Assignment 7
Testing with QuickCheck

Model-Based Testing
DIT848/GU and TDA260/Chalmers

May, 2012

1 Introduction

The goal of this assignment is for you to learn how to use QuickCheck generators to generate more precise test cases. In this assignment you will be given an implementation of red-black binary search trees (a type of balanced tree) in Haskell, and you will be required to write and test QuickCheck properties that they should satisfy.

2 Submitting your work

If you want to have feedback on your assignment, check with Pablo Buiras (`buiras #@# chalmers.se`) on how (and when) to submit. If you want to submit, please attach a .zip or .tar.gz archive, containing your source code and a .txt or .pdf file describing your answers. Please include the assignment number and your (last) name in the file name, as in the following example: `buiras_assignment07.zip`.

3 Testing Red-Black trees

Red-black trees are binary search trees that enforce a *balance* property, which ensures that elements are appropriately distributed among the branches. The goal of this "balancing" is to make binary search in the tree perform better (close to logarithmic time in the size of the tree), regardless of the order in which the elements are inserted into the tree.

Module `RBBST` includes an implementation of red-black binary search trees in Haskell. The interface is similar to the one of the binary search trees from the previous assignment. We introduce type `Tree` such that `Tree a` is a tree of `a`s, along with the following operations:

```
insert :: Ord a => a -> Tree a -> Tree a
member :: Ord a => a -> Tree a -> Bool
merge :: Ord a => Tree a -> Tree a -> Tree a
isEmpty :: Tree a -> Bool
empty :: Tree a
```

As usual, `insert` takes an element of type `a` and a RB tree, and returns a new RB tree with the extra element in the correct position. Repeated elements are stored only once. Moreover, `member` checks that a given element is present in the tree. The function `merge` takes two RB trees and merges them together into one RB tree. The function `isEmpty` checks whether the argument tree is empty, while the constant `empty` represents the empty tree.

The `Tree` data type includes a *colour* for every node, which can be either `Red` or `Black`. Colours will be used to enforce certain invariants that will guarantee that the tree is sufficiently balanced.

The red-black property. Given a tree t, we say that it has the *red-black property* if either of these conditions hold:

- t is empty;
- if t is nonempty, then:
  - the root of the tree is black;
  - the *immediate* children of a red node cannot be red; and
  - all paths from the root to a leaf have the same number of black nodes.

The function `insert` should preserve both the BST property (from Assignment 6) and also the red-black property as an invariant. The function `member` should run faster now (at the expense of a slower `insert` function) because of the red-black property.

A useful consequence of the red-black property is that the length of the longest path from the root to a leaf is at most *double* the length of the shortest path form the root to a leaf. This means the tree is not necessarily perfectly balanced, but it is still good enough to speed up `member`.

Examples. Here are some examples of the trees that should be generated by `insert`:

- `insert 3 (insert 1 (insert 4 empty)) ==`

  `Node (Node Empty (1,Black) Empty) (3,Black) (Node Empty (4,Black) Empty)`
- `insert 1 (insert 4 (insert 3 empty)) ==`

  `Node (Node Empty (1,Red) Empty) (3,Black) (Node Empty (4,Red) Empty)`
- `insert 3 (insert 3 empty) == Node Empty (3,Black) Empty`

1. Write a QuickCheck property that checks the invariance of the red-black property in this implementation. **NB.** In order to do this, you will have to write a *generator* for `Tree`, that simply inserts some number of random elements into the empty tree.
2. Red-black trees have the following property: let $N = 2^k - 1$ for some integer $k > 0$, then the tree generated by inserting all integers from 1 to N in sequence is a *complete* tree, *i.e.* all of its leaves are at the same level. For example, if $N = 3$, we have that `insert 1 (insert 2 (insert 3 empty))` is complete.

   Use a generator to write a QuickCheck property representing this proposition, checking it for relatively small values of N so as to make it tractable.
3. Write **four** more properties that you think should hold for red-black trees. Try to avoid repeating the same properties you checked for BSTs in the previous assignment. Think about suitable properties to test the new function, `merge`. If possible, use generators to avoid evaluating unnecessary test cases.
4. Test all these properties using QuickCheck. Should any test fail, you are expected to find and report the error, and tell whether it is in the implementation itself or in your properties.