Haskell exercise: Balanced Binary Search Trees

Department of Mathematics and Computer Science
University of Southern Denmark

September 29, 2017

An ordered dictionary is an abstract datatype storing elements of type (k, x), where the key k is from some ordered set (e.g. the real numbers $\mathbb{R}$) and the data x is from some arbitrary set.

The data structure we will use to implement dictionary is a balanced binary search tree of the kind *AVL-tree* (which is in fact, the first type of balanced search trees, invented in 1962 by Adelson-Velsky and Landis).

The balancing part of the exercise is postponed to the last few exercises, so for now, you only need to think of the tree as an ordinary binary search tree, without any balance guarantees.

1. We introduce the type definitions

   type *Height* = *Int*
   type *Balance* = *Int*
   data *Dict k a* = *Node Height* (k, a) (*Dict k a*) (*Dict k a*) | *Nil*

   to represent our tree.

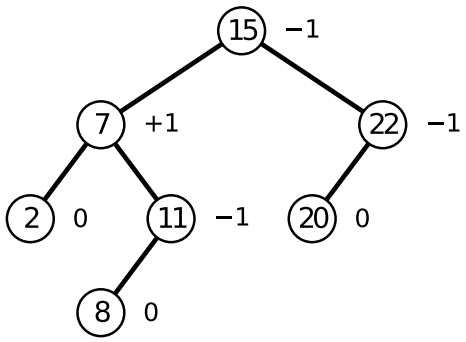


Figure 1: A binary search tree with a key written in each node. The balance of a node (defined in exercise 8) is written next to the node.

The data constructor *Nil* corresponds to the empty tree.

The data constructor *Node* corresponds to a node in the tree. When pattern matching on a *Node*, one gets access to

- the *height* of the node (stored as an *Int* value)
- a tuple of type (k, a), which associates a key of type k with a value of type a.
- the left subtree
- the right subtree

Implement the functions:

$$\begin{aligned} &\mathit{null} :: \mathit{Dict}\ k\ a \to \mathit{Bool} \\ &\mathit{empty} :: \mathit{Dict}\ k\ a \\ &\mathit{singleton} :: (k, a) \to \mathit{Dict}\ k\ a \\ &\mathit{height} :: \mathit{Dict}\ k\ a \to \mathit{Height} \end{aligned}$$

null returns *True* if the tree is empty, and *False* otherwise.
empty returns an empty tree.
singleton returns a tree with a single element. The height of a singleton node is 1.
height returns the height of a node. It is 0 for *Nil* and the height value stored in the node otherwise.

2. Implement the function

$$\mathit{node} :: (k, a) \to \mathit{Dict}\ k\ a \to \mathit{Dict}\ k\ a \to \mathit{Dict}\ k\ a$$

which is a *smart constructor* for a node. For a key, value and two subtrees, it constructs a *Node*, where the height is calculated from the height of the subtrees (mathematically, the height of the new node $h(t)$ is given by $h(t) = 1 + \max\{h(t_1), h(t_2)\}$, where $h(t_1), h(t_2)$ are the heights of the subtrees).

3. Implement the function

$$\mathit{insert} :: \mathit{Ord}\ k \Rightarrow (k, a) \to \mathit{Dict}\ k\ a \to \mathit{Dict}\ k\ a$$

which inserts the node at the correct position of the tree (respecting the binary search tree property). If there is already a node in the tree with the key, the node should be replaced. Use the smart constructor *node* when you insert.

4. Implement the functions

$$lookup :: Ord\ k \Rightarrow k \to Dict\ k\ a \to Maybe\ a$$
$$rangeLookup :: Ord\ k \Rightarrow k \to k \to Dict\ k\ a \to [(k, a)]$$

lookup k d returns a value, if there is a node in the tree d with the key k.
lookupRange k1 k2 d returns a list of all key-value pairs, which have keys k satisfying $k_1 \leqslant k \leqslant k_2$

5. Write a function

$$fromList :: Ord\ k \Rightarrow [(k, a)] \to Dict\ k\ a$$

which loads a list of key/value-pairs into the tree. Consider using either *foldl* or *foldr*.

6. Write a function

$$flatten :: Dict\ k\ a \to [(k, a)]$$

which outputs a list of all key-value pairs in the tree, ordered by *key*.

7. Implement the functions

$$leftmost :: Dict\ k\ a \to Maybe\ (k, a)$$
$$delete :: Ord\ k \Rightarrow k \to Dict\ k\ a \to Dict\ k\ a$$

leftmost finds the minimum value of a tree. If the tree is empty it returns *Nothing*. *delete* removes a node from the tree. The following cases should be handled

- If a node with the given key k does not exist in the tree, return the tree unchanged
- If the node with the given key k only has one child, replace the node with its child in the tree.
- If the node with the given key k node has two children t_1 (left) and t_2 (right), use *leftmost* to find the minimum element (k', v') of the right subtree t_2, and return a new node with value (k', v') and subtrees t_1 and t_2' where t_2' is the right subtree with k' removed.

Use the smart constructor *node* when creating new nodes in the deletion process.

8. Implement the functions

   $bal :: Dict\ k\ a \to Balance$
   $hasValidBalance :: Dict\ k\ a \to Bool$

   where the balance *bal* is defined to be the $h(t_2) - h(t_1)$ where $h(t_1)$ is the height of the left subtree and $h(t_2)$ is the height of the right subtree. The empty tree has zero balance.

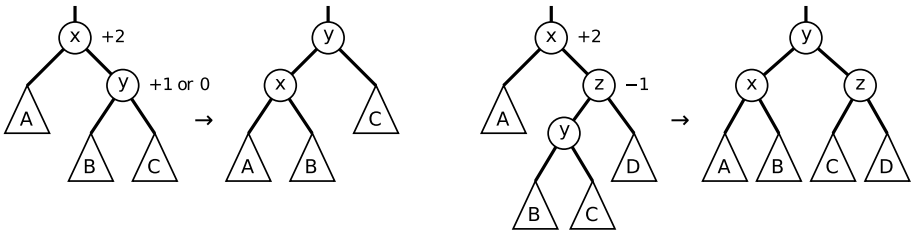
   hasValidBalance returns true, if the balance b satisfies $abs(b) \leqslant 1$ in every node (*abs* is the absolute value).

9. We are now ready to define rotation operations to ensure that the tree stays balanced, and gets $O(\log(n))$ time complexity for insertion and deletion.

   $rotate :: Ord\ k \Rightarrow Dict\ k\ a \to Dict\ k\ a$

   rotate is to be called everywhere a node is created elsewhere in the program.

   After an insertion or deletion, the heights of subtrees rooted at nodes on the path from the root to the node which was deleted have possibly changed by one. Hence, nodes v in this path may now have $abs\ (bal\ (v)) = 2$, which will violate the balance criterion. Nodes outside this path clearly change neither height nor balance. The balance criterion is restored in a bottom-up fashion along this path. For an unbalanced node v on this path, one of the two transformations (denoted a single rotation and a double rotation) shown below is performed.



   The choice between the two transformations depends on the balance of the tallest child of the unbalanced node v, as indicated on the figure. **Symmetric cases (reflected in a vertical line) of the transformations exist, in which all balance values shown have changed sign - so there are 4 cases in total to be handled!**

   It can be shown that these transformations will restore the balance at v, given that all unbalance below it has been handled. After v, the process continues with the parent of v, etc., until the root is reached.