The Least Common Ancestor (Racket+Haskell - 7+7 Points)

June 5, 2023

Suppose we have a binary tree t. For any two nodes x, y in the tree t, the *least common ancestor* of x and y is defined as the node z satisfying the following two conditions:

1. x and y are descendants of z,
2. if there is a node z' having x and y as descendants, then z is descendant of z'.

To find the least common ancestor of two nodes x and y in a tree t, we follow the steps below:

1. find the path p_x from the root r of t to x (i.e., a list of nodes starting in r and ending in x),
2. find the path p_y from r to y,
3. consider the common prefix of p_x and p_y, the last node in the common prefix is the least common ancestor.

Consider, for example, the binary tree depicted in Figure 1. The least common ancestor of 3 and 5 is 2. Indeed, the path from the root 1 to 3 is $1, 2, 3$. The path from 1 to 5 is $1, 2, 4, 5$. Their common prefix is $1, 2$ whose last element is 2.

Similarly, the least common ancestor of 5 and 8 is 1. The least common ancestor of 7 and 7 is 7.

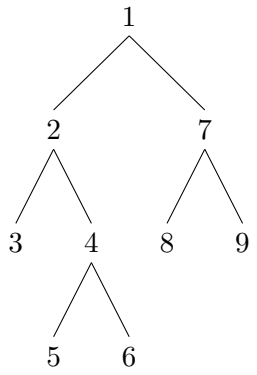

Figure 1: A binary tree

1 Task 3 - Racket

In **Racket**, implement a function `(common-ancestor x y tree)` that takes two nodes `x`, `y` and a binary tree `tree`, and returns the least common-ancestor of `x` and `y` in `tree`. If `x` or `y` does not belong to `tree`, the function returns `#f`.

To represent binary trees in Racket, use the following structures:

```
(struct node (val left right) #:transparent)
(struct leaf (val) #:transparent)
```

Thus the leaves (nodes without children) are represented as, for instance, `(leaf 6)`. The nodes with children are represented as, for instance, `(node 1 (leaf 2) (leaf 3))`.

To implement the function `common-ancestor`, implement first a function `(find-path x tree)` that finds a path from the root of `tree` to `x`. For example,

```
(define tree (node 1 (node 2 (leaf 5) (leaf 6))
                     (node 3 (leaf 4) (leaf 7))))

> (find-path 7 tree)
'(1 3 7)
```

Your file should be called `task3.rkt` and should export the `find-path` and `common-ancestor` functions and the structures `node` and `leaf`.

```
#lang racket
(provide find-path
         common-ancestor
         (struct-out node)
         (struct-out leaf))

(struct node (val kids) #:transparent)
(struct leaf (val) #:transparent)

(define (find-path x tree)
  ; Implement me!
  )

(define (common-ancestor x y tree)
  ; Implement me!
  )
```

Example

```
(define tree2 (node 1 (node 2 (leaf 3)
                              (node 4 (leaf 5)
                                      (leaf 6)))
                      (node 7 (leaf 8)
                              (leaf 9))))

> (common-ancestor 3 5 tree2)
2

> (common-ancestor 3 15 tree2)
#f
```

Hints

To find the common prefix of two lists, use the function `(take-common-prefix lst1 lst2)`.

2 Task 4 - Haskell

In **Haskell**, implement a function `commonAncestor :: Eq a => a -> a -> Tree a -> Maybe a` that takes two nodes and a binary tree, and returns the least common ancestor of these two nodes. If it does not exist, the function returns `Nothing`.

To represent binary trees in Haskell, use the following data type:

```
data Tree a = Leaf a
            | Node a (Tree a) (Tree a) deriving (Eq,Show)
```

Thus the leaves (nodes without children) are represented as, for instance, `Leaf 6`. The nodes with children are represented as, for instance, `Node 1 (Leaf 2) (Leaf 3)`.

To implement the function `commonAncestor`, implement first a function `findPath :: Eq a => a -> Tree a -> [a]` that finds for a given node and a binary tree the path from the root to that node. For example,

```
tree = Node 1 (Node 2 (Leaf 5) (Leaf 6)) (Node 3 (Leaf 4) (Leaf 7))

> findPath 7 tree
[1,3,7]
```

Your file should be called `Task4.hs` and should export the `commonAncestor`, `findPath` functions and the type `Tree`.

```
module Task4 (findPath, commonAncestor, Tree(..)) where

data Tree a = Leaf a
            | Node a (Tree a) (Tree a) deriving (Eq,Show)

findPath :: Eq a => a -> Tree a -> [a]
-- implement me!

commonAncestor :: Eq a => a -> a -> Tree a -> Maybe a
-- implement me!
```

Example

```
tree2 = Node 1 (Node 2 (Leaf 3)
                       (Node 4 (Leaf 5)
                               (Leaf 6)))
               (Node 7 (Leaf 8)
                       (Leaf 9))

> commonAncestor 3 5 tree2
Just 2

> commonAncestor 3 15 tree2
Nothing
```