Red-black trees implementation in Prolog

Roman Pitak
baracudafish@gmail.com

2009-06-16

Contents

1 Introduction to red-black trees

Red-black trees are self-balancing binary search trees. The implementation is complex, but it has a good worst-case running time for its operations and is effective in practice: it can search, insert and delete in O(log n) time, where n is the total number of elements in the tree. In red-black trees the leaf nodes are not relevant and do not contain data. Unlike binary search trees, each node of a red-black tree has a color attribute, the value of which is either red or black. In addition to the ordinary requirements imposed on binary search trees, the following additional requirements of any valid red-black tree apply:

1. A node is either red or black.
2. The root is black. (This rule has little effect on the analysis, since the color of the root can always be changed from red to black.)
3. All leaves are black.
4. Both children of every red node are black.
5. Every simple path from a given node to its descendant leaves contains the same number of black nodes.

These constraints enforce a critical property of red-black trees: that the longest path from the root to any leaf is no more than twice as long as the shortest path from the root to any other leaf in that tree. The result is that the tree is roughly balanced. To see why these properties guarantee this, it suffices to note that no path can have two red nodes in a row, due to property 4.

2 User's manual

This part of the manual focuses on the usage and functionality of the *rb_trees.pl* library. The algorithms used for the implementation are described later in the Programmers manual6.

2.1 Data structures

This implementation represents a node in a red-black tree as a tetrad :
t(Color, Key, LeftSubtree, RightSubtree) where

Color represents the color of the node

Key represents the value stored in the node

LeftSubtree is the left child of the node

RightSubtree is the right child of the node, both represented again as a tetrad, or as *nil*, if the node is a leaf.

Leaves are represented as a node with the *Key = nil* i.e. *t(b, nil, nil, nil)*.

All the data presented to the predicates must obey this format and also satisfy the rules mentioned in the introduction to red-black trees. The *rb_check* predicate can be used to determine whether your data satisfy all the above mentioned rules.

2.2 *rb_check(+Tree).*

The *rb_check* predicate determines whether the given tree is red-black tree valid by the standards of this implementation. The imput is a data structure (presumably a red-black tree) we want to test for the red-black tree properties. It fails if any of the requirements have not been met.

2.3 *rb_empty(?Tree).*

The *rb_empty* predicate, if given a tree succeeds if the tree is empty. If called with an previously ununified 'variable', outputs an empty tree:
t(b, nil, nil, nil).

2.4 *rb_insert(+Tree, +Key, -NewTree).*

The *rb_insert* predicate inserts a new node with the given Key into the given Tree and outputs a new tree, preserving all the required red-black tree properties. Note that due to rotions and color changes the new tree may have a different internal structure (this is only visible/important if you print the trees).

2.5 *rb_build(+List, -Tree).*

The *rb_build* predicate takes a list of elements and builds a red black tree from scratch.

2.6 *rb_delete(+Tree, +Key, -NewTree).*

The *rb_delete* deletes the node with the given Key from the given Tree and outputs a new tree, preserving all the required red-black tree properties. Note that due to rotions and color changes the new tree may have a different internal structure (this is only visible/important if you print the trees).

2.7 *rb_search(+Tree, +Element).*

The *rb_search* predicate searches in the given tree for the given element. Fails if the element is not found.

2.8 *rb_maximum(+Tree, -Maximum).*

The *rb_maximum* predicate finds the maximal element of the given tree (subtree), or fails if given an empty tree.

2.9 *rb_minimum(+Tree, -Minimum).*

The *rb_minimum* predicate finds the minimal element of the given tree (subtree), or fails if given an empty tree.

2.10 *rb_inOrder(+Tree, -List).*

The *rb_inOrder* predicate performs an inorder tree traversal of the given tree and outputs a list of it's elements as defined the rules of an inorder tree walk. Two similar aditional predicates are provided in this implementation:
rb_preOrder(+Tree, -List).
rb_postOrder(+Tree, -List).
These perform preorder and postorder tree traversals.

2.11 *rb_print(+Tree).*

The *rb_print* predicate prints the given tree on the standars output in a human readable format. An example of the *rb_print* output:

```
[ ] Black node
( ) Red node

       |—(19)
 |—[16]
 |     |—(15)
[12]
 |           |—(11)
 |     |—[10]
 |     |     |—(9)
 |—(5)
       |     |—(3)
       |—[2]
             |—(1)
```

3 Programmer's manual

Read-only operations on a red-black tree require no modification from those used for binary search trees, because every red-black tree is a special case of a simple binary search tree. However, the immediate result of an insertion or removal may violate the properties of a red-black tree. Therefore a small number of color changes and no more than three tree rotations may be required.

3.1 Read-only operations

3.1.1 Search

The *rb_search* predicate searches the given *Tree* for the given *Element* in O(log n) time using the binary search tree properties of a red-black tree.

```
% rb_search( +Tree, +Element ).

rb_search( t( b, nil, nil, nil ), _ ) :- !, fail.

rb_search( t( _, K, _, _ ), K ).

rb_search( t( _, K, Ls, _ ), Ks ) :-
  K > Ks,
  rb_search( Ls, Ks ).

rb_search( t( _, K, _, Rs ), Ks ) :-
  K < Ks,
  rb_search( Rs, Ks ).
```

3.1.2 Minimum

The *rb_minimum* predicate finds the minimal element of the given *Tree* (*subree*). That can be achieved by going 'left' in the tree structure.

```
% rb_minimum( +Tree, -Minimum ).
% fails on empty !
rb_minimum( t( _, K, t( b, nil, nil, nil ), _ ), K ).
rb_minimum( t( _, _, Ls, _ ), Minimum ) :-
  rb_minimum( Ls, Minimum ).
```

3.1.3 Maximum

The *rb_maximum* predicate finds the maximal element of the given *Tree* (*subtree*). That can be achieved by going 'right' in the tree structure.

```
% rb_maximum( +Tree, -Maximum ).
% fails on empty !
rb_maximum( t( _, K, _, t( b, nil, nil, nil ) ), K ).
rb_maximum( t( _, _, _, Rs ), Maximum ) :-
  rb_maximum( Rs, Maximum ).
```

3.1.4 Inorder tree traversal

The *rb_inOrder* predicate performs an inorder tree traversal of the given *Tree* by appending the Key of the actual node after the result of the inorder traversal of the left subtree and before the result of the inorder traversal of the right subtree. Resulting in a sorted list of the *Tree* elements.

```
% rb_inOrder( +Tree, -List ).

rb_inOrder( t( b, nil, nil, nil ), [] ).

rb_inOrder( t( _, K, Ls, Rs ), L ) :-
  rb_inOrder( Ls, Ll ),
  rb_inOrder( Rs, Lr ),
  append( Ll, [ K |Lr ], L ).
```

3.1.5 Preorder tree traversal

The *rb_preOrder* predicate performs a preorder tree traversal of the given *Tree*.

```
% rb_preOrder( +Tree, -List ).

rb_preOrder( t( b, nil, nil, nil ), [] ).

rb_preOrder( t( _, K, Ls, Rs ), L ) :-
  rb_preOrder( Ls, Ll ),
  rb_preOrder( Rs, Lr ),
  append( [ K |Ll ], Lr, L ).
```

3.1.6 Postorder tree traversal

The *rb_postOrder* predicate performs a postorder tree traversal of the given *Tree*.

```
% rb_postOrder( +Tree, -List ).

rb_postOrder( t( b, nil, nil, nil ), [] ).

rb_postOrder( t( _, K, Ls, Rs ), L ) :-

  rb_postOrder( Ls, Ll ),
  rb_postOrder( Rs, Lr ),
  append( Ll, Lr, Lp ),
  append( Lp, [ K ], L ).
```

3.1.7 Print

The *rb_print* predicate offers a way to print a red-black tree in a 'human readable' format. It performs a reversed inorder tree traversal printing 'on the go'. Each color of a tree node has it's unique representation:

(1) It prints the Key of a red node in parentheses,

[2] the Key of a black node in square brackets

{3} and the Key of a double black node in curly brackets.

Note: what a doble black node is will be explained later in the Delete section3.3.1.

```
% rb_print( +Tree ).

rb_print( T ) :-
  write( '[ ] Black node' ), nl,
  write( '( ) Red node' ), nl, nl,
  rb_print( T, [] , nodebug ), nl, !.
```

rb_print offers two modes of printing:

- debug mode may be used while debuging parts of the program, while it also prints the 'empty' black leaves. The result is a rather 'longer' tree, but possible leaf errors may be seen.
- no debug mode prints a nice compact tree.

The printing mode can only be secified in the code above.

```
rb_print( t( b, nil, nil, nil ), _, nodebug ).

rb_print( nil, N , debug ) :- !,
  rb_spaces( N ), write( 'nil' ), nl.

rb_print( t( b, nil, Ls, Rs ), N , debug ) :-
  append( N, [ 'r' ], Nr ), rb_print( Rs, Nr , debug ), !,
  rb_spaces( N ), put( '[' ), write( nil ), put( ']' ), nl,
  append( N, [ 'l' ], Nl ), rb_print( Ls, Nl , debug ), !.

rb_print( t( b, K, Ls, Rs ), N , D ) :-
  append( N, [ 'r' ], Nr ), rb_print( Rs, Nr , D ), !,
  rb_spaces( N ), put( '[' ), write( K ), put( ']' ), nl,
  append( N, [ 'l' ], Nl ), rb_print( Ls, Nl , D ), !.

rb_print( t( r, K, Ls, Rs ), N , D ) :-
  append( N, [ 'r' ], Nr ), rb_print( Rs, Nr , D ), !,
  rb_spaces( N ), put( '(' ), write( K ), put( ')' ), nl,
  append( N, [ 'l' ], Nl ), rb_print( Ls, Nl , D ), !.

rb_print( t( bb, K, Ls, Rs ), N , D ) :-
  append( N, [ 'r' ], Nr ), rb_print( Rs, Nr , D ), !,
  rb_spaces( N ), put( '' ), write( K ), put( '' ), nl,
  append( N, [ 'l' ], Nl ), rb_print( Ls, Nl , D ), !.
```

The *rb_spaces* predicate prints the indentation of the nodes, while adding edges to the picture resulting in a more readable representation. A '|' is added only on 'direction changes' and a ' |—' is printed right before the node (at the end of the indentation).

```
rb_spaces( [] ).

rb_spaces( [ _ ] ) :-
  write( ' |—' ).

rb_spaces( [ H, H |T ] ) :-
  write( '      ' ),
  rb_spaces( [ H |T ] ).

rb_spaces( [ _ |T ] ) :-
  write( ' |    ' ),
  rb_spaces( T ).
```

3.1.8 Red-black tree properties validity check

The *rb_check* predicate takes the Tree, checks if the root is black and sends it to the *rb_ch* predicate. The *rb_ch* predicate then checks if all the requirements

of a red-black tree have been met.

```
% rb_check( +Tree ).
rb_check( t( b, K, Ls, Rs ) ) :-
  rb_ch( t( b, K, Ls, Rs ), true, _ ),
  write( 'rb_check successfull' ), nl.
```

- A black empty tree is valid and has one black node in it's path.
- On a node with no children the only requirement is that the Key doesn't unify with 'nil'. Also there are two black nodes in the path : the node itself and it's black leaf.

```
% rb_ch( +Tree, -Validity, -NumberOfBlackNodes ).
% Nbn - number of black nodes

rb_ch( t( b, nil, nil, nil ), true, 1 ).

% black node with no children (subtrees).
rb_ch( t( b, K, t( b, nil, nil, nil ), t( b, nil, nil, nil ) ), true, 2 ) :- !,
  K \= nil.
```

- A black node with a left child (subtree) is valid if it's child is valid, the color of the child is red and the Key of the child is smaller than the Key of the node.

```
% black node with a left child
rb_ch( t( b, K, t( r, Kl, Lsl, Rsl ), t( b, nil, nil, nil ) ), true, 2 ) :- !,
  rb_ch( t( r, Kl, Lsl, Rsl ), Bl, Nl ), !,
  Bl = true,
  Nl = 1,
  K \= nil,
  Kl \= nil,
  K > Kl.
```

- For a black node with a right child the only difference from a black node with a left child is that the Key of the child must be greater than that of the parrent.

```
% black node with a right child
rb_ch( t( b, K, t( b, nil, nil, nil ), t( r, Kr, Lsr, Rsr ) ), true, 2 ) :- !,
  rb_ch( t( r, Kr, Lsr, Rsr ), Br, Nr ), !,
  Br = true,
  Nr = 1,
  K \= nil,
  Kr \= nil,
  K < Kr.
```

- For a black node with both children, the children must be valid for themselves and the count of black nodes in each path of the left child must equal the count of the black nodes in each path of the right child. Also the Key values must meet the requirement for a binary search tree.

```
% black node with both children
rb_ch( t( b, K, t( Cl, Kl, Lsl, Rsl ), t( Cr, Kr, Lsr, Rsr ) ), true, N ) :- !,
  rb_ch( t( Cl, Kl, Lsl, Rsl ), Bl, Nl ), !,
  rb_ch( t( Cr, Kr, Lsr, Rsr ), Br, Nr ), !,
  Bl = true,
  Br = true,
  Nl = Nr,
  N is Nl + 1,
  K \= nil,
  Kl \= nil,
  Kr \= nil,
  K > Kl,
  K < Kr.
```

- A red node with no children is valid if the Key is not 'nil'. It has one black node in each path.

```
% red node with no children
rb_ch( t( r, K, t( b, nil, nil, nil ), t( b, nil, nil, nil ) ), true, 1 ) :- !,
  K \= nil.
```

- A red node with an 'only-child' is never to be found in a red-black tree, because it always violates Property 5 (Every simple path from a given node to its descendant leaves contains the same number of black nodes).
- The last valid possibility is a red node with both children. They both have to be black and valid and the count of black nodes in each path of the left child must equal the count of the black nodes in each path of the right child.

```
% red node with both children
rb_ch( t( r, K, t( b, Kl, Lsl, Rsl ), t( b, Kr, Lsr, Rsr ) ), true, N ) :- !,
  rb_ch( t( b, Kl, Lsl, Rsl ), Bl, Nl ), !,
  rb_ch( t( b, Kr, Lsr, Rsr ), Br, Nr ), !,
  Bl = true,
  Br = true,
  Nl = Nr,
  N is Nl + 0,
  K \= nil,
  Kl \= nil,
  Kr \= nil,
  K > Kl,
  K < Kr.
```

- The *default false* case only unifies if none of the previous (valid) predicates did, which denotes the given subtree **T** 'invalid'. **T** is printed for debuging purposes and and the predicate *rb_check* fails.

```
% default FALSE
rb_ch( T, false, 0 ) :- !,
  rb_print( T ).
```

3.2 Insertion

The insertion begins by adding a node much as we would in a simple binary search tree and coloring it red, but instead of adding a leaf we add a red interior node with two black leaves in place of an existing black leaf.

3.2.1 *rb-insert*

```
% rb-insert( +Tree, +Key, -Tree ).

rb-insert( T1, K, T3 ) :-
   rb-ins( T1, K, T2 ),
   rb-reblackRoot( T2, T3 ), !.

% rb-ins( +Tree, +Key, -Tree ).

rb-ins( t( b, nil, nil, nil ), K, t( r, K, t( b, nil, nil, nil ), t( b, nil, nil, nil ) ) ).

rb-ins( t( C, K, Ls, Rs ), K, t( C, K, Ls, Rs ) ).

rb-ins( t( C, K, Ls, Rs ), Ki, Tfixed ) :-
   K < Ki,
   rb-ins( Rs, Ki, Rsi ),
   rb-insFixup( t( C, K, Ls, Rsi ), Tfixed ).

rb-ins( t( C, K, Ls, Rs ), Ki, Tfixed ) :-
   K > Ki,
   rb-ins( Ls, Ki, Lsi ),
   rb-insFixup( t( C, K, Lsi, Rs ), Tfixed ).

% rb-reblackRoot( +Tree, -Tree ).
rb-reblackRoot( t( _, K, Ls, Rs ), t( b, K, Ls, Rs ) ).
```

After each insertion an insert fixup procedure is called to perform rotations and color changes. What happens next depends on the color of the nearby nodes. The term 'uncle node' will be used to refer to the sibling of a nodes parrent, as in human family trees. Note that:

Property 3 (All leaves are black) always holds.

Property 4 (Both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation.

Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is threatened only by adding a black node, repainting a red node black, or a rotation.

- The label **N** will be used to denote the node being inserted, **P** will denote **N**'s parent node, **G** will denote **N**'s grandparent, and **U** will denote **N**'s uncle.

3.2.2 Insert fixup

Several cases may occur, depending on the position of the inserted node:

Case 1 The new node **N** is at the root of the tree. In this case, **N** is repainted black to satisfy Property 2 (The root is black). Since this adds one black node to every path at once, Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is not violated.

Case 2 The new node's parent **P** is black, so Property 4 (Both children of every red node are black) is not invalidated. In this case, the tree is still valid. Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is not threatened, because **N** is red.

Case 3 If both the parent **P** and the uncle **U** are red, then both nodes can be repainted black and the grandparent **G** becomes red (to maintain Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes)). Now, the new red node **N** has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed. However, the grandparent **G** may now violate properties 2 (The root is black) or 4 (Both children of every red node are black). To fix this, this procedure is recursively performed on **G**. Four symmetrical cases had to be implemented in the code (**P** being **G**'s left or right child and **N** being **P**'s left or right child.)

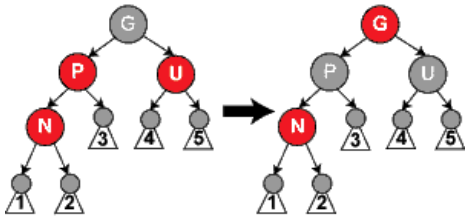

Figure 1: Transformation of a red-black subtree in case 3 of the insert fixup.

```
% CASE #3

rb-insFixup( t( b, Kz, t( r, Ky, t( r, Kx, Lsx, Rsx ), Rsy ), t( r, Ku, Lsu, Rsu ) ), t( r,
Kz, t( b, Ky, t( r, Kx, Lsx, Rsx ), Rsy ), t( b, Ku, Lsu, Rsu ) ) ).

rb-insFixup( t( b, Kz, t( r, Ky, Lsy, t( r, Kx, Lsx, Rsx ) ), t( r, Ku, Lsu, Rsu ) ), t( r,
Kz, t( b, Ky, Lsy, t( r, Kx, Lsx, Rsx ) ), t( b, Ku, Lsu, Rsu ) ) ).

rb-insFixup( t( b, Kz, t( r, Ku, Lsu, Rsu ), t( r, Ky, t( r, Kx, Lsx, Rsx ), Rsy ) ), t( r,
Kz, t( b, Ku, Lsu, Rsu ), t( b, Ky, t( r, Kx, Lsx, Rsx ), Rsy ) ) ).

rb-insFixup( t( b, Kz, t( r, Ku, Lsu, Rsu ), t( r, Ky, Lsy, t( r, Kx, Lsx, Rsx ) ) ), t( r,
Kz, t( b, Ku, Lsu, Rsu ), t( b, Ky, Lsy, t( r, Kx, Lsx, Rsx ) ) ) ).
```

Case 4 The parent **P** is red but the uncle **U** is black; also, the new node **N** is the right child of **P**, and **P** is the left child of its parent **G**. In this case, a left rotation is performed on **P**. Because Property 4 (Both children of every red node are black) is still violated, *rb-insFixup* is called again on the same subtree. As can be easily seen from the Figures 2 and 3, Case 4 always leeds on Case 5. The rotation causes some paths (those in the subtree labelled "1") to pass through the new node where they did not before, but both these nodes are red, so Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is not violated by the rotation.

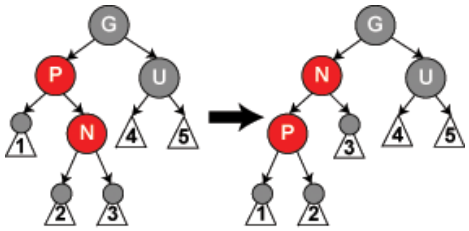

Figure 2: Transformation of a red-black subtree in case 4 of the insert fixup.

```
% CASE #4

rb-insFixup( t( b, Kz, t( r, Ky, Lsy, t( r, Kx, Lsx, Rsx ) ), Uncle ), Tfixed ) :-
  rb-insFixup( t( b, Kz, t( r, Kx, t( r, Ky, Lsy, Lsx ), Rsx ), Uncle ), Tfixed ).

rb-insFixup( t( b, Kz, Uncle, t( r, Ky, t( r, Kx, Lsx, Rsx ), Rsy ) ), Tfixed ) :-
  rb-insFixup( t( b, Kz, Uncle, t( r, Kx, Lsx, t( r, Ky, Rsx, Rsy ) ) ), Tfixed ).
```

Case 5 The parent **P** is red but the uncle **U** is black, the new node **N** is the left child of **P**, and **P** is also the left child of its parent **G**. In this case a right rotation is performed on **G**. Then, the colors of **P** and **G** are switched, and the resulting tree satisfies Property 4 (Both children of every red node are black). Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) also remains satisfied, since all paths that went through any of these three nodes went through **G** before, and now they all go through **P**. In each case, this is the only black node of the three.

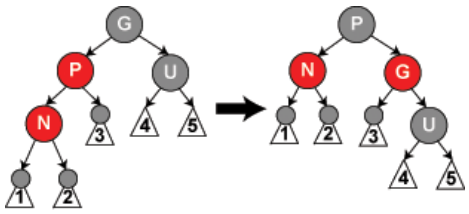

Figure 3: Transformation of a red-black subtree in case 5 of the insert fixup.

```
% CASE #5

rb-insFixup( t( b, Kz, t( r, Ky, t( r, Kx, Lsx, Rsx ), Rsy ), Uncle ), t( b, Ky, t( r, Kx,
Lsx, Rsx ), t( r, Kz, Rsy, Uncle ) ) ).

rb-insFixup( t( b, Kz, Uncle, t( r, Ky, Lsy, t( r, Kx, Lsx, Rsx ) ) ), t( b, Ky, t( r, Kz,
Uncle, Lsy ), t( r, Kx, Lsx, Rsx ) ) ).
```

If none of the cases 3, 4 or 5 unifies with the subtree given to the *rb-insFixup* predicate, the tree is presumed to satisfy the properties of a red-black tree (except for Property 2 (The root is always black), which is handled separately by the *rb-reblackRoot* predicate on the very end of the insertion). In this case the tree passes through the 'default' predicate, that does not alter it's structure.

```
rb-insFixup( T, T ).
```

3.2.3 Building a red-black tree from a list of elements

To build a red-black tree from a list of elements, *rb-insert* is called on every element from the list.

```
% rb-build( +List, -Tree )

rb-build( L, T ) :-
  rb-empty( T0 ),
  rb-b( L, T0, T ).

rb-b( [], T, T ).

rb-b( [ K ], T1, T2 ) :-
  rb-insert( T1, K, T2 ).

rb-b( [ H |T ], T1, T3 ) :-
  rb-insert( T1, H, T2 ),
  rb-b( T, T2, T3 ).
```

3.3 Delete

In a normal binary search tree, when deleting a node with two non-leaf children, we find either the maximum element in its left subtree or the minimum element in its right subtree, and move its value into the node being deleted (as shown in Figure 4). We then delete the node we copied the value from, which must have less than two non-leaf children. Because merely copying a value does not violate any red-black properties, this reduces the problem of deleting to the problem of deleting a node with at most one non-leaf child. It does not matter whether this node is the node we originally wanted to delete or the node we copied the value from.

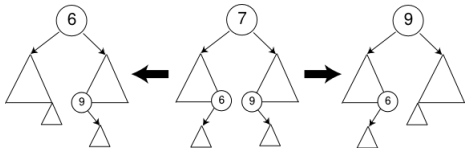

Figure 4: Binary search tree delete.

3.3.1 rb-delete

When deleting from a red-black tree several cases may occur most of which are trivial to implement.

:) For the remainder of this discussion we can assume we are deleting a node with at most one non-leaf child, which we will call its child (if it has only leaf children, let one of them be its child).

:) If we are deleting a red node, we can simply replace it with its child, which must be black. All paths through the deleted node will simply pass through one less red node, and both the deleted node's parent and child must be black, so properties 3 (All leaves, including nulls, are black) and 4 (Both children of every red node are black) still hold.

:) Another simple case is when the deleted node is black and its child is red. Simply removing a black node could break Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes), but if we repaint its child black, both of these properties are preserved.

:(The complex case is when both the node to be deleted and its child are black. We begin by replacing the node to be deleted with its child. We will call (or label) this child (in its new position) **N**, and its sibling (its new parent's other child) **S**. In the diagrams below, we will also use **P** for **N**'s new parent, $\mathbf{S}_L$for **S**'s left child, and $\mathbf{S}_R$for **S**'s right child (it can be shown that S cannot be a leaf).

The implementation of the delete consists of calling an accessory predicate *rbdel* and then reblacking the root (by the same predicate as in *rb_insert*).

```
% rb_delete( +Tree, +Key, -Tree ).
rb_delete( T, K, Trb ) :-
  rb_del( T, K, Td ),
  rb_reblackRoot( Td, Trb ).
```

There are several simple cases of delete. These do not violate the restrictions of red-black trees.

```
% The Key has not been found
rb_del( t( b, nil, nil, nil ), _, t( b, nil, nil, nil ) ).

% deleting a red node with no children
rb_del( t( r, K, t( b, nil, nil, nil ), t( b, nil, nil, nil ) ), K, t( b, nil, nil, nil ) ).

% deleting a red node with a left child
rb_del( t( r, K, Ls, t( b, nil, nil, nil ) ), K, Ls ).

% deleting a red node with a right child
rb_del( t( r, K, t( b, nil, nil, nil ), Rs ), K, Rs ).

% deleting a black node with a red left child
rb_del( t( b, K, t( r, Kl, Lsl, Rsl ), t( b, nil, nil, nil ) ), K, t( b, Kl, Lsl, Rsl ) ).

% deleting a black node with a red right child
rb_del( t( b, K, t( b, nil, nil, nil ), t( r, Kr, Lsr, Rsr ) ), K, t( b, Kr, Lsr, Rsr ) ).
```

Note: Since a black node cannot have a black 'only-child' (It would always violate Property 5 (Every simple path from a given node to its descendant leaves contains the same number of black nodes.)), this case doesn't have to be implemented.

The only case violating the red-black tree properties is the delete of a black node with no children. If doing so, we have to make up for the lost black node. A nice way to implemet this is recoloring the new node 'double black'. The *rb_delFixup* will take care of that later.

```
% black node with no children
rb_del( t( b, K, t( b, nil, nil, nil ), t( b, nil, nil, nil ) ), K, t( bb, nil, nil, nil ) ).
```

The process of deleting a node with both children as described above is implemented here, along with branching the delete. In both cases this is the place where *rb_delFixup* is actually called, since these are the places where the recursion is performed. The result of this is, that whatever violations are not repaired, but passed higher, are taken care of in the next step (previous iteration).

```
% deleting a node with both children

rb_del( t( C, K, Ls, Rs ), K, Tfixed ) :-
  rb_maximum( Ls, Max ),
  rb_del( Ls, Max, Lsd ),
  rb_delFixup( t( C, Max, Lsd, Rs ), Tfixed ).

% branching the delete

rb_del( t( C, K, Ls, Rs ), Kd, Tfixed ) :-
  K > Kd,
  rb_del( Ls, Kd, Lsd ),
  rb_delFixup( t( C, K, Lsd, Rs ), Tfixed ).

rb_del( t( C, K, Ls, Rs ), Kd, Tfixed ) :-
  K < Kd,
  rb_del( Rs, Kd, Rsd ),
  rb_delFixup( t( C, K, Ls, Rsd ), Tfixed ).
```

3.3.2 rb-delFixup

Note: In order that the tree remains well-defined, we need that every null leaf remains a leaf after all transformations (that it will not have any children). If the node we are deleting has a non-leaf (non-null) child **N**, it is easy to see that the property is satisfied. If, on the other hand, **N** would be a null leaf, it can be verified from the diagrams (or code) for all the cases that the property is satisfied as well.

If both N and its original parent are black, then deleting this original parent causes paths which proceed through N to have one fewer black node than paths that do not. As this violates Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes), the tree must be rebalanced. There are several cases to consider:

Case 1 **N** is the new root. In this case, we are done. We removed one black node from every path, and the new root is black, so the properties are preserved. This is actually taken care of by the *reblackRoot* predicate at the very end of the *rb_delete*.

Case 2 **S** is red. In this case we reverse the colors of **P** and **S**, and then rotate left at **P**, turning **S** into **N**'s grandparent. Note that **P** has to be black as it had a red child. Although all paths still have the same number of black nodes, now **N** has a black sibling and a red parent, so we can proceed to step 4, 5, or 6. (Its new sibling is black because it was once the child of the red S.) In later cases, we will relabel **N**'s new sibling as **S**.

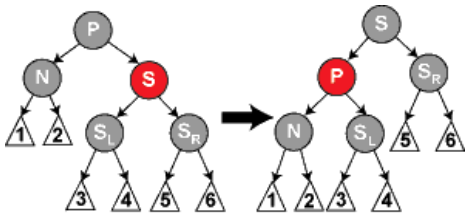

Figure 5: Transformation of a red-black subtree in case 2 of the delete fixup.

```
% CASE #2

rb_delFixup( t( b, Kp, N, t( r, Ks, Lss, Rss ) ), t( b, Ks, Tfixed, Rss ) ) :-
  N = t( bb, _, _, _ ),
  rb_delFixup( t( r, Kp, N, Lss ), Tfixed ).

rb_delFixup( t( b, Kp, t( r, Ks, Lss, Rss ), N ), t( b, Ks, Lss, Tfixed ) ) :-
  N = t( bb, _, _, _ ),
  rb_delFixup( t( r, Kp, Rss, N ), Tfixed ).
```

Case 3 **P**, **S**, and **S**'s children are black. In this case, we simply repaint **S** red. The result is that all paths passing through **S**, which are precisely those paths not passing through **N**, have one less black node. Because deleting **N**'s original parent made all paths passing through **N** have one less black node, this evens things up. However, all paths through **P** now have one fewer black node than paths that do not pass through **P**, so Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is still violated. To correct this, we perform the rebalancing procedure on **P**, starting at case 1.

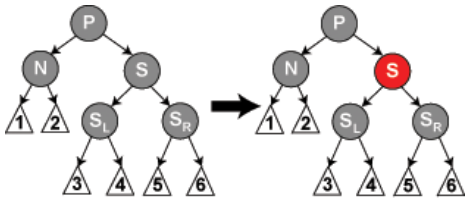

Figure 6: Transformation of a red-black subtree in case 3 of the delete fixup.

```
% CASE #3

rb_delFixup( t( b, Kp, t( bb, Kn, Lsn, Rsn ), t( b, Ks, Lss, Rss ) ),
             t( bb, Kp, t( b, Kn, Lsn, Rsn ), t( r, Ks, Lss, Rss ) ) ) :-
  Lss = t( b, _, _, _ ),
  Rss = t( b, _, _, _ ).

rb_delFixup( t( b, Kp, t( b, Ks, Lss, Rss ), t( bb, Kn, Lsn, Rsn ) ),
             t( bb, Kp, t( r, Ks, Lss, Rss ), t( b, Kn, Lsn, Rsn ) ) ) :-
  Lss = t( b, _, _, _ ),
  Rss = t( b, _, _, _ ).
```

Case 4 **S** and **S**'s children are black, but **P** is red. In this case, we simply exchange the colors of **S** and **P**. This does not affect the number of black nodes on paths going through **S**, but it does add one to the number of black nodes on paths going through **N**, making up for the deleted black node on those paths.

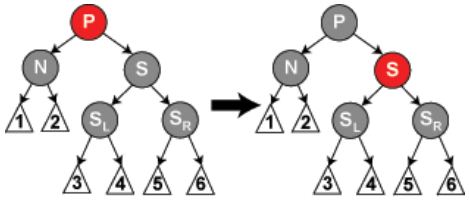

Figure 7: Transformation of a red-black subtree in case 4 of the delete fixup.

```
% CASE #4

rb_delFixup( t( r, Kp, N, t( b, Ks, Lss, Rss ) ),
             t( b, Kp, N2, t( r, Ks, Lss, Rss ) ) ) :-
  N = t( bb, Kn, Lsn, Rsn ),
  N2 = t( b, Kn, Lsn, Rsn ),
  Lss = t( b, _, _, _ ),
  Rss = t( b, _, _, _ ).

rb_delFixup( t( r, Kp, t( b, Ks, Lss, Rss ), N ),
             t( b, Kp, t( r, Ks, Lss, Rss ), N2 ) ) :-
  N = t( bb, Kn, Lsn, Rsn ),
  N2 = t( b, Kn, Lsn, Rsn ),
  Lss = t( b, _, _, _ ),
  Rss = t( b, _, _, _ ).
```

Case 5 **S** is black, **S**'s left child is red, **S**'s right child is black, and **N** is the left child of its parent. In this case we rotate right at **S**, so that **S**'s left child becomes **S**'s parent and **N**'s new sibling. We then exchange the colors of **S** and its new parent. All paths still have the same number of black nodes, but now **N** has a black sibling whose right child is red, so we fall into case 6. Neither **N** nor its parent are affected by this transformation. (Again, for case 6, we relabel **N**'s new sibling as **S**.)

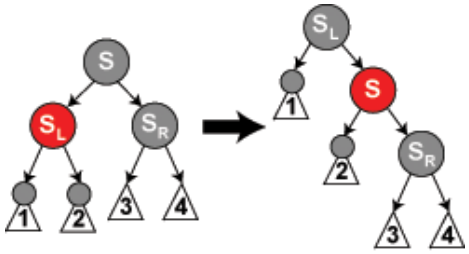

Figure 8: Transformation of a red-black subtree in case 5 of the delete fixup.

```
% CASE #5

rb_delFixup( t( Cp, Kp, N, S ), Tfixed ) :-
  N = t( bb, _, _, _ ),
  S = t( b, Ks, Lss, Rss ),
  Lss = t( r, Ksl, Lsl, Rsl ),
  Rss = t( b, _, _, _ ),
  rb_delFixup( t( Cp, Kp, N, t( b, Ksl, Lsl, t( r, Ks, Rsl, Rss ) ) ), Tfixed ).

rb_delFixup( t( Cp, Kp, S, N ), Tfixed ) :-
  N = t( bb, _, _, _ ),
  S = t( b, Ks, Lss, Rss ),
  Lss = t( b, _, _, _ ),
  Rss = t( r, Ksr, Lsr, Rsr ),
  rb_delFixup( t( Cp, Kp, t( b, Ksr, t( r, Ks, Lss, Lsr ), Rsr ), N ), Tfixed ).
```

Case 6 **S** is black, **S**'s right child is red, and **N** is the left child of its parent **P**. In this case we rotate left at **P**, so that **S** becomes the parent of **P** and **S**'s right child. We then exchange the colors of **P** and **S**, and make **S**'s right child black. The subtree still has the same color at its root, so Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) are not violated. However, **N** now has one additional black ancestor: either **P** has become black, or it was black and **S** was added as a black grandparent. Thus, the paths passing through **N** pass through one additional black node.

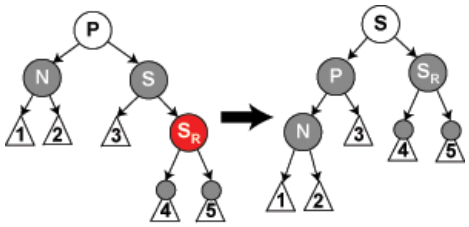

Figure 9: Transformation of a red-black subtree in case 6 of the delete fixup.

```
% CASE #6

rb_delFixup( t( Cp, Kp, N, S ),
             t( Cp, Ks, t( b, Kp, N2, Lss ), Rss2 ) ) :-
  N = t( bb, Kn, Lsn, Rsn ),
  N2 = t( b, Kn, Lsn, Rsn ),
  S = t( b, Ks, Lss, Rss ),
  Rss = t( r, Kr, Lsr, Rsr ),
  Rss2 = t( b, Kr, Lsr, Rsr ).

rb_delFixup( t( Cp, Kp, S, N ),
             t( Cp, Ks, Lss2, t( b, Kp, Rss, N2 ) ) ) :-
  N = t( bb, Kn, Lsn, Rsn ),
  N2 = t( b, Kn, Lsn, Rsn ),
  S = t( b, Ks, Lss, Rss ),
  Lss = t( r, Kl, Lsl, Rsl ),
  Lss2 = t( b, Kl, Lsl, Rsl ).
```

Meanwhile, if a path does not go through **N**, then there are two possibilities:

- It goes through **N**'s new sibling. Then, it must go through **S** and **P**, both formerly and currently, as they have only exchanged colors and places. Thus the path contains the same number of black nodes.
- It goes through **N**'s new uncle, **S**'s right child. Then, it formerly went through **S**, **S**'s parent, and **S**'s right child (which was red), but now only goes through **S**, which has assumed the color of its former parent, and **S**'s right child, which has changed from red to black (assuming **S**'s color: black). The net effect is that this path goes through the same number of black nodes.

Either way, the number of black nodes on these paths does not change. Thus, we have restored Properties 4 (Both children of every red node are black) and

5 (All paths from any given node to its leaf nodes contain the same number of black nodes). The white node in the diagram can be either red or black, but must refer to the same color both before and after the transformation.

4 Summary

The work on this implemetation went fairly well. It was not an easy task, particularly *rb_delete* gave me a hard time, since I had to implement it twice. The first version did not work in some cases and was impossible to debug, because of my horrible coding style. I tried to write the second version a bit more comprehensible and I am pleased with the result. The *rb_insert* is written ugly but correctly. If anyone would intend to alter it I would recomend rewriting it into a more comprehensible form first. I only implemeted basic operation like insert and delete, because using these it is fairly simple to write more complex ones like merge e.i.

Materials issued in [3] gave me a nice inside into red-black trees.

Mostly when writing the documentation I have used materials from [2]. (All the pictures and some text are directly copied from there)

References

[1] Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms* MIT Press

[2] `http://en.wikipedia.org/wiki/Red-black_trees`

[3] `http://kti.mff.cuni.cz/~cepek/ADS1.ppt`

[4] `http://reptar.uta.edu/NOTES5311/REDBLACK/RedBlack.html`