CSCI 2041: Advanced Programming Principles
Fall 2017, University of Minnesota
Homework 8

Posted: Dec 9, 2017
Due: Dec 13, 2017 by 23:59

There are only two problems in this homework, one of which is also part of the work you will do in the last lab. As discussed in class, the homework has been kept short because of the limited time you have for doing it. Doing the homework is important, though, insofar as it is exercises that help you really understand the things we discuss in class.

Submission Protocol

You should submit a file called `hw8.ml` that contains your solutions to the problems in this homework. The solutions for the different parts of Problem 1 should appear in a comment. When doing Problem 2, make sure to adhere to the names and types provided for the functions that you are required to write. *Mark the solution to each problem and each part clearly, through a line next to the solution you are providing; in the case of Problem 2, these lines should appear within OCaml comments.*

The files that you are to submit must be **pushed** to your private git repository in the github structure for the course, i.e., umn-csci-2041S15/repo-abcde001. As usualy, they must be placed at the top-level in this directory.

Before you start working on the homework, make sure to read the comments on the protocol for homeworks and the issues we consider when grading. Note in particular that you must pay attention to the structure of your programs, you must not have excessively long lines and you must use indentation to make your program text readable. Also avoid tabs: they show up differently under different editors and can make your code look ugly, something that *does matter* when we grade your work.

Finally, before submitting the final version of your homework, you should ensure it is well-formed and type-correct, i.e., contain no syntax or type errors. You may check this by running the following commands in the same directory as your homework submission on a cselabs machine:

```
ocamlbuild hw8.byte || echo "\n\n---INCORRECT SUBMISSION---\n\n"
```

If you see:

```
---INCORRECT SUBMISSION---
```

when you run this command, it means that the corresponding homework submission is not well-formed or well-typed. You should correct this problem before submission.

Problem 1 ((2 + 1 + 2) + (2 + 2) + (2 + 2 + 1 + 1) points)

The following code defines an OCaml function `mergesort` that sorts a list of objects on which the < relation is defined:

```
let rec partition l =
  match l with
    | [] -> ([],[])
    | [_] -> (l,[])
    | (h1::h2::l') ->
        let (t1,t2) = partition l' in (h1::t1,h2::t2)

let rec merge l1 l2 =
  match (l1,l2) with
  | (([],l) | (l,[])) -> l
  | (h1::t1,h2::t2) ->
        if (h1 < h2) then h1 :: merge t1 l2
        else h2 :: merge l1 t2

let rec mergesort l =
  match l with
  | ([] | [_]) -> l
  | _ -> let (l1,l2) = partition l
         in let l1' = mergesort l1
         in let l2' = mergesort l2
         in merge l1' l2'
```

For simplicity, assume that the length of the list input to `mergesort` is always a power of 2. The task in this problem is to eventually show that the time taken by `mergesort` is $O(n * log_2(n))$ where n is the length of the list to be sorted. Recall that what this means is that we have to show that there is some constant c such that, for sufficiently large n, the time taken by `mergesort` is no worse than $c * n * log_2(n)$.

1. This part requires you to show that the (worst-case) running time for `partition` is both $\Omega(n)$ and $O(n)$ where n is the length of the input list. Specifically, do the following:

    a. Provide a recurrence relation for $T(n)$, the (worst-case) running time for `partition` expressed as a function of the length of the input list.

    b. Guess a closed form solution to $T(n)$; a closed form solution would be one that is not recursive, i.e. does not use T again in it, and that depends only on n and some constants. Clearly, in this case, $T(n)$ should depend linearly on n.

    c. Show that your closed form solution is correct using induction on n.

2. This part requires you to show that the worst-case running time for `merge` is $O(n1 + n2)$, where $n1$ and $n2$ are, respectively the lengths of the input lists. Specifically, do the following:

    a. Explain what will be the worst case for `merge`.

    b. Explain intuitively why, for sufficiently large $n1 + n2$, the time needed by `merge` will not exceed $c * (n1 + n2)$ for some fixed constant c.

3. This part concludes the analysis by showing that the worst-case running time for `mergesort` is $O(n * log_2(n))$ where n is the length of the input list.

a. Write down a recurrence relation that describes an upper bound for *S(n)*, the worst-case running time for `mergesort`, expressed as a function of the length of its input list. As indicated at the beginning of the problem, assume that the length of the list is either 0 or a power of 2. To be more specific, rather than expressing a equality, your recurrence relation should take the form

```
S(n) ≤ ...
```

where the ... part may use S(n') for some n' that is related to n. Use the results/observations of the previous parts but be sure to justify the relation you write down; *the justification will carry the most points*.

b. Guess a closed form expression that provides an upper bound for *S(n)*. Once again, what is meant by "closed form" is that the expression can use *n* and known functions of *n*, but it must not use *S* recursively. Note that your guess must be of a form that enables you to complete the last part of the problem; guesses that do not have this character, and that you cannot prove to be correct as needs to be done in the next part, will not get credit.

c. Verify your guess using induction on *n*.

d. Use the result of the previous two sub-parts to conclude that the worst-case running time for `mergesort` is $O(n * log_2(n))$.

Problem 2 (5 + 1 + 1 points)

This problem is one that you will also be solving in the lab; it is okay to re-use the solution you develop there.

We provided the following type declaration for red-black trees in class:

```
type color = R | B

type 'a rbtree =
   Empty
 | Node of color * 'a * 'a rbtree * 'a rbtree
```

Note that this type declaration does not include a colour with leaf nodes which are implicitly assumed to be coloured black. We also observed that "good" red-black trees needed to satisfy some additional properties:

1. if a node is coloured red, its children must be black, and
2. every path from a given node to its leaves must have the same number of black nodes.

However, these properties are not enforced by the type declarations. In this problem, we will define functions that will check that a given object of the type `('a rbtree)` in fact satisfies these properties.

1. Define a function `is_RBTree_aux : 'a rbtree -> int * bool` that returns the value `(n,true)` if the input satisfies the required properties and `n` is then the black height of the tree and the value `(n,false)` if it violates one or the other of the required properties; in the latter case, `n` can be chosen arbitrarily. As examples that show the behaviour desired of this function, consider the following:

```
# is_RBTree_aux Empty;;
- : int * bool = (0, true)
# is_RBTree_aux (Node (R, 5, Node(B, 3, Node(R, 1, Empty, Empty),
                                          Empty),
                                  Node(B, 7, Empty, Empty)));;
```

```
- : int * bool = (1, true)
# is_RBTree_aux (Node (B, 5, Node(R, 3, Node(R, 1, Empty, Empty),
                                         Empty),
                                  Node(B, 7, Empty, Empty)));;
- : int * bool = (1, false)
# is_RBTree_aux (Node (B, 5, Node(B, 3, Node(R, 1, Empty, Empty),
                                      Empty),
                                  Node(B, 7, Empty, Empty)));;
- : int * bool = (2, true)
# is_RBTree_aux (Node (R, 5, Node(B, 3, Node(B, 1, Empty, Empty),
                                         Empty),
                                  Node(B, 7, Empty, Empty)));;
- : int * bool = (2, false)
# is_RBTree_aux (Node (B, 5, Node(R, 3, Node(B, 1, Empty, Empty),
                                         Empty),
                                  Node(B, 7, Empty, Empty)));;
- : int * bool = (2, false)
# is_RBTree_aux (Node (B, 5, Node(R, 3, Node(B, 1, Empty, Empty),
                                      Node(B, 4, Empty,
                                                 Node(R, 5, Empty, Empty))),
                                  Node(B, 7, Empty, Empty)));;
- : int * bool = (2, true)
```

One thing to note in these examples: the black height of a (good) red-black tree does not count the leaf node that terminates a path, something that I missed discussing specifically in class but that was clarified in a Forum posting.

2. Using is_RBTree_aux, define the function `is_RBTree : 'a rbtree -> bool` that returns `true` just in the case that its input satisfies the properties required of a red-black tree. Shown below are the results of invoking this function on the trees considered above:

```
# is_RBTree Empty;;
- : bool = true
# is_RBTree (Node (R, 5, Node(B, 3, Node(R, 1, Empty, Empty),
                                         Empty),
                                  Node(B, 7, Empty, Empty)));;
- : bool = true
# is_RBTree (Node (B, 5, Node(R, 3, Node(R, 1, Empty, Empty),
                                  Empty),
                           Node(B, 7, Empty, Empty)));;
- : bool = false
# is_RBTree (Node (B, 5, Node(B, 3, Node(R, 1, Empty, Empty),
                                           Empty),
                                  Node(B, 7, Empty, Empty)));;
- : bool = true
# is_RBTree (Node (B, 5, Node(B, 3, Node(R, 1, Empty, Empty),
                                  Empty),
                           Node(B, 7, Empty, Empty)));;
- : bool = true
# is_RBTree (Node (R, 5, Node(B, 3, Node(B, 1, Empty, Empty),
                                  Empty),
                           Node(B, 7, Empty, Empty)));;
- : bool = false
# is_RBTree (Node (B, 5, Node(R, 3, Node(B, 1, Empty, Empty),
                                  Empty),
                           Node(B, 7, Empty, Empty)));;
- : bool = false
# is_RBTree (Node (B, 5, Node(R, 3, Node(B, 1, Empty, Empty),
                                  Node(B, 4, Empty,
                                             Node(R, 5, Empty, Empty))),
                           Node(B, 7, Empty, Empty)));;
- : bool = true
#
```

3. Using `is_RBTree_aux` (or otherwise) define the function `bh_RBTree : 'a rbtree -> int option` that returns a value of the form `(Some n)` if its input is a red-black tree with black height `n` and `None` otherwise. The expected results of using this function on the examples considered earlier are shown below:

```
# bh_RBTree Empty;;
- : int option = Some 0
# bh_RBTree (Node (R, 5, Node(B, 3, Node(R, 1, Empty, Empty),
                                    Empty),
                         Node(B, 7, Empty, Empty)));;
- : int option = Some 1
# bh_RBTree (Node (B, 5, Node(R, 3, Node(R, 1, Empty, Empty),
                                    Empty),
                         Node(B, 7, Empty, Empty)));;
- : int option = None
# bh_RBTree (Node (B, 5, Node(B, 3, Node(R, 1, Empty, Empty),
                                    Empty),
                         Node(B, 7, Empty, Empty)));;
- : int option = Some 2
# bh_RBTree (Node (R, 5, Node(B, 3, Node(B, 1, Empty, Empty),
                                    Empty),
                         Node(B, 7, Empty, Empty)));;
- : int option = None
# bh_RBTree (Node (B, 5, Node(R, 3, Node(B, 1, Empty, Empty),
                                    Empty),
                         Node(B, 7, Empty, Empty)));;
- : int option = None
# bh_RBTree (Node (B, 5, Node(R, 3, Node(B, 1, Empty, Empty),
                                    Node(B, 4, Empty,
                                               Node(R, 5, Empty, Empty))),
                         Node(B, 7, Empty, Empty)));;
- : int option = Some 2
#
```
