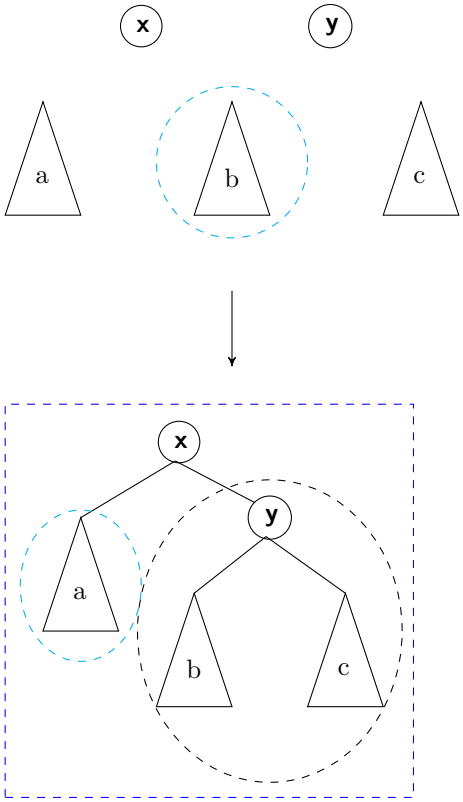
```
threeformB :: PRed t a -> a -> RR t a -> a -> PRed t a -> RL t a

threeformB a x (R(b,y,c)) z d = R(B(a,x,b),y,B(c,z,d))
threeformB a x (C b) y c = balleftB (C a) x (B(b,y,c))
```

es análoga en cuanto a objetivo a la función *threeformR*, la diferencia aquí es que se reciben de entrada dos elementos de tipo `a` y tres árboles potencialmente rojos, uno de los cuales pudiera tener una violación explícita a la invariante (el de tipo `RR t a`). Devolvemos como resultado un árbol potencialmente rojo con subárboles negros de tipo `AddBLayer t a`, es decir, un árbol de tipo `RL t a`. Observemos que en el segundo caso se llama a una función auxiliar de balanceo, la cual, explicaremos más adelante, se manda llamar a esta función porque al unir dos árboles negros se desbalancean las alturas negras, entonces se debe se balancear. Las siguientes figuras muestran los casos de esta función.

El primer caso es



y el segundo caso es

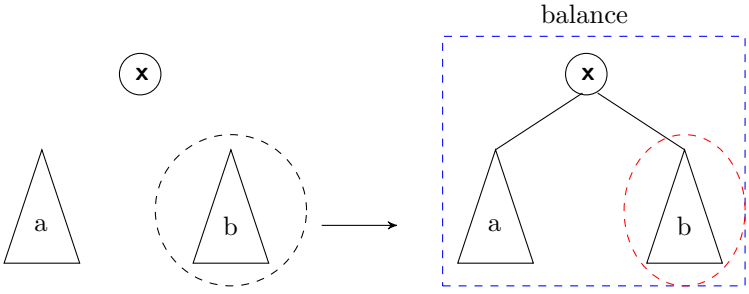


balleftB

La función *balleftB*, que en HASKELL se observa como sigue

```
balleftB :: RR t a -> a -> AddBLayer t a -> RL t a

balleftB bl x (B y) = balance bl x (R y)
```

se encarga de hacer un balanceo común de los árboles que recibe de entrada, pero cambiándole el color de negro a rojo al segundo árbol de entrada, tal y como se muestra en la siguiente figura



$balleftB$ es una función requerida por $balleft$, que es el constructor inteligente encargado de hacer el balanceo manteniendo de manera válida el árbol de entrada izquierdo, y cuya implementación es la que se muestra a continuación

```
balleft :: RR t a -> a -> RL t a -> RR (AddBLayer t) a

balleft (R a) y c = R(C(B a),y,c)
balleft (C t) x (R(B(a,y,b),z,c)) =
   R(C(B(t,x,a)),y,balleftB (C b) z c)
balleft b x (C t) = C (balleftB b x t)
```

observemos que $balleft$ lleva exactamente la misma lógica que la función $balleft$ dada en la sección 3 (página 18). Un aspecto importante que hay que notar es que la función $balleftB$ juega el papel de la función red de aquella implementación.

Lo que sigue es implementar la operación de borrado, para eso necesitaremos definir tres clases más:

- La clase `Delred t` que dependerá de la clase `Append t` y que implementa tres funciones ($delTup$, $delLeft$, $delRight$), las cuales explicaré un poco más adelante.

  ```
  class Append t => Delred t where
  delTup :: Ord a => a -> Tr t a -> PRed t a
  delLeft :: Ord a => a -> t a -> a -> PRed t a -> RR t a
  delRight :: Ord a => a -> PRed t a -> a -> t a -> RR t a
  ```

- La clase `Del t`, que incluye la función del

  ```
  class Append t => Del t where
  del :: Ord a => a -> AddBLayer t a -> RR t a
  ```

- La clase `Deletion t`, la cual no tiene funciones propias y que simplemente corresponde a la unión de las clases `Delred t` y `Del t`

  ```
  class (Delred t, Del t) => Deletion t
  ```

Observemos que todas las funciones en estas clases devuelven un árbol potencialmente rojo y en algunos casos éste puede tener subárboles potencialmente rojos, violando la invariante de nodos rojos. Esto no es problema ya que al final de la eliminación se pintará la raíz de negro como en las otras implementaciones. Comencemos por implementar las instancias de las clases que acabamos de definir.

La primera instancia que se define es la de `Delred EmptyT` donde se implementan las eliminaciones triviales.

```
instance Delred EmptyT where
delTup z t@(E,x,E) = if x==z then C E else R t
delLeft x E y b = R(C E,y,b)
delRight x a y E = R(a,y,C E)
```

La función *delTup* elimina la raíz de una hoja, esta función toma un elemento de tipo `a` y una hoja de tipo `Tr Empty a`, y lo que hace es verificar si el elemento de entrada es igual a la raíz de la hoja, en cuyo caso se elimina y devolvemos el árbol vacío encapsulado en `C`, pues por convención el vacío es negro; si el elemento de entrada es distinto a la raíz de la hoja, entonces dejamos la hoja tal cual pero la pintamos de rojo.

Las funciones *delLeft* y *delRight* reciben un elemento `x`, y un árbol (desarmado) y construyen el árbol resultante de borrar el elemento dado del subárbol correspondiente. En este caso como el subárbol izquierdo o derecho es vacío (pues el constructor de árboles es `EmptyT`), no se realiza ninguna operación de borrado y el árbol devuelto se construye armando el árbol de entrada de manera que se obtenga un árbol rojo.

La instancia donde se implementan las eliminaciones no triviales es la siguiente;

```
instance Deletion t => Delred (AddBLayer t) where
delTup z (a,x,b)
| z<x = balleftB (del z a) x b
| z>x = balrightB a x (del z b)
| otherwise = app a b
delLeft x a y b = balleft (del x a) y b
delRight x a y b = balright a y (del x b)
```

la función *delTup* ahora recibe el elemento `z` que se desea eliminar y una terna (es decir un árbol desarmado) donde los árboles `a` y `b` son construidos por `t`. Si el elemento a eliminar es menor que el elemento raíz, entonces la eliminación se hace en el subárbol izquierdo y posteriormente se balancea mediante la función $balleftB$. El caso en que el elemento a eliminar es menor que el nodo raíz se resuelve análogamente y si son iguales entonces se llama a la función *app* de la clase `Append` para que una a los subárboles `a` y `b`.

La siguiente instancia que se define es la de `Del t` para `t` un constructor de árbol arbitrario. En esta instancia se implementa la operación de borrado pero sobre un árbol negro.

```
instance Delred t => Del t where
del z (B(a,x,b))
    | z<x = delfromLeft a
    | z>x = delfromRight b
    | otherwise = app a b
              where delfromLeft(C t) = delLeft z t x b
```

```
delfromLeft(R t) = R(delTup z t,x,b)
delfromRight(C t) = delRight z a x t
delfromRight(R t) = R(a,x,delTup z t)
```

Si el elemento a eliminar es menor que la raíz, entonces se manda llamar a la función $delfromLeft$ que es el constructor inteligente encargado de hacer la eliminación sobre el subárbol izquierdo. Si el elemento a eliminar es mayor que la raíz, entonces se manda llamar a la función $delfromRight$ que es el constructor inteligente encargado de hacer la eliminación pero sobre el subárbol derecho. Si el elemento a eliminar y la raíz son iguales, entonces llama a la función *app* de la clase `Append t` para unir ambos subárboles.

Como el borrado de un nodo rojo se hace mediante las funciones $delfromLeft$ y $delfromRight$ entonces ya no es necesaria definir la instancia `Delred (PRed t)`.

Las funciones $delfromLeft$ y $delfromRight$ son análogas, por lo que, nuevamente, sólo revisaremos la primera. Esta función hace inteligentemente la eliminación sobre el subárbol izquierdo, como dicho subárbol debe ser potencialmente rojo, entonces $delfromLeft$ analiza dos casos: cuando el árbol de entrada esta encapsulado en `C` y cuando es rojo. En el primer caso se hace la eliminación mediante la función $delLeft$; en el segundo caso se devuelve un árbol rojo, haciendo la eliminación sobre el subárbol izquierdo con ayuda de la función $delTup$.

Como ya se dijo anteriormente la clase `Deletion t` depende de la unión de `Delred t` y `Del t` y como la instancia de `Del t` depende de `Delred t` entonces ya tenemos las instancias de esta clase, por lo que las siguientes instancias son automáticas.

```
instance Deletion EmptyT
instance Deletion t => Deletion (AddBLayer t)
```

El siguiente paso es definir la función *rbdelete*, cuya signatura es la siguiente:

```
rbdelete::(Ord a,Deletion t) => a -> RBT (AddBLayer t) a -> RBT t a
```

esta función toma un elemento x y un árbol t del tipo `RBT (AddBLayer t) a`, y devuelve el árbol resultado de borrar x en t, transformando al mismo tiempo el resultado a un árbol de tipo `RBT t a`. La implementación es:

```
rbdelete x (Zero t) = blackenDel (del x t)
rbdelete x (Suc t) = Suc (rbdelete x t)
```

Si el árbol de entrada está encapsulado en el constructor `Zero`, entonces simplemente hacemos el borrado sobre el árbol `t` y pintamos de negro la raíz del árbol

resultado empleando la función *blackenDel* que explicaremos en un momento más. Si el árbol de entrada está encapsulado en el constructor `Suc`, hacemos la recursión polimorfíca sobre `t` de manera que el resultado se encapsula nuevamente con el constructor `Suc`.

La función *blackenDel* tiene como objetivo pintar de negro la raíz de un árbol potencialmente rojo de tipo `RR t a`, devolviendo un árbol de tipo `RBT t a`. Debido a que el árbol de entrada es de tipo `RR t a` debemos revisar varios casos. Si el árbol que se recibe está encapsulado en `C` y además su parámetro es un árbol `t` encapsulado en `C`, entonces devolvemos el árbol `t` pero encapsulado con `Zero`, esto porque ya sabemos que `t` es negro asıque no hay nada que hacer. El contructor `Zero` indica que la altura negra permanecio igual. Si el árbol de entrada es rojo pero esta encapsulado en `C`, entonces se devuelve un árbol negro cuyos subárboles son los mismos pero encapsulados en `C`. Debido a que este árbol proviene de uno rojo, debemos encapsularlo en los constructores `Zero` y `Suc` pues estamos añadiendo un nivel negro y por consiguiente estamos incrementando la altura negra en uno. Si el árbol de entrada es rojo pintamos de negro su raíz (es decir, cambiamos el constructor `R` por `B`) y lo encapsulamos en los constructores `Zero` y `Suc`, pues al igual que el patrón anterior, se esta agregando un nivel negro por que la altura negra aumenta en uno. La implementación de *blackenDel* es la que se observa enseguida.

```
blackenDel :: RR t a -> RBT t a

blackenDel (C(C t)) = Zero t
blackenDel (C(R(a,x,b))) = Suc(Zero(B(C a,x,C b)))
blackenDel (R p) = Suc(Zero(B p))
```

Finalmente, definimos la función de borrado para árboles roji-negros, cuya implementación es la siguiente

```
delete:: Ord a => a -> RBTree a -> RBTree a

delete x (Zero E) = Zero E
delete x (Suc u) = rbdelete x u
```

Así, *delete* recibe el elemento a eliminar, el cual debe de ser tipo `a`, y el árbol sobre el que se va a hacer la eliminación, el cual debe ser de tipo `RBTree a`. Esta función devuelve un árbol roji-negro de tipo `RBTree a`. La función *delete* es sencilla, el primer caso se explica solo. Si el árbol del que queremos hacer la eliminación está encapsulado en `Suc`, lo único que hacemos es llamar a la función general de borrado *rbdelete* en el árbol sin encapsular. Observemos que en este caso *delete* no es simplemente una instancia particular de *rbdelete*, puesto que esta última función modifica el tipo del árbol de entrada.

Con esto finalizamos de codificar la función de borrado para nuestra implementacion de árboles roji-negros con tipos anidados.

En este capítulo hemos desarrollado tres implementaciones ordinarias para la función de borrado para árboles roji-negros. Adicionalmente la última implementación con tipos anidados define tanto membresía como inserción y borrado. Por lo que ahora contamos con cuatro opciones para implementar conjuntos finitos. Esto se hará mediante el mecanismo de clases de Haskell en el siguiente capítulo.

Capítulo 4

Estudio de un caso: Conjuntos finitos

Matemáticamente un conjunto es una colección bien definida de elementos, no necesariamente ordenada y sin elementos repetidos, en computación, por lo general, a esta definición se le agrega la propiedad de que los elementos deben ser del mismo tipo.

Los conjuntos finitos pueden ser implementados con diversas estructuras de datos. Las listas son un claro ejemplo, pero implementar conjuntos con listas es muy ineficiente debido, entre otras cosas, a que la operación de borrado requiere destruir la lista dada hasta encontrar el elemento que se desea borrar y entonces reconstruir la lista nuevamente. Para resolver este problema se deben utilizar estructuras más eficientes, como nuestros árboles roji-negros.

Para utilizar árboles roji-negros como conjuntos, podemos seguir la siguiente correspondencia:

- `E` corresponde al conjunto $\emptyset$.
- `T Color l a r` corresponde a conjuntos con cardinalidad mayor o igual a 1, de la forma $c_1 \uplus \{a\} \uplus c_2$ donde la operación $\uplus$ denota a la unión de conjuntos ajenos, y los conjuntos c_1 y c_2 corresponden a los subárboles `l` y `r`, que pueden ser vacíos.

Implementamos conjuntos finitos mediante la siguiente clase de Haskell:

```
class Set s where
empty :: Ord a => s a
single :: Ord a => a -> s a
size :: s a -> Int
```

```
member :: Ord a => a -> s a -> Bool
insert1 :: Ord a => a -> s a -> s a
delete :: Ord a => a -> s a -> s a
fromList :: Ord a => [a] -> s a
toList :: s a -> [a]
```

donde

- `empty` construye al conjunto vacío.
- `single` construye un conjunto unitario.
- `size` calcula la cardinalidad del conjunto.
- `member` es la función de pertenencia.
- `insert` inserta un elemento al conjunto.
- `delete` borra un elemento de un conjunto.
- `fromList` transforma una lista en un conjunto.
- `toList` transforma un conjunto en una lista.

Se observa que en la definición de la clase `Set` el parámetro `s` es un constructor de tipo (un constructor de conjuntos), y que para poder utilizar el tipo de conjuntos de elementos de `a`, es decir el tipo `s a`, es necesario que el tipo `a` pertenezca a la clase de tipo `Ord`, es decir, que tenga un orden total.

A continuación definimos las instancias particulares de la clase `Set` utilizando nuestras implementaciones de árboles roji-negros. Para las primeras tres implementaciones la definición es directa puesto que ya contamos con las operaciones de pertenencia, inserción y borrado. Más aún, las funciones *empty*, *single*, *size*, $fromList$ y *toList* se definen de la misma manera. Como ejemplo mostramos la instancia para el caso de la aritmética de colores (`ARNac`).

```
instance Set RB where
empty = E

single x = T B E x E

size E = 0
size (T _ E x E) = 1
size (T _ tl x tr) = 1 + size tl + size tr

member = ARNac.member

insert1 = ARNac.insert

delete = ARNac.deleteRB
```

```
fromList = foldr ARNac.insert E

toList E = []
toList (T c tl x tr) = toList tl ++ [x] ++ toList tr
```

La función $fromList$ utiliza el operador nativo de recursión para listas $foldr$ siendo la idea de su definición la inserción, uno a uno, de los elementos de la lista dada en un árbol que originalmente es vacío.

Para el caso de la implementación con tipos anidados, como es de esperarse, la instanciación a la clase `Set` es más complicada. En particular para la definición de las funciones *size* y *toList* requerimos nuevamente del mecanismo de clases de HASKELL. Para el caso de la función *size* definimos la clase `Sizable`

```
class Sizable t where
   size :: t a -> Int
```

esta clase colecta a todos los constructores de árbol que tienen definida la función de tamaño *size*.

Las instancias necesarias son:

```
instance Sizable EmptyT where
   size E = 0

instance Sizable t => Sizable (PRed t) where
   size (C x) = size x
   size (R (l,x,r)) = size l + size r + 1

instance Sizable t => Sizable (AddBLayer t) where
   size (B (l,x,r)) = size l + size r + 1

instance Sizable t => Sizable (RBT t) where
  size (Zero b) = size b
  size (Suc b) = size b
```

De manera análoga, la definición de la función *toList* se sirve de la siguiente clase:

```
class Flatten t where
   flat :: t a -> [a]
```

aquí estamos colectando todos aquellos constructores de árbol que tienen defnida una función de aplanamiento $flat$, la cual devuelve una lista con todos los elementos de un árbol dado.

Las instancias necesarias son:

```
instance Flatten EmptyT where
   flat E = []

instance Flatten t => Flatten (PRed t) where
   flat (C x) = flat x
   flat (R (l,x,r)) = flat l ++ [x] ++ flat r

instance Flatten t => Flatten (AddBLayer t) where
   flat (B (l,x,r)) = flat l ++ [x] ++ flat r

instance Flatten t => Flatten (RBT t) where
  flat (Zero b) = flat b
  flat (Suc b) = flat b
```

Finalmente, la instancia de la clase `Set` mediante nuestra implementación de árboles roji-negros con tipos anidados (`ARNta`) es:

```
instance Set (RBT EmptyT) where

  empty = Zero E

  single x = Suc(Zero(B(C(E), x, C(E))))

  ssize = size

  member = ARNta.member

  insert = ARNta.insert

  delete = ARNta.delete

  fromList = foldr ARNta.insert (Zero E)

  toList = flat
```

Conclusiones

A lo largo de este trabajo se desarrollaron diversas implementaciones de árboles roji-negros en el ámbito puramente funcional. Se estudiaron las operaciones de balanceo e inserción dadas por Okasaki [13, 14], y consideramos que nuestra aportación principal consiste en el análisis y explicación de la operación de borrado en árboles roji-negros, para lo cual nos basamos en las implementaciones de Kahrs [11], Hinze [9] (constructores inteligentes) y Matthew [12] (aritmética de colores). Adicionalmente, presentamos una implementación, basada en [11], que utiliza conceptos avanzados de la programación funcional, a saber tipos anidados, y que captura las invariantes de árboles roji-negros de manera estática, es decir, mediante el tipado. Finalmente mostramos una aplicación al utilizar los árboles roji-negros para implementar conjuntos finitos.

Un tema importante que no se discutió en este trabajo es la construcción de árboles roji-negros. Por supuesto que una manera de construirlos es mediante la inserción continua de elementos en el árbol vacío, la cual corresponde a la función $fromList$. Sin embargo, existen otras maneras de construir árboles roji-negros desarrolladas en base a ciertas restricciones, por ejemplo, pedir árboles de altura mínima o máxima, o árboles con una proporción mínima o máxima de nodos rojos. A este respecto es de importancia relacionar las implementaciones de este trabajo con las ideas desarrolladas en el artículo [8].

Otro tópico de interés para trabajos futuros consiste en la verificación formal de nuestras implementaciones de árboles roji-negros, utilizando herramientas de vanguardia como lo es el asistente de pruebas Coq. En esta dirección podemos citar el trabajo [2] (véase también la dirección `https://coq.inria.fr/library/Coq.MSets.MSetRBT.html`).

Finalmente, queremos mencionar que la implementación de la operación de borrado mediante aritmética de colores, originalmente presentada en [12], se ha publicado recientemente en el artículo [6]. Este artículo incluye una implementación en Haskell, por lo que sería adecuado comparar dicha implementación con la nuestra, que fue desarrollada de manera independiente.

Conceptos básicos de Haskell

Para una mejor comprensión del código desarrollado en este trabajo, explicaremos a continuación brevemente algunos de los mecanismos del lenguaje funcional Haskell, utilizados en el mismo.

Cuando definimos una función en Haskell y queremos hacer explícito su tipo, definimos lo que se conoce como signatura de la función, por ejemplo si deseamos definir una función que sume dos números enteros y devuelva un número entero, damos la siguiente signatura:
`suma :: Int -> Int -> Int`
donde los primeros dos `Int` son el tipo de los argumentos de la función y el último `Int` es el tipo del valor devuelto por la función. Otro ejemplo puede ser la signatura para definir una función que niegue un valor de verdad, dicha signatura se ve como sigue:
`not :: Bool -> Bool`
donde el primer `Bool` es el tipo del argumento y el segundo `Bool` es el tipo del valor devuelto por `not`. Es posible también definir signaturas más generales, tomando el ejemplo de la función `suma`, si queremos que esa función no solo sume números enteros, si no que sea capaz de operar con cualquier tipo de número podemos escribir la signatura como:
`suma :: a -> a -> a`
donde la `a` es cualquier tipo númerico (`Int`, `Float`, `Double`, `Integer`). Este tipo de funciones se conocen como funciones polimórficas, mientras que el primer ejemplo de `suma` y la función `not` se conocen como funciones monomórficas. Finalmente observemos que habrá ocasiones en que necesitemos que los argumentos de nuestras funciones sean elementos de alguna clase específica, en este caso la signatura se escribe como sigue:
`suma :: Num a => a -> a -> a`.
Aquí la expresión `Num a =>` significa que la función `suma` está restringida a tipos que pertenecen a la clase `Num`.

Una clase en Haskell es un conjunto de tipos para los que tiene sentido defi-

nir una serie de operaciones sobrecargadas y que los elementos de dichas clases puedan usar. Las clases relevantes en este trabajo son `Eq`, `Ord`, `Show`, `Num`, donde `Eq` es la clase de los tipos que definen igualdad, `Ord` es la clase los tipos que tienen un orden definido, `Show` es la clase de los tipos que pueden mostrarse en consola y `Num` es la clase de los tipos númericos.

Usamos la instrucción `data` cuando deseamos definir un nuevo tipo de datos. Por ejemplo si quisieramos definir el tipo `Bool` hacemos
`data Bool = False | True`
donde la palabra `Bool` después de la palabra `data` es el nombre del nuevo tipo de dato y las palabras `False` y `True` son los constructores de valores del nuevo tipo, en este caso los dos valores booleanos. Es posible definir tipos de datos más complicados, como son tipos de datos definidos recursivamente y que dependan de uno o más parámetros. Por ejemplo, la definición usual del tipo de datos lista es:
`data List a = Nil | Cons a (List a)`
donde de igual forma que en el ejemplo anterior, `Nil` y `Cons` son los constructores de listas y la `a` es un parámetro de tipo, que corresponde al tipo de los elementos en la lista.

Un tipo de datos de gran utilidad en este trabajo es el siguiente:
`data Maybe a = Just a | Nothing`
esta definición corresponde al tipo `a`, pero etiquetando sus valores mediante `Just`. Además, incluye un valor extra, denotado `Nothing`, el cual representa una excepción.

Cuando definimos tipos nuevos podemos incluirlos automáticamente en ciertas clases mediante el uso de la instrucción `deriving`. Por ejemplo, para poder visualizar elementos del tipo lista, es necesario que éstos pertenezcan a la clase `Show`, lo cual se logra mediante la siguiente instrucción:
`data List a = Nil | Cons a (List a) deriving Show`.

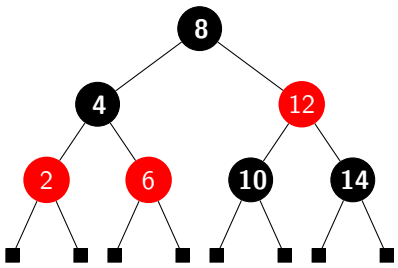
En este trabajo, se utiliza en la definición de algunas funciones el símbolo `@`, el cual sirve para declarar una etiqueta que denote a un argumento en particular. Por ejemplo, en la siguiente función:

```
duplica :: [a] -> [a]
duplica [] = []
duplica l@(x:xs) = l ++ l
```

la etiqueta `l` denota al argumento `(x:xs)`.

Acerca de la corrección

En este trabajo hemos presentado diversas implementaciones de árboles roji-negros, en especial de la operación de borrado, siendo necesaria una discusión acerca de la corrección de nuestras definiciones e implementaciones. A este respecto queremos mencionar que, dado que estamos utilizando el paradigma funcional puro, es posible verificar la corrección con técnicas matemáticas, por lo general inducción estructural. Si bien en los artículos utilizados para la realización de este trabajo se discute brevemente la corrección de las definiciones, la verificación de la misma a detalle, ya sea mediante un razonamiento matemático o usando herramientas de verificación formal, como es el caso presentado en [2], aún cuando consideramos es de gran importancia e interés, cae fuera de nuestro alcance y propósitos. En su lugar presentamos evidencia de la corrección de nuestras definiciones como es usual en programación, mediante pruebas en un ejemplo concreto, elegidas de manera que se cubran todos los casos del proceso de inserción y borrado en un árbol roji-negro. El ejemplo elegido es el siguiente:



Este árbol cumple con ser roji-negro, y es el que tomaremos como árbol base, sobre él haremos inserción y borrado de elementos usando cada una de las implementaciones. Si los árboles resultado cumplen ser roji-negros entonces podemos justificar que tanto las funciones auxiliares como las funciones principales de inserción y borrado son válidas.

El árbol anterior se representa en las implementaciones de constructores inteligentes I y aritmética de colores como:

```
T B (T B (T R E 2 E) 4 (T R E 6 E))
```

```
    8
    (T R (T B E 10 E) 12 (T B E 14 E))
```

Y en la implementación de constructores inteligentes II como:

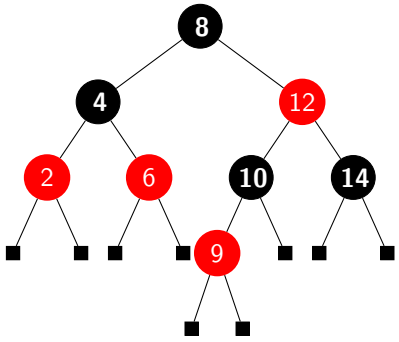
```
(Node B (Node B (Node R E 2 E) 4 (Node R E 6 E))
        8
        (Node R (Node B E 10 E) 12 (Node B E 14 E)))
```

Primero verificamos la corrección del procedimiento de inserción, el cual es el mismo en todas las implementaciones excepto en el caso de tipos anidados que verificaremos más adelante.

Insertemos los nodos 9,3,1,15 y cada vez que se inserte un nuevo nodo tomaremos de entrada el árbol resultado anterior.

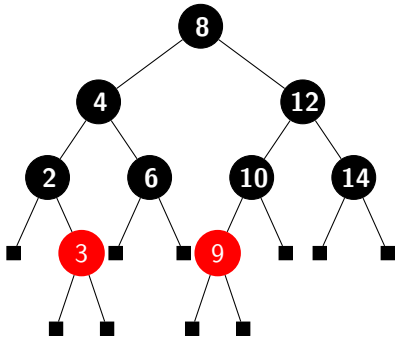
Insertamos 9

```
T B (T B (T R E 2 E) 4 (T R E 6 E))
     8
    (T R (T B (T R E 9 E) 10 E) 12 (T B E 14 E))
```



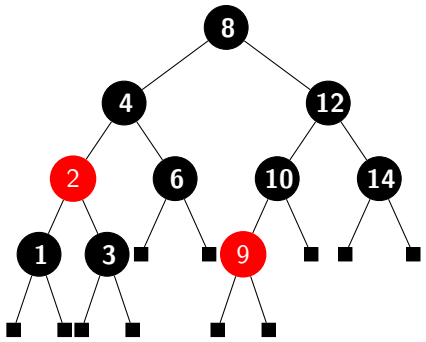
Insertamos 3

```
T B (T B (T B E 2 (T R E 3 E)) 4 (T B E 6 E))
     8
    (T B (T B (T R E 9 E) 10 E) 12 (T B E 14 E))
```



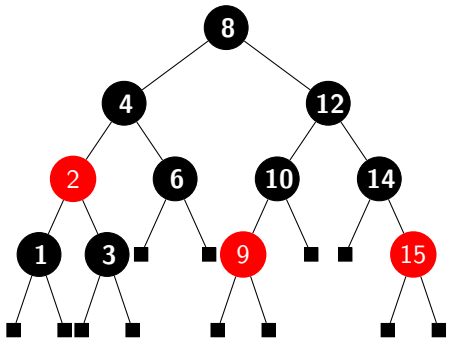
Insertamos 1

```
T B (T B (T R (T B E 1 E) 2 (T B E 3 E)) 4 (T B E 6 E))
     8
    (T B (T B (T R E 9 E) 10 E) 12 (T B E 14 E))
```



Insertamos 15

```
T B (T B (T R (T B E 1 E) 2 (T B E 3 E)) 4 (T B E 6 E))
     8
    (T B (T B (T R E 9 E) 10 E) 12 (T B E 14 (T R E 15 E)))
```

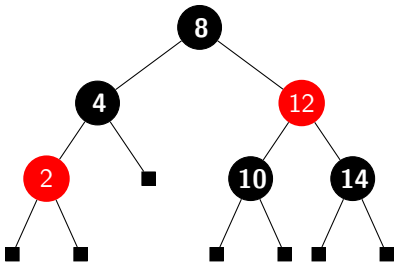


Como podemos observar, al hacer la inserción de un nuevo nodo el árbol resultado mantiene las invariantes, por lo que podemos decir que las funciones auxiliares en la inserción y la inserción misma funcionan.

Para la operación de borrado, tomaremos nuevamente el árbol base de la página 79 y eliminaremos los nodos 6,12 y 8, es decir, cubriremos los casos de quitar un nodo hoja, un nodo interno y un nodo raíz. Cada vez que borremos un nodo, la siguiente eliminación la haremos sobre el árbol resultado anterior.

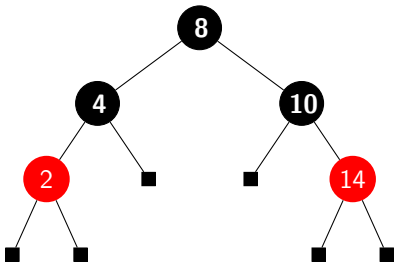
- Constructores Inteligentes I
  Borremos 6

```
T B (T B (T R E 2 E) 4 E)
     8
    (T R (T B E 10 E) 12 (T B E 14 E))
```



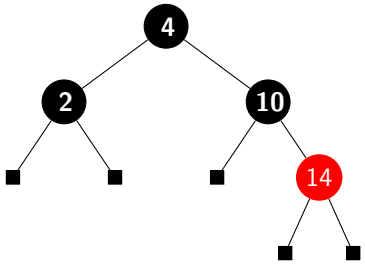
  Borremos 12

```
T B (T B (T R E 2 E) 4 E)
     8
    (T B E 10 (T R E 14 E))
```



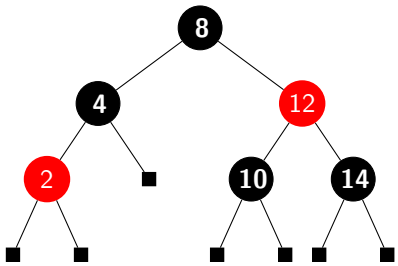
  Borremos 8

```
T B (T B E 2 E)
     4
    (T B E 10 (T R E 14 E))
```



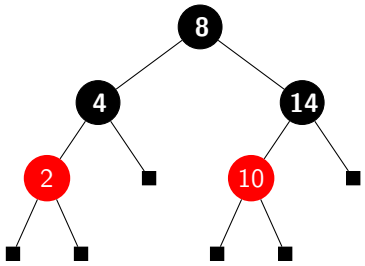
- Constructores Inteligentes II
  Borremos 6

```
Node B (Node B (Node R E 2 E) 4 E)
        8
       (Node R (Node B E 10 E) 12 (Node B E 14 E))
```



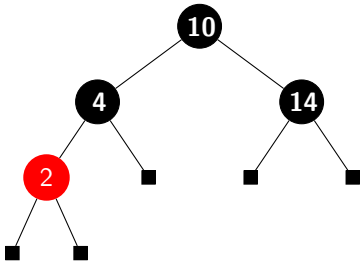
  Borremos 12

```
Node B (Node B (Node R E 2 E) 4 E)
        8
       (Node B (Node R E 10 E) 14 E)
```



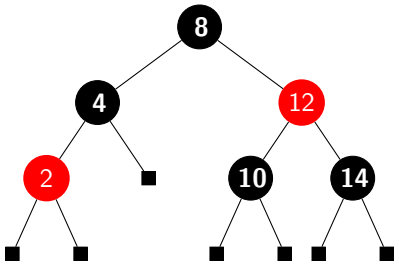
  Borremos 8

```
Node B (Node B (Node R E 2 E) 4 E)
        10
       (Node B E 14 E)
```



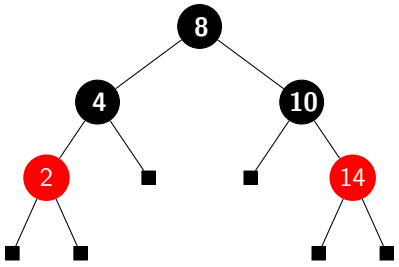
- Aritmética de colores
  Borremos 6

```
T B (T B (T R E 2 E) 4 E)
     8
    (T R (T B E 10 E) 12 (T B E 14 E))
```



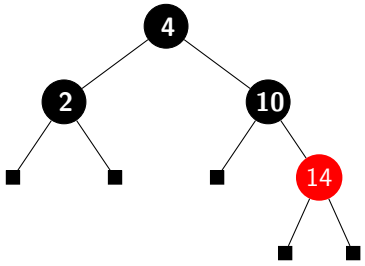
Borremos 12

```
T B (T B (T R E 2 E) 4 E)
     8
    (T B E 10 (T R E 14 E))
```



Borremos 8

```
T B (T B E 2 E) 4 (T B E 10 (T R E 14 E))
```



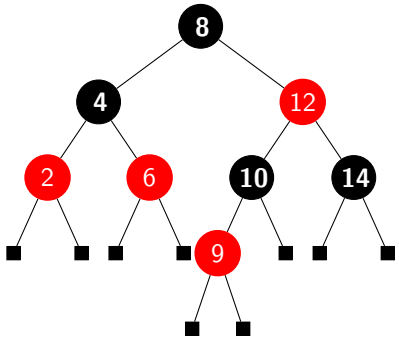
Como podemos ver, al eliminar un nodo de cada árbol resultado se mantienen las invariantes, además para el caso de la aritmética de colores no se tienen nodos de color distinto al establecido, por lo que los procesos auxiliares para el borrado y el borrado funcionan.

Lo siguiente es justificar que la inserción y el borrado con tipos anidados funciona, la metodología que usaremos será la misma, al árbol base de la página 79 insertaremos los elementos 9,3,1 y 15, y posteriormente tomando el mismo árbol base borraremos los nodos 6,12 y 8. Nuestro árbol base se representa como sigue:

```
Suc(Suc(Zero(B(C(B(R(E,2,E),4,R(E,6,E))),
             8,
             R(B(C(E),10,C(E)),12,B(C(E),14,C(E)))))))
```

Insertamos 9

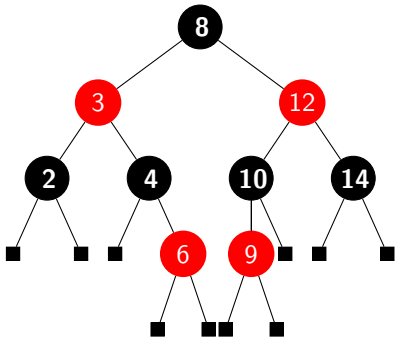
```
Suc(Suc(Zero(B(C(B(R(E,2,E),4,R(E,6,E))),
               8,
               R(B(R(E,9,E),10,C(E)),12,B(C(E),14,C(E)))))))
```



Insertamos 3

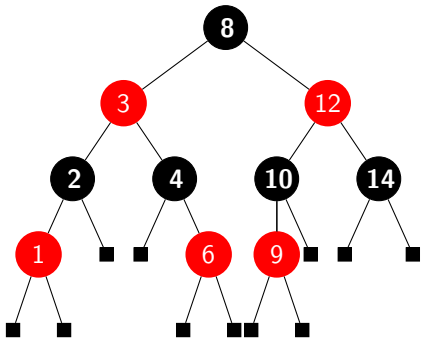
```
Suc(Suc(Zero(B(R(B(C(E),2,C(E)),3,B(C(E),4,R(E,6,E))),
```

```
                8,
                R(B(R(E,9,E),10,C(E)),12,B(C(E),14,C(E)))))))
```



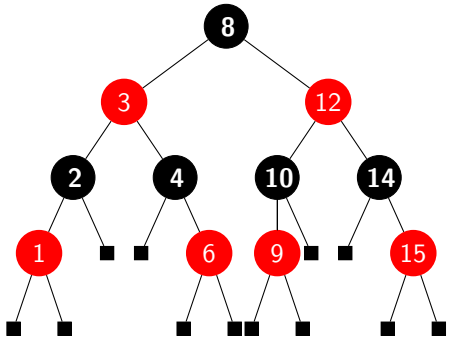
Insertamos 1

```
Suc(Suc(Zero(B(R(B(R(E,1,E),2,C(E)),3,B(C(E),4,R(E,6,E))),
                8,
                R(B(R(E,9,E),10,C(E)),12,B(C(E),14,C(E)))))))
```



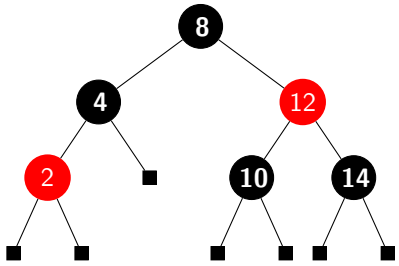
Insertamos 15

```
Suc(Suc(Zero(B(R(B(R(E,1,E),2,C(E)),3,B(C(E),4,R(E,6,E))),
                8,
                R(B(R(E,9,E),10,C(E)),12,B(C(E),14,R(E,15,E)))))))
```



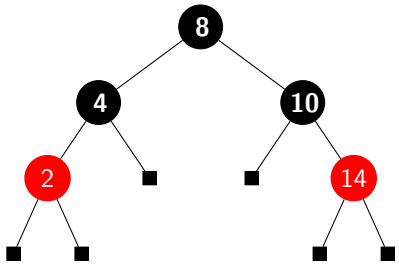
Borramos 6

```
Suc(Suc(Zero(B(C(B(R(E,2,E),4,C(E))),
               8,
               R(B(C(E),10,C(E)),12,B(C(E),14,C(E)))))))
```



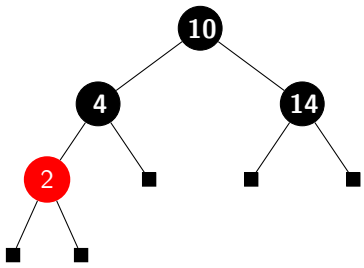
Borramos 12

```
Suc(Suc(Zero(B(C(B(R(E,2,E),4,C(E))),
               8,
               C(B(C(E),10,R(E,14,E)))))))
```



Borramos 8

```
Suc(Suc(Zero(B(C(B(R(E,2,E),4,C(E))),
               10,
               C(B(C(E),14,C(E)))))))
```



Observemos nuevamente que en todos los árboles resultado se mantienen las invariantes, por lo que podemos justificar la validez de los procesos de inserción y borrado con tipos anidados.

Bibliografía

[1] Stephen Adams, *Functional Pearls: Efficient sets—a balancing act.* Journal of functional programming, 3(4), 553-561. 1993.

[2] Andrew W. Appel, *Efficient Verified Red-Black Trees.* Princeton University, Princeton NJ 08540, USA, septiembre 2011.

[3] Rudolf Bayer, *Symmetric binary B-Trees: Data structure and maintenance algorithms.* Acta Informatica 1 (4): 290–306. doi:10.1007/BF00289509, 1972.

[4] Peter Brass, *Advanced data structures.* Cambridge University Press, 1ra Edición 2008.

[5] Thomas H. Cormen, Charles E. Leirserson, et. al., *Introduction to algorithms.* Cambridge, Massachusetts London, England, 3ra Edición 2009.

[6] Kimball Germane, Matthew Might, *Functional pearl. Deletion: The curse of the red-black tree* University of Utah, UT, US, JFP24(4): 423–433, 2014. Cambridge University, doi:10.1017/S0956796814000227

[7] Leonidas J. Guibas, Robert Sedgewick, *A Dichromatic Framework for Balanced Trees.* Proceedings of the 19th Annual Symposium on Foundations of Computer Science. pp. 8–21. doi:10.1109/SFCS.1978.3, 1978.

[8] Ralf Hinze, *Constructing red-black trees.* En Chris Okasaki, editor, Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99), Paris, Francia, 30 Septiembre 1999, pp. 89-99.

[9] Ralf Hinze, *Datentypen zur Darstellung von Mengen und Abbildungen.* Institut für Informatik III, Universität Bonn, Römerstrasse 164, D-53117 Bonn, 15 april 1998.

[10] Graham Hutton, *Red-black trees with types.* J. Functional Programming 11 (4): 425-432, July 2001, Cambridge University Press 2001.

[11] Stefan Kahrs, *Red-black trees with types.* J. Functional Programming 11 (4): 425-432, July 2001, Cambridge University Press 2001.

[12] Matthew Might, `http://matt.might.net/articles/red-black-delete/`, consultada en Julio 2014.

[13] Chris Okasaki, *Purely Functional Data Structures.* Cambridge University Press 1998.

[14] Chris Okasaki, *Functional pearl: Red-black trees in a functional setting.* J. Functional Programming, 9(4), 471-477, 1999.