Problem Description

Suppose you are the boss of a company, you want to throw a party for your employees. Your company is modelled as a binary tree and each employee in the company is represented as a node in the tree with you (the boss) at the top. You get to assign each employee a non-negative integer that represents their fun score where a higher score corresponds to an employee you want to have at the party because you think they would make it more fun. However, you also realize that no employee will have fun at the party if their direct boss is also there. So, you cannot invite two employees that are directly connected in the binary tree (i.e. if you invite an employee you cannot invite any of their direct subordinates (child nodes) or if you invite an employee, you cannot also invite their direct boss (parent node)). In this exercise, you will implement four different variations of an algorithm that gives us the most fun party.

The first algorithm is the naive solution to this problem. In this algorithm we will model the company as type tree = Empty | Node of int * tree * tree where in a node, the int represents the fun value of the employee and the two trees are the left and right subtrees of this node. The algorithm should be implemented using three mutually recursive functions (1) party: tree -> int which given a tree returns an integer of the maximum possible fun value for a party from that tree (2) party_in: tree -> int which given a tree returns an integer of the maximum possible fun value for a party from that tree where the root node is invited (3) party_out: tree -> int which given a tree returns an integer of the maximum possible fun value for a party from that tree where the root note is not invited. In any of these functions, if the given tree is empty, return 0.

The second algorithm is the solution to this problem that optimizes the runtime using memoization directly inside the tree. The company will now be modelled as a type tree = Empty | Node of int * string * tree * tree * ((int*string list) option) ref where in a node, the int represents the fun value of the employee, the string is the name of the employee, the two trees are the left and right subtree of the node, and the reference cell stores an option type of a tuple of the optimal fun value of a party with the list of names that make up this maximal fun value for the subtree starting at that node. Again, this algorithm should be implemented using three mutually recursive functions (1) party: tree -> int * string list which given a tree returns a tuple of the maximum possible fun value for a party from that tree and the list of names for that maximally fun party (2) party_in: tree -> int * string list which given a tree returns a tuple of the maximum possible fun value for a party from that tree and the list of names for that maximally fun party where the root node is invited (3) party_out: tree -> int * string list which given a tree returns a tuple of the maximum possible fun value for a party from that tree and the list of names for that maximally fun party where the root note is not invited. In any of these functions, if the given tree is empty, return (0, []).

Then, working off the memoized algorithm (should still memoize your results for this algorithm) and using the same tree structure, implement each of these functions tail-recursively where (1) party: tree -> (int * string list -> ‘a) -> ‘a (2) party_in: tree -> (int * string list -> ‘a) -> ‘a (2) party_out: tree -> (int * string list -> ‘a) -> ‘a. In any of these functions, if the given tree is empty, return (0, []).

Lastly, implement an algorithm that can be used for a company of a generalized structure (not just a binary tree) where type tree = Empty | Node of int * string * tree list * ((int * string list) option) ref. Again, (1) party: tree -> (int * string list -> ‘a) -> ‘a (2) party_in: tree -> (int * string list -> ‘a) -> ‘a (2) party_out: tree -> (int * string list -> ‘a) -> ‘a. In any of these functions, if the given tree is empty, return (0, []).