Homework 07

2016-11-22

此次请提交如下的结构：

```
YourName_2016123456.zip
├── avltree/
|    └── Solution.hs
```

无需定义 `solution` 函数。

这次的作业，简单地说，就是用 Haskell 实现 AVL 树这个数据结构。

我猜你很熟悉二叉搜索树，可能也在命令式语言中实现过 AVL 树或其他平衡二叉树。用 Haskell 实现起来，会有一些新奇的体验。包括但不限于 `Maybe` 类型、类型参数的语法、类型类（type class）的语法、 `Functor` 类型类、 `Foldable` 类型类。

`AvlTree.hs` 是我们提供的一个模板。它既规定了 AvlTree 的基本结构，又包含一部分指导性的代码来给上述“新奇体验”带路。要完全明白 `AvlTree.hs` ，你需要学习网络学堂的 `07.Algebraic_Data_Type` 课件。 `TestAvlTree.hs` 里有些测试数据。

这次作业中，你需要做的是：

1. 为 `AvlTree a` 类型实现 `SortedBinaryTree` 所声明的 `search` 、 `insert` 和 `remove` 函数，从而使 `AvlTree a` 成为 `SortedBinaryTree` 类型类的实例。
2. 为 `AvlTree a` 类型实现 `fmap` 函数，从而使它成为 `Functor` 类型类的实例。为 `AvlTree a` 类型实现 `foldr` 函数，从而使它成为 `Foldable` 类型类的实例。

分数构成如下：

- search 20%
- insert 30%
- remove 30%
- fmap 10%
- foldr 10%

不出意外，你可以在 `AvlTree.hs` 找到“该去哪儿实现这些函数”。不出意外， `AvlTree.hs` 和 `TestAvlTree.hs` 可以直接通过 `ghci` 的编译检查。如果出了意外，请联系助教：）

对 AvlTree 的约定

在任意次 `search` 、 `insert` 或 `remove` 操作之后，作为一个 `AvlTree` 的 `t` 应一直满足的性质（不变式）：

1. 保存高度信息： `t` 内任意结点 `(Node _ h l r)` ， `h` 应保存该结点（或叫子树）的高度。高度定义为 `1 + max [height l, height r]` ，其中

   ```
   height Nil = -1
   height (Node _ h _ _) = h
   ```

2. 键不重复。即：t 内任意两个不同结点 `(Node k _ _ _)` 和 `(Node k' _ _ _)` , 应有 `k /= k'` .

3. 有序。即：t 内任意结点 `(Node k _ l r)` ，应有性质：若左子树 `l` 不为 `Nil` ，则 `key l < Just k` ; 若右子树 `r` 不为 `Nil` ，则 `key r > Just k` . 其中

   ```
   key Nil = Nothing
   key (Node k _ _ _) = Just k
   ```

   （由于标准库定义 `Maybe a` 类型的时候 `deriving Ord` , 故对任意 `Ord a, x::a, y::a` , 有 `Just x <= Just y` 当且仅当 `x <= y` .）

4. 平衡。即： `t` 内任意结点 `(Node _ _ l r)` ，应有性质： `(abs $ height l - height r) < 2` .

对 search、insert 和 remove 的约定

类型

`insert` , `search` 和 `remove` 三个函数的类型遵循 `SortedBinaryTree` 中的声明。

解释一下：比如，SortedBinaryTree 类型类声明了函数 insert :: (Ord a) => a -> t a -> t a，那么对于 AvlTree a 类型，就有 insert :: (Ord a) => a -> AvlTree a -> AvlTree a. 如果继续填入类型参数，AvlTree Int 就有 insert :: Int -> AvlTree Int -> AvlTree Int. 但是总之，你不需要在 SortedBinaryTree 类型类的定义之外再次声明 insert 等函数的类型签名。希望我这几句解释没有带来更多的混乱……

功能

search

`search k t` 应返回 `t` 中的以 `k` 所在结点为根的子树，并用 `Just` 构造器包裹起来。

如果 `t` 不含键 `k`， `search k t` 应返回 `Nothing` 。

可能你需要 `Maybe a -> a` 的函数，用于从 `Just x` 中取出 `x` 。在 `https://www.haskell.org/hoogle/` 上找找。

insert

讲道理， `insert k t` 应该在 `t` 中插入键为 `k` 的结点。但是，由于 Haskell 的数据都 immutable，没有办法真的“插入”，只能新构造一棵树。所以，

`insert k t` 返回一棵比 `t` 恰好多了键 `k` 的树（的根结点）。

如果 `t` 已经含有键 `k`，`insert k t` 返回恰好含有 `t` 含有的所有键的树。当然返回 `t` 最方便了，但不是一定要这样做。

注意维护上述几个不变式：保存高度信息，键不重复，有序，平衡。

remove

讲道理， `remove k t` 应该从 `t` 中移除键为 `k` 的结点。当然，immutable，没法真的移除。所以，

`remove k t` 返回一棵比 `t` 恰好少了键 `k` 的树（的根结点）。

如果 `t` 不含键 `k`，`remove k t` 返回恰好含有 `t` 含有的所有键的树。当然返回 `t` 最方便了，但不是一定要这样做。

注意维护上述几个不变式。