Type-Safe Red-Black Trees with Java Generics

Stefan Kahrs

University of Kent at Canterbury
Department of computer Science
smk@kent.ac.uk

Abstract. Java Generics[1] are a powerful extension of Java's class system, allowing the static type checker to ensure (as their most typical application) the type-safe usage of homogeneous collections.
The paper shows that the expressiveness of generics goes a very long way, by using them to enforce the tree balancing in a red-black tree implementation. The idea originates from Functional Programming [2, 3], though Java Generics are too different to allow a one-to-one adaptation. Based on experiences with this application, a number of suggestions are made for the improvement of Java Generics.

1 Introduction

Java Generics came about as a response to a long-standing problem in Java: the need for dynamic type casts used to make many Java programs potentially unsafe, with the possibility of class cast exceptions at run-time. In the late 1990s Phil Wadler et al. [4, 5] showed how the Java language could be extended to provided for a richer static analysis that would allow to eliminate dynamic casts almost entirely. Since Java 5 these ideas have entered the language standard [6].

Red-black trees provide a well-known way of implementing balanced 2-3-4 trees as binary trees. They were originally introduced (under a different name) in [7] and are nowadays extensively discussed in the standard literature on algorithms [8, 9]. The basic operations for search-trees (search, insertion, deletion) are realised in $O(\log(n))$ time, for tree size n.

Red-black trees are *binary* search trees with an additional "colour" field which is either *red* or *black*. The root of the tree will always be black. In a red-black tree each red node is required to have black subtrees and is regarded as an intermediate auxiliary node. All binary nodes are required to be of the same *black height*, which is the nesting depth of binary black nodes. Therefore, every binary black node has a black height of $n + 1$, where n is the black height of its subtrees. Moreover, viewing red nodes only as auxiliary nodes, a black binary node of height $n + 1$ has between 2 and 4 black subtrees of height n, depending on whether 0, 1 or 2 of its direct subtrees have red roots. This is the reason why red-black trees can be seen as an implementation of 2-3-4 trees.

Since the mid 1990s it has been known in the Functional Programming community that invariances such as tree-balance can be enforced through a sufficiently rich type system [2], using the feature of "nested" datatypes. Characteristic for these types is that the unfolding of their recursion (see [10]) results in

non-regular infinite trees; this causes problems for type inference [11] and indeed Standard ML's type-inference based type system fails to cope with nested types [12]. However, this does not affect the type-checking of type-annotated code, as in Java Generics.

It would be fairly straightforward to "ape" the FP implementation of red-black trees in an object-oriented setting. However, this would require to move fields into the implementing concrete classes, and consequently operations on fields higher up the class hierarchy would have to be expressed in terms of abstract methods instead. This paper uses a more object-oriented approach that permits directly operating with fields at abstract classes. However, this requires to abstract the types of the fields and as a consequence the generic parameter lists for the classes become more complex.

2 Inheriting Generic Fields

All our red-black trees (even the empty ones) are special cases of binary search trees. Moreover, the abstract class of binary search trees also contains *the only fields* for all of our various tree types. This inheritance of the fields is what makes the implementation overall rather different from its FP cousin.

```
public abstract class Bin<T extends Comparable<? super T>,
                          F extends Find<T>>
    implements Find<T>
{   public T co;
    public F le,ri;

    public T find(T x) {
        int res=co.compareTo(x);
        if (res==0) return co;
        if(res<0) return le.find(x);
        return ri.find(x);
    }
}
```

The fields for the left and right branches of a tree node (`le` and `ri`) are of the generic type `F`, which at this stage just suffices to implement binary search. Subclasses of `Bin` will restrict the type for `F` further, to trees of a particular height and (possibly) a particular colour.

Consequently, other classes in the hierarchy of tree types will need a matching type parameter for `F` — it can only be removed from the class interface once it no longer changes. In a context in which these fields are both read and written to this concrete type needs to be made available as a generic parameter.

3 Typing Red-Black Trees

Figure 1 shows the raw type hierarchy used for the red-black tree implementation; it is "raw" in the sense that it does not include the type parameters, most

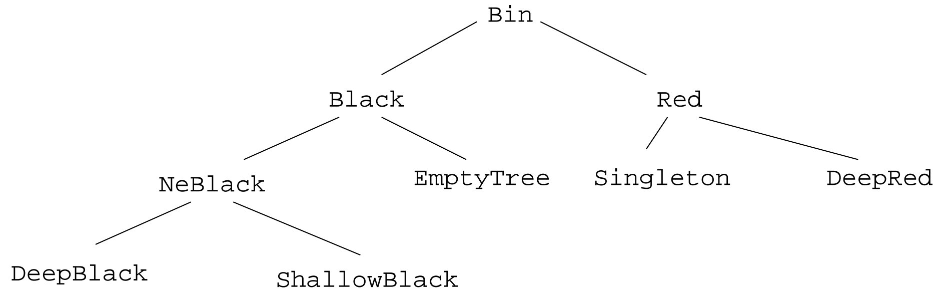

Fig. 1. Raw Type Hierarchy

notably the ones responsible for the GADT effect, though the node colouring is already modelled. Also not included are the interfaces which guide how these types are used, and the types modelling tree underflow.

Clients for this implementation would use the trees through the interface `PureTree` which hides the information about black heights: the trees need to know how high they are, the clients do not. Providing a similar abstraction in Functional Programming is comparatively awkward, as it would typically involve the relatively advanced feature (it is not part of the Haskell-98 standard [13], though all Haskell compilers support it) of existential types, as used in [3]. Java Generics have a form of existential type abstraction of their own: wildcards. To provide a tree type for clients the use of wildcards would have been less appropriate, as `PureTree<T>` would have been replaced by the rather bewildering `Black<T,?,?,?>`.

Generally, the class hierarchy regards all red-black trees as cases of binary search trees. The tree-manipulation methods for red and black trees share common interfaces, but their implementation is considerably different, so that they are given different classes. Within colours, classes are distinguished by height: `Singleton` is the class of red-topped trees of black height 0, `DeepRed` the class of red-topped trees of black height $n+1$; black-topped trees have an even finer distinction, with `DeepBlack` having black height $n+2$. Why this particular height distinction is appropriate will be discussed later.

Notice that such a hierarchy has already considerable implications for the classes: as trees can shrink through deletion and grow through insertion, trees need to have some knowledge of the types that would be attached to the neighbouring levels.

There is an immediate price to pay for this design: we can neither change the colour, nor the height of an existing tree node, because either change requires a change of type. Thus, where the standard imperative implementation of red-black trees freely rewires (and recolours) tree nodes to realise tree rotations to compensate for insertion overflow or deletion underflow, a type-safe implementation will have to frequently discard existing nodes and create fresh ones instead.

4 Tree Heights

Red-black trees are a form of balanced trees. Left and right branch of any node are required to have the same black height. This is enforced in this implementation through the type system, as follows:

- As already seen, left and right branch of any tree node have the same generic type F; thus, if the branch types of the implementing classes encode height information the tree is necessarily balanced.
- Empty trees have (by definition) black height 0, as have singleton red trees.
- Non-empty black trees have type NeBlack<T,B,O>, and their black height is 1 plus the black height of O, which is the type of their branches.
- Red trees have type Red<T,B,O> where B is the type of their (black) branches. Their black height is the same as the black height of type B.
- Red-black trees appear as branches of non-empty black trees. They are instances of the interface RBTree<T,B,O>. The only implementing classes of this interface, i.e. red trees and black trees, set the B parameter to a black tree class of black height n which is equal to their own black height.

Therefore the black height of a red-black tree can be read off its type. The reason this is unambiguous is that the type of the branches occurs as a subexpression in the whole type expression.

The security this gives to the implementation is relative: tree height is maintained as an *invariant*. In itself this does not stop end users from writing further subclasses which could break the invariant — such as ordinary binary search trees, without any balancing whatsoever.

5 Black Trees

The code that sits at abstract class Black looks mostly rather peculiar and mysterious. The reason is that most code that can be placed at that abstract class is code that deals with tree operations relative to some context, such as a tree rotation w.r.t. a parent and grandparent node. Without knowing where that context is coming from these operations make very little sense. However, the type interface for class Black is less obscure:

```
abstract public class Black<
    T extends Comparable<? super T>,
    F extends Find<T>,
    O extends RBTree<T,B,O>,
    B extends Black<T,F,O,B>
> extends Bin<T,F>
  implements PureTree<T>, RBTree<T,B,O>,Embed<T,B,O>
```

The type T is the type of search values; type F is some type which has a method find for searching T values, and type F is used as the types for the left and right

children of the tree. The type `O` is a red-black tree of the same black height, and it is the type of insertion results.

Based on this, class `Black` can already provide the implementation for the insertion and deletion methods when used as a pure tree:

```
public PureTree<T> insert(T x) { return ins(x).makePure(); }
public PureTree<T> delete(T x) { return del(x).makePure(); }
abstract public O ins(T x);
abstract public MakePure<T> del(T x);
```

Subclasses of this will need to constrain the result type of the `del` method further — but this is possible in Java since version 5, using covariant overriding. Without that language feature, class `Black` would need an additional parameter.

The final parameter of the class `B` is a standard trick in Java Generics (see e.g. [6]) to refer to an implementing subclass. The value expressed by `this` is of type `Black<T,F,O,B>`, but this can be too general to be of good use, as it forgets too much information. Instead, the class `Black` leaves its concrete subclasses an abstract method `B self()` to implement; inside the class `Black` calls to `self()` will therefore lose less information than the constant `this` does.

The class implements (promises to implement) three interfaces: all black trees are pure trees, all black trees are red-black trees, and furthermore black trees of height n can be embedded into the red-black trees of height n. The last of these three promises (the interface `Embed`) may appear rather peculiar, and is worth a further look, as it also reveals a weakness in the expressiveness of Java Generics.

The problem is this: all red-black trees implement the `RBTree` interface; however, for height 0 (and height > 0) the interface is extended with methods specific to zero heights and non-zero heights. What we need is that our specialised subclass `B` needs to implement that associated more specialised interface, e.g. be a subclass of `O`. However, Java Generics do not allow subclassing constraints between variables, only between variables and classes. Therefore, within the class `Black` we cannot even make the promise that `B` will be a subclass of `O`.

```
public interface Embed<
    T extends Comparable<? super T>,
    A extends Embed<T,A,B>,
    B extends RBTree<T,A,B>
> extends Find<T> {
    public B embed();
    public B ins(T x);
    public B mkPair(T x,A sibling);
    public NeBlack<T,A,B> mkBpair(T x,A sibling);
}
```

The way around this problem is to require instead a method `embed` which coerces a black tree of whatever height into its associated red-black variety. As all implementing black classes do maintain the property that they subclass their respective red-black variety, the method always has a trivial implementation.

Thus this works but it is not particularly nice and calls for a slight overhaul of Java Generics, e.g. supporting something like this as a replacement for the specification of `B`:

```
B extends Black<T,F,O,B> & O
```

This should express that `B` not only subclasses `Black` in this way, but that it also will maintain a subtype relationship with `O`. This is legal Java syntax for class/interface names, but not for type variables.

The other methods in that class are needed for forming larger trees, e.g. the method `mkBpair` is applied to a value of type `T` and another black tree of the same height n to form a non-empty black tree of height $n+1$; this particular method is used for buffering deletion underflow.

As a final remark, it may appear a little bit peculiar that type `O` is part of the generic parameter list at all. After all, height information is already recoverable from the `F` parameter. It is indeed necessary, because the results of various operations of the `Embed` interface will be stored in fields of a node and thus have to match their type exactly (at this abstract level), and thus when the class `Black` promises to implement `Embed` this type needs to be mentioned itself. There is also a pragmatic reason to keep generic parameters at class level: even if we could remove type `O` from the class interface we would then have to move that abstraction to the level of individual methods, turning them into generic methods — generic methods with rather complex type interfaces.

6 Generalised Algebraic Data Types

Algebraic Data Types in Haskell introduce a fresh type, plus constructors that can be used to create values of this type. In particular, if that new type is a parametric type `A t` then all constructors of the type will be polymorphic functions with return type `A t`. Generalised Algebraic Data Types weaken this condition slightly by allowing that constructors can have return type `A u`, where `u` is a substitution instance of `t`. This generalisation can be used to define, for example, length-aware lists (vectors) as follows:

```
data Vector a n
    where
    Nil :: Vector a ()
    Cons :: a -> Vector a k -> Vector a (S k)
```

In (GHC) Haskell, this defines a fresh type `Vector`, with two type parameters `a` and `n`. The first parameter is the component type of the vector, the second the length of the vector, though this length is encoded as a type. This second parameter has no bearance on the contents of the vectors, it is a phantom type whose sole purpose is to make the use of vectors more rigid. With this definition, vectors of different lengths have different types, e.g. `Cons 4(Cons 7 Nil)` has type `Vector Int (S(S()))`. A motivation for defining length-aware lists is that

they allows us to express indexing operations that are guaranteed to stay within bounds.

When translating ADTs to Java, the type as a whole becomes an abstract class, and each data constructor becomes a new subclass. This works without change in the generalised setting, because when we are subclassing a generic class in Java we can specialise its type parameters. For example, the subclass corresponding to the above `Cons` would commence like this:

```
class Cons<A,K> extends Vector<A,S<K>> {
    A val;
    Vector<A,K> tail;
```

Here, the parameters of Haskell's data constructor become fields of the new class.

Going back to our case study, the subclasses for `Black` and `Red` also use this technique, though as explained earlier the subclasses have a rigid structure anyway and thus there is no need to use phantom types (this would be different for red-black trees without deletion). For example, `EmptyTree` extends `Black` but it specialises the type parameters, as its class header shows:

```
final public class EmptyTree<
    T extends Comparable<? super T>>
    extends Black<T,Find<T>,FlatTree<T>,EmptyTree<T>>
```

Explanation: an empty tree has left and right branches of type `Find<T>` (which will always be `null`), insertion into an empty tree gives a flat tree (red-black tree of black height 0), and the `EmptyTree<T>` in the final argument position is the reflection type of empty trees, which is used e.g. as result type of the `self()` method.

Although empty trees already make use of GADT-style subclassing, the exploitation of the expressive power of the feature is perhaps better understood in connection with the class `NeBlack` of non-empty black trees:

```
abstract public class NeBlack<
    T extends Comparable<? super T>,
    B extends Embed<T,B,O>,
    O extends RBTree<T,B,O>>
extends Black<T,O,DeepTree<T,B,O>,NeBlack<T,B,O>>
```

Explanation: non-empty-black trees (of height $n+1$) store red-black trees of some description (type `O`) as their left and right children. Type `B` is really the type of black trees of height n, but `NeBlack` only needs to know that it implements the `Embed` interface — and using the weaker constraint helps to keep the parameter list of `NeBlack` short, because `Embed` mentions fewer type parameters than `Black`.

The result type of insertions into non-empty black trees is `DeepTree<T,B,O>`, the type of red-black trees of height $n + 1$ (which has a richer interface than `RBTree`). Finally, the reflection type is `NeBlack<T,B,O>`; this suggests that no subclasses of `NeBlack` are needed, as they would reflect into their superclass —

thus, the subclasses `ShallowBlack` and `DeepBlack` are created more for convenience than necessity, as issue that will be discussed later.

Because the class header ensures that the left and right subtree of an `NeBlack` object are themselves red-black trees, it can implement the `ins` method which is required for all black trees:

```
public DeepTree<T,B,O> ins(T x) {
    int res=co.compareTo(x);
    if (res==0) return this;
    if (res<0) return le.insL(x,this);
    return ri.insR(x,this);
}
```

The methods `insL` and `insR` are required in the `RBTree` interface which the fields `le` and `ri` must implement since their type `O` extends that interface. The purpose of these methods is to insert into a subtree, but to provide the parent as a context, in case there is an insertion overflow.

7 Iterative vs. Recursive Insertion

A consequence of the type structure for red-black trees is that insertion and deletion cannot be implemented iteratively (i.e. through loops), the implementation has to be recursive. The reason is that the loop state would not have a proper type: as the algorithms traverse through the tree, the visited nodes have different types. However, a recursive implementation should not need to be less efficient than the iterative implementation: the standard imperative realisation of red-black trees [8] represents every edge in the tree through two pointers going in opposite directions. For the recursive version one pointer per edge suffices, as all required references to parent nodes sit on the run-time stack; in other words, the method-call overhead in the recursive version is quite similar to the double-pointer overhead in iterative implementations.

During insertion and deletion a sufficient amount of parent context is provided to realise the invariant promised in the types of these operations. Moreover, if the root of a subtree changes (because a tree rotation is required) this is signalled to the calling context simply through its return value — the iterative version would instead update the parent through its parent pointer.

Consequently, empty trees are easier to deal with. In particular, the `ins` operation of class `EmptyTree` is the following:

```
public Singleton<T> ins(T x) {
    return new Singleton<T>(x,this);
}
```

Because the `ins` methods return the new root, insertion into an empty tree can be realised by creating and returning a new node. When a tree grows to n internal nodes it will have $n+1$ occurrences of empty trees; however, instead of

doubling the memory requirements for a tree all these empty trees are *the same object*, which is achieved by passing this to the Singleton constructor.

The need for doing this reveals again a slight weakness of Java generics. Because the class EmptyTree is generic in T, the Singleton class cannot keep a static copy of an empty tree, and thus it is passed to it at construction time. This is correct and proper, but why has EmptyTree to be generic in T? The answer is this: EmptyTree "could" implement Black without being given the type parameter T. However, this would make its implementing methods themselves generic, e.g. for the ins method we would get this signature:

```
public <T extends Comparable<? super T>> FlatTree<T> ins(T x);
```

The actual problem with this is that Java does not recognise a generic method as an implementation of a less generic abstract one, and therefore a type-unaware empty tree cannot be used to implement type-aware tree interfaces.

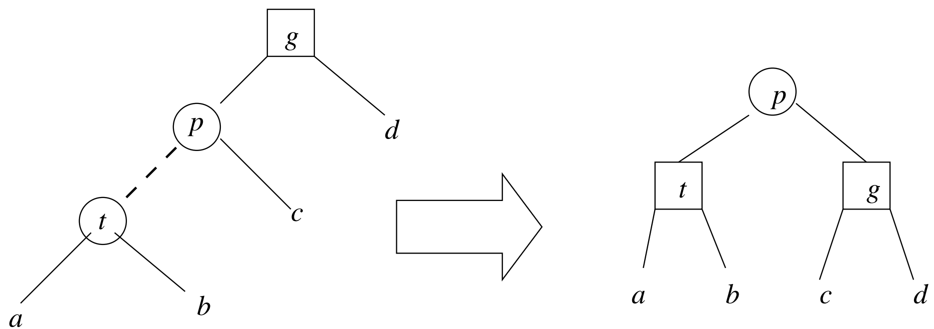

Fig. 2. Insertion with required right rotation

If insertions return new roots so should tree rotations. While this is possible in a recursive and imperative realisation of red-black trees, tree rotations (as methods) do not fit terribly well in this typed setting, because they are not universally applicable with universally meaningful types. Instead they occur in certain contexts, and these contexts need to be provided so that the same effect can be achieved.

An example is shown in figure 2; here, the circular nodes indicate red colour, the square ones black colour. In the example, an insertion into a tree has proceeded in the left child of the red node p and returned another red tree rooted at t. Because of the colour clash this node cannot be placed as the new left child of p — this is indicated by the dashed line. Instead, the attempt to place t as the left child of p is informed by the context, nodes p and g. This is sufficient to be able to form the desired (legal) tree fragment on the right. This tree fragment cannot entirely be created through pointer rewiring either: although node t only changes colour this is now a change of class which consequently requires the creation of a new object; the node p would be re-used as indicated in the

picture, but the height of p on the left and right differs, which again requires a fresh object. Only node g retains colour and height and can be updated as indicated.

Tree rotations of this kind sit fairly high up in the class hierarchy at classes `Black` and `Red` themselves. The object responsible for performing this tree rotation is t; as t is red, the corresponding method sits in the class `Red`:

```
public DeepTree<T,B,O>
goLL(Red<T,B,O> pa, NeBlack<T,B,O> gpa) {
        gpa.le=pa.ri.embed();
        return blacken().mkPair(pa.co,gpa);
}
```

The method `goLL` generally attempts to place *this* object (if possible) as the left child of (red) parent `pa` who is itself left child of (black) grandparent `gpa`; there are four such methods for the four possible left/right combinations. The type parameters to `Red` and `NeBlack` indicate that these have indeed types that fit this scenario. The result type of the rotation is a red-black tree of height $n+1$ and will thus implement the corresponding `DeepTree` interface.

Class `Black` has also four `goXY` methods which in its case (when t is black) will not require tree rotations.

The reason for the call of `embed` is the earlier mentioned problem that Java does not allow us to specify that `B` is a subtype of `O`; to avoid this call the method definition could have been duplicated to the more specific classes `Singleton` and `DeepRed` where this subtype relationship is established.

In class `Red` this method creates two fresh nodes of height $n{+}1$ (one black, one red), and abandons two red nodes of height n. An implementation of red-black trees in which these nodes have the same types could reclaim the abandoned ones instead of allocating new memory.

These four `goXY` methods deal with (potential) insertion overflow, i.e. when an insertion into a black tree might not result in another black tree, in a context where a black tree is required. The classes also have methods that deal with deletion underflow, i.e. when the deletion from a black tree results in a black tree of lower height. These are more complex, because the trees involved in the required rotation are of different heights.

8 Deletion

When the root node of a subtree is earmarked for deletion, the deletion process attempts to replenish that node with the nearest node to it from one of its subtrees (in the actual code it is always the left child, but this is not important), in order to keep the effect of deletion local. This attempt can go wrong in one of two different ways:

- the subtree in question could be empty
- otherwise, the removal of that nearest element might make the subtree underflow, i.e. reduce its height

The first of these problems goes away if the strategy is only applied to red trees of height $n + 1$, and black trees of height $n + 2$; the second problem goes away for trees lower in height. This case distinction is the reason for the chosen class hierarchy.

In its most elementary form a tree underflow arises when the only node of a black tree of height 1 is removed. Deletion underflow is propagated up the tree, until either a red node is encountered which can absorb the underflow, or the root is reached — in which case the overall tree shrinks in height.

Generally, the methods for underflow propagation/absorption are more complex than the methods that deal with insertion overflow, because at absorption point tree nodes from three different height levels are involved. An example for

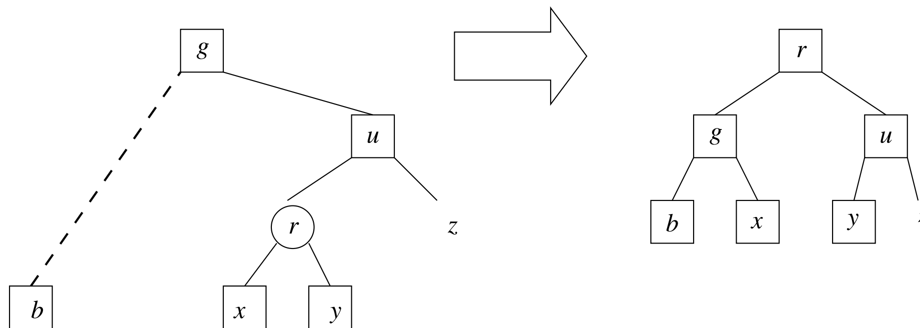

Fig. 3. Absorption of deletion underflow

the absorption process is in figure 3: a deletion has taken place in the left branch of black tree g (of height $n + 2$). The returned result is b, a black tree of height n — the deletion in this branch has underflown. An attempt to absorb the underflow is made by calling a method `absorbL` on u, the right child of g. As it is black, it cannot itself absorb the underflow, but it makes a 2nd-level absorption attempt, by calling `absorbL2` on its left child r.

In this case r is red, and can therefore absorb the deletion underflow by creating the tree fragment on the right, which is a tree of the same height as the original tree. The corresponding code for r sits in class `Red`:

```
public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
absorbL2(B un,DeepBlack<T,B,I> gpa,NeBlack<T,B,I> pa){
    pa.le=ri.embed();
    T aux=gpa.co; gpa.co=co;
    gpa.le=un.mkBpair(aux,le);
    return gpa;
}
```

One could slightly improve the efficiency of this, as it is not really necessary to create a fresh black node: an abandoned black node of the right height still sits at `gpa.le`, though the types do not tell us that it does.

Notice that the result type of `absorbL2` is an instance of `Grey`, an interface that allows for potential deletion underflow. In this case, there was no further underflow, but all non-empty black trees of height $n + 1$ implement the `Grey` interface at height n, so returning `gpa` is well-typed.

However, the type indicates that a further deletion underflow would have been possible. Had r been black then the absorption attempt would have failed, with the situation as in figure 4: In this case, the node r is paired with b, creating

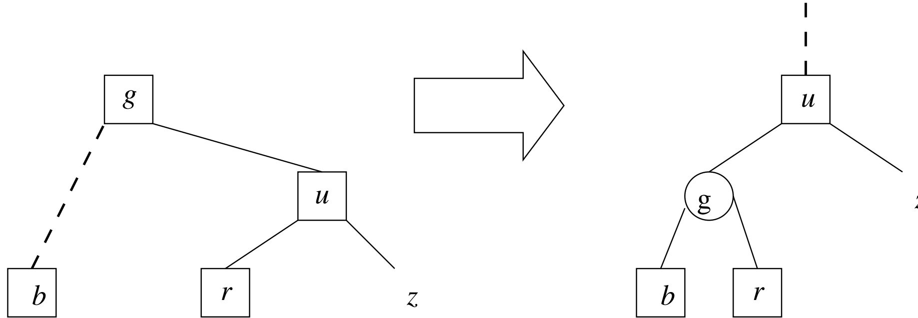

Fig. 4. Propagation of deletion underflow

a new red node as the left child of u. The overall result would have been u, a black tree of height $n+1$ — and therefore the overall result in this section of the tree is another underflow. However, even underflow has a height attached to it, as is apparent in the `absorbL2` method of class `Black` which produces this tree fragment:

```
public Grey<T,NeBlack<T,B,O>,DeepTree<T,B,O>>
absorbL2(B un,DeepBlack<T,B,O> gpa,NeBlack<T,B,O> pa) {
    pa.le=un.mkPair(gpa.co,self());
    return new Underflow<T,NeBlack<T,B,O>,
                         DeepTree<T,B,O>>(pa);
}
```

As `pa` (the u in figure 4) is a non-empty black tree, it does implement the `Grey` interface; however, it implements it at height n, and the signature of the `absorbL2` method requires a grey of height $n + 1$. Therefore a new underflow object is created. The class `Underflow` is just a wrapper for black trees that are "too low" in height. It implements the `Grey` interface, which solely consists of methods for attempting to place a deletion result back into a parent context. The method that triggered the absorption attempts in figures 3 and 4 is the method `putL`, implemented by `Underflow` as follows:

```
public Grey<T,NeBlack<T,B,O>,DeepTree<T,B,O>>
    putL(DeepBlack<T,B,O> pa) {
    return pa.ri.absorbL(body,pa);
}
```

The tree `body` is a field of `Underflow`; it is that mentioned "too low" black tree. The reason this wrapper class is needed is that method selection is not informed by the type parameters of generic classes. Thus we could not make a black tree class implement two different type instances of the `Grey` interface, i.e. one where the heights match and the parent node can be updated, and one where they are one level out of step and the absorption methods are called.

9 Java Language Issues

In the process of developing this application a number of problematic issues about various details in Java Generics were encountered. Only some of them remain nuisances for the final version of the red-black tree implementation, but the other issues remain worth discussing anyway.

Beside language issues, there were also problems with the compiler, e.g. certain type dependencies made it crash, and some complex class/interface relations were unjustly rejected, but this section focuses on language design issues only.

Already mentioned earlier were the problems that it is impossible to specify subtype relationships between type variables, and that generic methods cannot implement non-generic ones. It is not easy to see how the second of these ideas can be sensibly incorporated into Java, but the first one should be possible and worthwhile to do.

A number of further issues are dedicated a subsection each:

9.1 Type Abbreviations

Java has no feature to abbreviate type expressions. Before generics there was very little need for that, although `import` clauses also have a name-abbreviating effect and purpose.

Within generics, type expressions can grow to rather unmanageable levels. Earlier we have seen the return type of `absorbL2` barely fitting onto a single line, and this is a return type quite a few methods share. It is possible to introduce a shorter name by simply subclassing a type expression. This is indeed one of the reasons for providing the (not strictly necessary) class `DeepBlack`; without this subclass, all methods mentioning `DeepBlack<T,B,O>` in their parameter list would have to replace each occurrence of this type by the rather unwieldy:

```
NeBlack<T,NeBlack<T,B,O>,DeepTree<T,B,O>>
```

However, abbreviating a type expression through subclassing still introduces a proper subclass instead of an alias, and this can create problems — which is why the same technique has not been used for example for the return type of methods such as `absorbL2`.

Thus, the author proposes the following Java extension: permit inner class declarations like the following:

```
private class DG =
    Grey<T,NeBlack<T,B,O>,DeepTree<T,B,O>>;
```

This should introduce `DG` as an *alias* for the type expression on the right-hand side, i.e. it should be freely exchangeable with that type expression. Of course, this would not be necessary if Java had a macro language.

9.2 GADTs with closed world assumptions

The issue of not always being able to use subclassing for type abbreviations also points to a subtle distinction between GADTs Haskell-style and their realisation in Java: GADTs come with a closed-world assumption, in that the values formed with their constructors are *the only* values of their defined type.

Going back to the vector example from section 6, values of type `Vector a (S k)` can only be formed using the constructor `Cons`. In the Java realisation of this type, nothing would stop us from defining a third subclass of vectors which also extends `Vector<A,S<K>>`. This distinction can be rather irksome. We may have code that produces a `Vector<A,S<K>>`, and we have written `Cons` as the only instance of that class, and yet they are only assignment-compatible in one direction (unless we resort to casts).

Looking at it differently: in Java we can only *place code* at a generic class when all its generic parameters are uninstantiated. In order to place code at a class like `Vector<A,S<K>>` we have to subclass `Vector` with its second parameter instantiated to `S<K>` and place the code at that subclass. This leaves the possibility of other subclasses of `Vector<A,S<K>>` that may not provide such code. Potential discrepancies of this ilk have complicated the red-black tree implementation tremendously, as precise type information needs to be threaded through various class interfaces to make types fit.

There is not an obvious Java language extension that fits into the Java system and permits such behaviour, which is unfortunate as in practice this is probably the biggest obstacle in the usability of this approach.

10 Conclusions

On the example of red-black trees this paper has shown how properties such as tree-balancing can be expressed within Java Generics, to the extent that the type-checker will guarantee the data invariants. The ideas are known from functional programming but have been adapted for a better fit to Java's features.

In contrast to functional programming, the inability to place code at an instantiated class (without introducing a fresh subclass) is extremely inconvenient in this setting: the resulting potential type incompatibilities have to be counteracted by extending generic parameter lists throughout the implementation.

Some of the implementation comes with a performance penalty: each object has a fixed invariant in this setting. Therefore, object reuse is restricted to situations with the same invariant; instead of changing the invariant of an object a fresh object with different invariants would have to be created that takes its place. In particular, this applies to some of the objects involved in tree rotations.

One can use the height-verified code as the basis for a more efficient (but still recursive) implementation of red-black trees that eliminates this overhead. This involves mostly fine-tuning tree-rotation methods by reclaiming memory we know will become free. While this can be seen as a program transformation, it is one informed by global assumptions about the memory such as single use of tree pointers, and thus goes beyond mere refactoring.

Expressing red-black tree balancing tests Java Generics a fairly long way, and so this journey discovered a few deficiencies of Java Generics, which leads the paper to make several suggestions for improvements.

References

1. Bracha, G.: Add Generic Types to the Java Programming Language. Technical Report JSR-000014, Sun Microsystems (2004)
2. Hinze, R.: Constructing red-black trees. In: Workshop on Algorithmic Aspects of Advanced Programming Languages, Paris (1999) 89–99
3. Kahrs, S.: Red-black trees with types. Journal of Functional Programming **11**(4) (2001) 425–432
4. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In Chambers, C., ed.: ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, BC (1998) 183–200
5. Bruce, K.B., Odersky, M., Wadler, P.: A statically safe alternative to virtual types. In: ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (1998) 523–549
6. Naftalin, M., Wadler, P.: Java Generics. O'Reilly (2007)
7. Bayer, R.: Symmetric binary b-trees: Data structure and maintenance algorithms. Acta Informatica **1** (1972) 290–306
8. Cormen, T.H., Leiserson, C.E., Rivett, R.L.: Introduction to Algorithms. MIT Press (1990)
9. Sedgewick, R.: Algorithms. Addison-Wesley (1988)
10. Cardone, F., Coppo, M.: Two extensions of Curry's type inference system. In Odifreddi, P., ed.: Logic and Computer Science. Academic Press (1990) 19–76
11. Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: Type reconstruction in the presence of polymorphic recursion. ACM Transactions on Programming Languages and Systems **15**(2) (1993) 290–311
12. Kahrs, S.: Limits of ML-definability. In: Proceedings of PLILP'96. (1996) 17–31 LNCS 1140.
13. Jones, S.P., ed.: Haskell 98, Languages and Libraries, The Revised Report. Cambridge University Press (2003)

A The Code

A.1 General search tree classes and interfaces

```
public interface Find<T extends Comparable<? super T>>
    { public T find(T x); }

public interface PureTree<T extends Comparable<? super T>>
    extends Find<T>
{    public PureTree<T> insert(T x);
     public PureTree<T> delete(T x);
}

public abstract class Bin<T extends Comparable<? super T>,F extends Find<T>>
    implements Find<T>
{   public T co; public F le,ri;
    public T find(T x) {
        int res=co.compareTo(x);
        if (res==0) return co;
        if(res<0) return le.find(x);
        return ri.find(x);
    }
}

public interface MakePure<T extends Comparable<? super T>>
{   public PureTree<T> makePure();   }
```

A.2 Interfaces for Red-Black Trees

```
public interface RBTree<
    T extends Comparable<? super T>,
    B extends Embed<T,B,I>,
    I extends RBTree<T,B,I>
>   extends Find<T>,MakePure<T>
{   public DeepTree<T,B,I> insL(T x,NeBlack<T,B,I> pa);
    public DeepTree<T,B,I> insR(T x,NeBlack<T,B,I> pa);
    public DeepTree<T,B,I> goLL(Red<T,B,I> pa, NeBlack<T,B,I> gpa);
    public DeepTree<T,B,I> goLR(Red<T,B,I> pa, NeBlack<T,B,I> gpa);
    public DeepTree<T,B,I> goRL(Red<T,B,I> pa, NeBlack<T,B,I> gpa);
    public DeepTree<T,B,I> goRR(Red<T,B,I> pa, NeBlack<T,B,I> gpa);
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
    absorbL2(B un,DeepBlack<T,B,I> gpa,NeBlack<T,B,I> pa);
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
    absorbR2(B un,DeepBlack<T,B,I> gpa,NeBlack<T,B,I> pa);
    public DeepTree<T,B,I> absorbL(B un,DeepRed<T,B,I> gpa,NeBlack<T,B,I>pa);
    public DeepTree<T,B,I> absorbR(B un,DeepRed<T,B,I> gpa,NeBlack<T,B,I>pa);
}
```

```
public interface FlatTree<T extends Comparable<? super T>>
extends RBTree<T,EmptyTree<T>,FlatTree<T>>
{   public FlatTree<T> del(T x);
    public Grey<T,EmptyTree<T>,FlatTree<T>>join(ShallowBlack<T> pa);
    public Grey<T,EmptyTree<T>,FlatTree<T>>lift(ShallowBlack<T> pa);
    public Grey<T,EmptyTree<T>,FlatTree<T>>
        dropMin(Bin<T,?> subroot,ShallowBlack<T> pa);
}

public interface DeepTree<
    T extends Comparable<? super T>,
    B extends Embed<T,B,I>,
    I extends RBTree<T,B,I>
> extends RBTree<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
{   public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        delL(T x, DeepBlack<T,B,I> p);
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        delR(T x, DeepBlack<T,B,I> p);
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        dropMinL(Bin<T,?> subroot,DeepBlack<T,B,I> pa);
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        dropMinR(Bin<T,?> subroot,DeepBlack<T,B,I> pa);
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        absorbL(B un,DeepBlack<T,B,I> pa);
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        absorbR(B un,DeepBlack<T,B,I> pa);
}
```

A.3 Red Trees

```
public abstract class Red<
    T extends Comparable<? super T>,
    B extends Embed<T,B,I>,
    I extends RBTree<T,B,I>
> extends Bin<T,B>
    implements RBTree<T,B,I>
{
    abstract NeBlack<T,B,I> blacken();
    public DeepTree<T,B,I> goLL(Red<T,B,I> pa, NeBlack<T,B,I> gpa) {
        gpa.le=pa.ri.embed();
        return blacken().mkPair(pa.co,gpa);
    }
    public DeepTree<T,B,I> goLR(Red<T,B,I> pa, NeBlack<T,B,I> gpa) {
        gpa.le=ri.embed(); pa.ri=le;
        return pa.blacken().mkPair(co,gpa);
    }
```

```
    public DeepTree<T,B,I> goRL(Red<T,B,I> pa, NeBlack<T,B,I> gpa) {
        gpa.ri=le.embed(); pa.le=ri;
        return gpa.mkPair(co,pa.blacken());
    }
    public DeepTree<T,B,I> goRR(Red<T,B,I> pa, NeBlack<T,B,I> gpa) {
        gpa.ri=pa.le.embed();
        return gpa.mkPair(pa.co,blacken());
    }
    public DeepTree<T,B,I> insL(T x,NeBlack<T,B,I> p) {
        int res=co.compareTo(x);
        if (res==0) return p;
        if (res<0) return le.ins(x).goLL(this,p);
        return ri.ins(x).goLR(this,p);
    }
    public DeepTree<T,B,I> insR(T x,NeBlack<T,B,I> p) {
        int res=co.compareTo(x);
        if (res==0) return p;
        if (res<0) return le.ins(x).goRL(this,p);
        return ri.ins(x).goRR(this,p);
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
    absorbL2(B un,DeepBlack<T,B,I> gpa,NeBlack<T,B,I> pa){
        pa.le=ri.embed(); T aux=gpa.co; gpa.co=co;
        gpa.le=un.mkBpair(aux,le);
        return gpa;
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
    absorbR2(B un,DeepBlack<T,B,I> gpa,NeBlack<T,B,I> pa){
        pa.ri=le.embed(); T aux=gpa.co; gpa.co=co;
        gpa.ri=ri.mkBpair(aux,un);
        return gpa;
    }
    public DeepTree<T,B,I>
    absorbL(B un,DeepRed<T,B,I> gpa,NeBlack<T,B,I> pa) {
        pa.le=ri.embed(); T aux=gpa.co; gpa.co=co; co=aux;
        ri=le; le=un; gpa.le=blacken();
        return gpa;
    }
    public DeepTree<T,B,I>
    absorbR(B un,DeepRed<T,B,I> gpa,NeBlack<T,B,I> pa) {
        pa.ri=le.embed(); T aux=gpa.co; gpa.co=co; co=aux;
        le=ri; ri=un; gpa.ri=blacken();
        return gpa;
    }
}
```

```
final public class Singleton<T extends Comparable<? super T>>
    extends Red<T,EmptyTree<T>,FlatTree<T>> implements FlatTree<T> {
    public NeBlack<T,EmptyTree<T>,FlatTree<T>> blacken() {
        return new ShallowBlack<T>(le,co,ri);
    }
    public PureTree<T> makePure() { return blacken(); }
    public FlatTree<T> del(T x) {
        int res=co.compareTo(x);
        if (res==0) return le;
        return this;
    }
    public Grey<T,EmptyTree<T>,FlatTree<T>> join(ShallowBlack<T> pa) {
        pa.co=co; pa.le=le;
        return pa;
    }
    public Grey<T,EmptyTree<T>,FlatTree<T>> lift(ShallowBlack<T> pa) {
        pa.co=co; pa.le=le; pa.ri=ri;
        return pa;
    }
    Singleton(T x,EmptyTree<T> nil) {
        co=x; le=ri=nil;
    }
    public Grey<T,EmptyTree<T>,FlatTree<T>>
    dropMin(Bin<T,?> subroot, ShallowBlack<T> pa) {
        subroot.co=co; pa.ri=ri;
        return pa;
    }
}

public class DeepRed<
    T extends Comparable<? super T>,
    B extends Embed<T,B,I>,
    I extends RBTree<T,B,I>
>   extends Red<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
    implements DeepTree<T,B,I>
{
    DeepRed(NeBlack<T,B,I> l, T c, NeBlack<T,B,I> r) {
        le=l; co=c; ri=r;
    }
    DeepBlack<T,B,I> blacken() { return new DeepBlack<T,B,I>(le,co,ri); }
    public PureTree<T> makePure() { return blacken(); }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        dropMinL(Bin<T,?> subroot, DeepBlack<T,B,I> pa) {
        pa.le=ri.dropMin(subroot).rputR(this);
        return pa;
    }
```

```
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        dropMinR(Bin<T,?> subroot, DeepBlack<T,B,I> pa) {
        pa.ri=ri.dropMin(subroot).rputR(this);
        return pa;
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        delL(T x,DeepBlack<T,B,I> pa) {
        int res=co.compareTo(x);
        if (res==0) pa.le=le.dropMin(this).rputL(this);
        else if (res<0) pa.le=le.del(x).rputL(this);
        else pa.le=ri.del(x).rputR(this);
        return pa;
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        delR(T x,DeepBlack<T,B,I> pa) {
        int res=co.compareTo(x);
        if (res==0) pa.ri=le.dropMin(this).rputL(this);
        else if (res<0) pa.ri=le.del(x).rputL(this);
        else pa.ri=ri.del(x).rputR(this);
        return pa;
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        absorbL(B un,DeepBlack<T,B,I> pa){
        T aux=pa.co; pa.ri=ri; pa.co=co; co=aux; ri=le;
        pa.le=ri.le.absorbL(un,this,le);
        return pa;
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        absorbR(B un,DeepBlack<T,B,I> pa){
        T aux=pa.co; pa.le=le; pa.co=co; co=aux; le=ri;
        pa.ri=le.ri.absorbR(un,this,le);
        return pa;
    }
}
```

A.4 Black Trees

```
abstract public class Black<
    T extends Comparable<? super T>,
    F extends Find<T>,
    O extends RBTree<T,B,O>,
    B extends Black<T,F,O,B>
>   extends Bin<T,F> implements PureTree<T>,RBTree<T,B,O>,Embed<T,B,O>
{   public PureTree<T> insert(T x) { return ins(x).makePure(); }
    public PureTree<T> delete(T x) { return del(x).makePure(); }
    public PureTree<T> makePure() { return this; }
```

```
    abstract public O ins(T x);
    abstract public MakePure<T> del(T x);
    abstract public B self();
    public DeepTree<T,B,O> insL(T x,NeBlack<T,B,O> p) {
        p.le=ins(x); return p;
    }
    public DeepTree<T,B,O> insR(T x,NeBlack<T,B,O> p) {
        p.ri=ins(x); return p;
    }
    public Grey<T,NeBlack<T,B,O>,DeepTree<T,B,O>>
    absorbL2(B un,DeepBlack<T,B,O> gpa,NeBlack<T,B,O> pa) {
        pa.le=un.mkPair(gpa.co,self());
        return new Underflow<T,NeBlack<T,B,O>,DeepTree<T,B,O>>(pa);
    }
    public Grey<T,NeBlack<T,B,O>,DeepTree<T,B,O>>
    absorbR2(B un,DeepBlack<T,B,O> gpa,NeBlack<T,B,O> pa) {
        pa.ri=mkPair(gpa.co,un);
        return new Underflow<T,NeBlack<T,B,O>,DeepTree<T,B,O>>(pa);
    }
    public DeepTree<T,B,O> absorbL(B un,DeepRed<T,B,O> gpa,NeBlack<T,B,O> pa) {
        pa.le=un.mkPair(gpa.co,self());
        return pa;
    }
    public DeepTree<T,B,O> absorbR(B un,DeepRed<T,B,O> gpa,NeBlack<T,B,O> pa) {
        pa.ri=mkPair(gpa.co,un);
        return pa;
    }
    public DeepTree<T,B,O> goLL(Red<T,B,O> pa, NeBlack<T,B,O> gpa) {
        return gpa;
    }
    public DeepTree<T,B,O> goLR(Red<T,B,O> pa, NeBlack<T,B,O> gpa) {
        return gpa;
    }
    public DeepTree<T,B,O> goRL(Red<T,B,O> pa, NeBlack<T,B,O> gpa) {
        return gpa;
    }
    public DeepTree<T,B,O> goRR(Red<T,B,O> pa, NeBlack<T,B,O> gpa) {
        return gpa;
    }
}

final public class EmptyTree
    <T extends Comparable<? super T>>
    extends Black<T,Find<T>,FlatTree<T>,EmptyTree<T>>
    implements FlatTree<T>
{   public T find(T x) { return null; }
```

```
    public EmptyTree<T> self() { return this; }
    public FlatTree<T> embed() { return this; }
    public Singleton<T> ins(T x) { return new Singleton<T>(x,this); }
    public FlatTree<T> mkPair(T x,EmptyTree<T> sib)
        { return new Singleton<T>(x,sib); }
    public NeBlack<T,EmptyTree<T>,FlatTree<T>>
        mkBpair(T x,EmptyTree<T> sib) {
        return new ShallowBlack<T>(this,x,sib); }
    public EmptyTree<T> del(T x) { return this; }
    public Grey<T,EmptyTree<T>,FlatTree<T>>
        join(ShallowBlack<T> pa) {
        return pa.ri.lift(pa);
    }
    public Grey<T,EmptyTree<T>,FlatTree<T>>
        lift(ShallowBlack<T> pa) {
        return underflow;
    }
    public Grey<T,EmptyTree<T>,FlatTree<T>>
    dropMin(Bin<T,?> subroot,ShallowBlack<T> pa) {
        subroot.co=pa.co; return pa.le.lift(pa);
    }
    private Underflow<T,EmptyTree<T>,FlatTree<T>> underflow=
        new Underflow<T,EmptyTree<T>,FlatTree<T>>(this);
}

abstract public class NeBlack<
    T extends Comparable<? super T>,
    B extends Embed<T,B,I>,
    I extends RBTree<T,B,I>
>   extends Black<T,I,DeepTree<T,B,I>,NeBlack<T,B,I>>
    implements DeepTree<T,B,I>,Grey<T,B,I>
{   public NeBlack<T,B,I> self() { return this; }
    public DeepTree<T,B,I> ins(T x) {
        int res=co.compareTo(x);
        if (res==0) return this;
        if (res<0) return le.insL(x,this);
        return ri.insR(x,this);
    }
    public DeepTree<T,B,I>
        mkPair(T x,NeBlack<T,B,I> sibling) {
        return new DeepRed<T,B,I>(this,x,sibling);
    }
    public NeBlack<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        mkBpair(T x,NeBlack<T,B,I> sibling) {
        return new DeepBlack<T,B,I>(this,x,sibling);
    }
```

```
    public DeepTree<T,B,I> embed() { return this; }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        delL(T x,DeepBlack<T,B,I> pa) {
        return del(x).putL(pa);
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        delR(T x,DeepBlack<T,B,I> pa) {
        return del(x).putR(pa);
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        putL(DeepBlack<T,B,I> pa) {
        pa.le=this; return pa;
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        putR(DeepBlack<T,B,I> pa) {
        pa.ri=this; return pa;
    }
    public DeepTree<T,B,I> rputL(DeepRed<T,B,I> pa) {
        pa.le=this; return pa;
    }
    public DeepTree<T,B,I> rputR(DeepRed<T,B,I> pa) {
        pa.ri=this; return pa;
    }
    abstract public Grey<T,B,I> dropMin(Bin<T,?> subroot);
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        dropMinR(Bin<T,?> subroot,DeepBlack<T,B,I> pa){
        return dropMin(subroot).putR(pa);
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        dropMinL(Bin<T,?> subroot, DeepBlack<T,B,I> pa){
        return dropMin(subroot).putL(pa);
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        absorbL(B un,DeepBlack<T,B,I> pa){
        return le.absorbL2(un,pa,this);
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        absorbR(B un,DeepBlack<T,B,I> pa) {
        return ri.absorbR2(un,pa,this);
    }
}

final public class ShallowBlack
    <T extends Comparable<? super T>>
    extends NeBlack<T,EmptyTree<T>,FlatTree<T>>
{
```

```
    public Grey<T,EmptyTree<T>,FlatTree<T>> del(T x) {
        int res=co.compareTo(x);
        if (res==0) return le.join(this);
        if (res<0) le=le.del(x);
        else ri=ri.del(x);
        return this;
    }
    ShallowBlack(FlatTree<T> l, T c, FlatTree<T> r)
        { le=l; co=c; ri=r; }
    public Grey<T,EmptyTree<T>,FlatTree<T>>
        dropMin(Bin<T,?> subroot) {
        return ri.dropMin(subroot, this);
    }
}

final public class DeepBlack<
    T extends Comparable<? super T>,
    B extends Embed<T,B,I>,
    I extends RBTree<T,B,I>>
    extends NeBlack<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
{   public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>> del(T x) {
        int res=co.compareTo(x);
        if (res==0) return le.dropMinL(this,this);
        if (res<0) return le.delL(x,this);
        return ri.delR(x,this);
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        dropMin(Bin<T,?> subroot) {
        return ri.dropMinR(subroot, this);
    }
    DeepBlack(DeepTree<T,B,I> l,T c,DeepTree<T,B,I> r) {
        le=l; co=c; ri=r;
    }
}

public interface Embed<
    T extends Comparable<? super T>,
    A extends Embed<T,A,B>,
    B extends RBTree<T,A,B>
> extends Find<T>
{   public B embed();
    public B mkPair(T x,A sibling);
    public B ins(T x);
    public NeBlack<T,A,B> mkBpair(T x,A sibling);
}
```

A.5 Deletion Underflow

```
public interface Grey<
    T extends Comparable<? super T>,
    B extends Embed<T,B,I>,
    I extends RBTree<T,B,I>
>
    extends MakePure<T>
{
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        putL(DeepBlack<T,B,I> pa);
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        putR(DeepBlack<T,B,I> pa);
    public DeepTree<T,B,I> rputL(DeepRed<T,B,I> pa);
    public DeepTree<T,B,I> rputR(DeepRed<T,B,I> pa);
}

public class Underflow<
    T extends Comparable<? super T>,
    B extends Embed<T,B,I> & PureTree<T>,
    I extends RBTree<T,B,I>
>
    implements Grey<T,B,I>
{
    B body;
    Underflow (B x) { body=x; }

    public PureTree<T> makePure() { return body; }

    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        putL(DeepBlack<T,B,I> pa) {
        return pa.ri.absorbL(body,pa);
    }
    public Grey<T,NeBlack<T,B,I>,DeepTree<T,B,I>>
        putR(DeepBlack<T,B,I> pa) {
        return pa.le.absorbR(body,pa);
    }
    public DeepTree<T,B,I> rputL(DeepRed<T,B,I> pa){
        return pa.ri.le.absorbL(body,pa,pa.ri);
    }
    public DeepTree<T,B,I> rputR(DeepRed<T,B,I> pa) {
        return pa.le.ri.absorbR(body,pa,pa.le);
    }
}
```