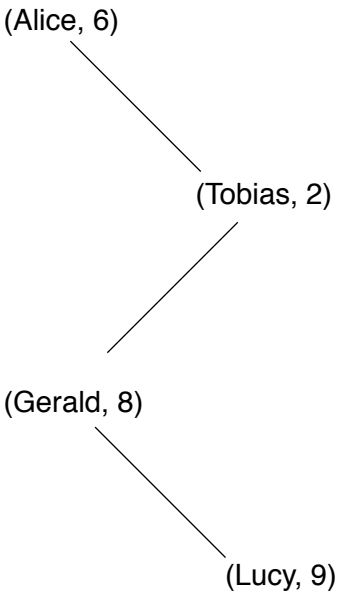
Supervision work
- 27.10.2015 -

6.3

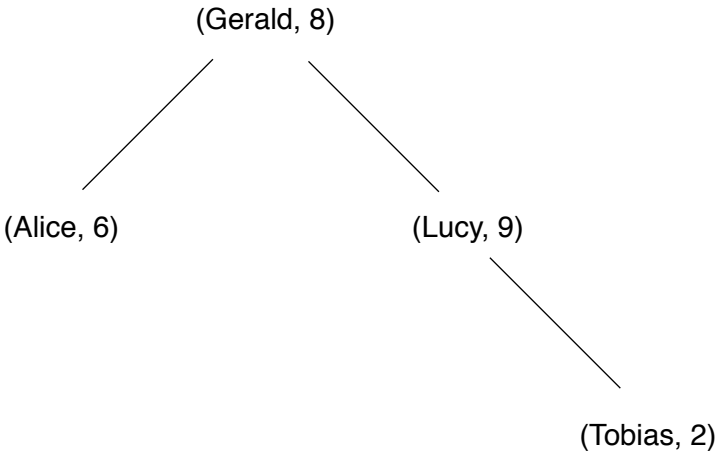
The function ftree(k,n) returns a binary tree of integers, with the root k and depth n. So, for ftree(1,n) will return a binary tree with root 1 and depth n. In other words, it will insert the numbers from 1 to 2^n -1.

7.1

(i) If they are inserted in this order : (Alice, 6), (Tobias, 2), (Gerald, 8), (Lucy, 9)



(ii) If they are inserted in this order : (Gerald, 8), (Alice, 6), (Lucy, 9), (Tobias, 2)



The results are different because the order the data is inserted in the tree matters. Although both trees are binary, their depth is different and this is an important aspect if we want to have an efficient binary search tree. Just like in the quick sort, if we choose the wrong root(or pivot) we end up with a linear complexity of searching, instead of a logarithmic one, which we want to accomplish.

7.3

The insertion function is a nested one, so, even if we find a collision and we raise the exception, it would be difficult just to return the value previously stored in the dictionary, because of the multiple recursive calls. In order to do so, we would need another checking variable that becomes false when we find a collision and doesn't let the function return other values.

7.4

In order to delete an entry from a binary search tree, we also need to update the remaining tree so that it can still keep its properties. A possible algorithm to do that would be to store all the elements from the tree except the one we want to delete in a list and to create a new tree using the remaining elements. If we consider the initial tree to be balanced, then a good method to store the elements in the list would be the preorder, which will give us an O(N) complexity for this part of the algorithm (where N is the number of entries). This way, inserting the other entries into the dictionary will be done in O(N*logN). Therefore, the final complexity of the algorithm would be O(N*logN) (even though, if the initial tree is not balanced, the complexity can go as high as $O(N^2)$).

7.6

Functions *preorder, inorder* and *postorder* initially go through all the N entries from the tree and call themselves twice, using "@" function to create the final list. So, it may seem that they are linear. But on the way back from the recursion, they have to append one list to the other and this operation also takes linear time. In conclusion, the overall time complexity of these algorithms is $O(N^2)$.

7.7

The functions *preord, inord* and *posord* use the same principle as *postorder, inorder* and *postorder,* but in stead of using append to create the list, they use the "::" function. So, instead of appending two lists, they just put a new element at the beginning of the new list, action that can be done in constant time (O(1)). Therefore, the time complexity of these functions is O(N).

8.1

The function `fun sw f x y = f y x;` returns the result of the curried function `f y x.`