COMP50001: Algorithm Design & Analysis

Sheet 5 (Week 6)

Exercise 5.1

When implementing an AVL tree, at each node it is possible to store just the difference in height between the two children. According to the invariants on an AVL tree, the bias can either be -1, 0, or 1.

Implement $insert :: Ord\ a \Rightarrow a \to DTree\ a \to DTree\ a$ and $delete :: Ord\ a \Rightarrow a \to DTree\ a \to DTree\ a$ on *DTree*, an AVL tree which stores differences rather than heights.

```
data HTree a = HTip
             | HNode Int (HTree a) a (HTree a)
data DTree a = DTip
             | DNode Diff (DTree a) a (DTree a)
data Diff = MinusOne | Zero | PlusOne
```

Exercise 5.2

Write a function $fromOrdList :: [a] \to HTree\ a$ which builds an AVL tree from a sorted list in worst case linear time.

```
fromOrdList :: [a] → HTree a
fromOrdList xs = fst (go (length xs) xs)
  where
    go :: Int → [a] → (HTree a, [a])
    go = ...
```

Exercise 5.3

The implementation of *member* presented in lectures uses $2d$ comparisons in the worst case, where d is the depth of the tree. Reimplement *member* to perform at most $d + 1$ comparisons, using only $(\leqslant)$ to compare elements.

```
member :: Ord a ⇒ a → HTree a → Bool
member x HTip = False
member x (HNode _ lte p gt)
  | x < p      = member x lte
  | x ≡ p      = True
  | otherwise = member x gt
```

A possible implementation strategy is to carry the most recent element that *could* be equal to the one being searched for with you; when you hit the bottom of the tree the you can test for equality.

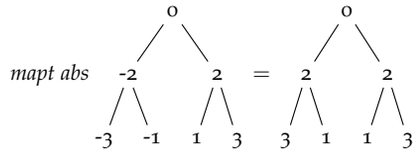
Exercise 5.4

We can define a kind of *map* on search trees as follows:

```
mapt :: (a → b) → HTree a → HTree b
mapt f HTip                = HTip
mapt f (HNode b ls x rs) = HNode b (mapt f ls) (f x) (mapt f rs)
```

Describe the minimal condition to which f must adhere in order for the output of *mapt f* to be a valid search tree, given valid input.

Define a function $mapt' :: Ord\ b \Rightarrow (a \to b) \to Tree\ a \to Tree\ b$ which does not require f to adhere to your condition in order to produce a valid tree.



Exercise 5.5

There is a monoid instance on pairs of lists. Define a monoid instance for *Ordering* such that *uncurry compare* on pairs of lists *of the same length* is a monoid homomorphism.

```
instance Monoid ([a], [a]) where
  ϵ                = ([], [])
  (xl, xr) ◇ (yl, yr) = (xl ++ yl, xr ++ yr)
```

Here is the type of *compare*:

```
compare :: Ord a ⇒ a → a → Ordering
```

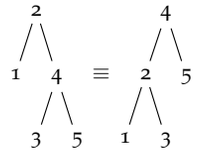
The result of *compare x y* returns *LT*, *EQ* or *GT* depending on whether $x < y$, $x \equiv y$, or $x > y$.

```
data Ordering = LT | EQ | GT
```

Exercise 5.6

Consider a data type representing a set, implemented using AVL trees. The trees will contain no duplicates. Write an *Eq* instance for this type. $(\equiv)$ should be linear, and should not differentiate between trees which are balanced differently.

Given sets xs and ys, f is defined to be *well behaved* if it obeys:



$xs \equiv ys \Rightarrow f\ xs \equiv f\ ys$

Discuss whether the functions *member x* or *insert x* are well behaved. Give other examples of well behaved and non-well behaved functions.

Exercise 5.7

The Boom hierarchy is a classification of data structures based on laws. Given some data structure $S\ a$, with a function $insert :: a \to S\ a \to S\ a$, and $(\cup) :: S\ a \to S\ a \to S\ a$, we can classify it as one of 16 different structures depending on which of the following laws hold:

$$(x \cup y) \cup z = x \cup (y \cup z) \qquad \text{(associativity, 1)}$$
$$\emptyset \cup x = x = x \cup \emptyset \qquad \text{(identity, 2)}$$
$$x \cup y = y \cup x \qquad \text{(commutativity, 3)}$$
$$x \cup x = x \qquad \text{(idempotence, 4)}$$

Name the data structures which obey the following laws:

1. 1 only.
2. 1 and 2 only.
3. 1, 2, and 3 only.
4. 1, 2, 3, and 4.

Exercise 5.8

While AVL trees measure imbalance by the height of siblings, *weight*-balanced trees use the number of elements in each sibling.

As in an AVL tree, the tree is rebalanced by single or double rotations. A rebalancing is triggered when the ratio of sizes of two subtrees exceeds some predefined factor, called Δ. A double rotation is triggered if, inside the larger subtree, the ratio of sizes of its two children exceed some factor Γ.

Write a function $insert :: Ord\ a \Rightarrow a \to WTree\ a \to WTree\ a$. Your code should define Δ and Γ as constants; you may set them to 3 and 2 respectively for testing.

Describe how the performance characteristics of *insert* and *member* might change if we were to raise or lower Δ and Γ.

```
data WTree a = WTip
             | WNode Int (WTree a) a (WTree a)
```

Exercise 5.9

Define a linear-time function $invariants :: Ord\ a \Rightarrow RBTree\ a \to Bool$ which tests that the following invariants hold on an *RBTree*:

1. It is an ordered binary search tree with no duplicates.
2. Every red node has a black parent.
3. There are the same number of black nodes on any path from root to tip.

```
data Colour = R | B deriving Eq
data RBTree a
  = RBTip
  | RBNode Colour (RBTree a) a (RBTree a)
```

Exercise 5.1

When implementing an AVL tree, at each node it is possible to store just the difference in height between the two children. According to the invariants on an AVL tree, the bias can either be -1, 0, or 1.

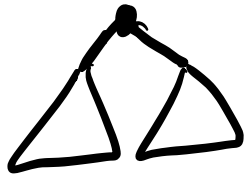
Implement *insert* :: *Ord a* $\Rightarrow$ *a* $\rightarrow$ *DTree a* $\rightarrow$ *DTree a* and *delete* :: *Ord a* $\Rightarrow$ *a* $\rightarrow$ *DTree a* $\rightarrow$ *DTree a* on *DTree*, an AVL tree which stores differences rather than heights.

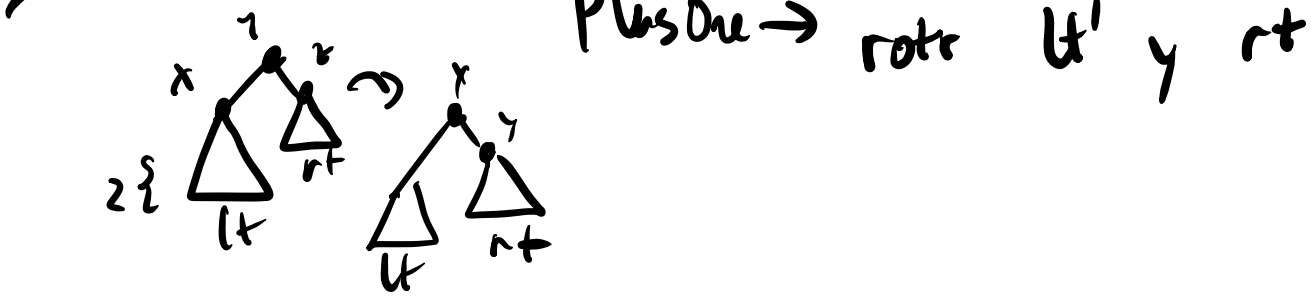
```
> data DTree a = DTip
>              | DNode Diff (DTree a) a (DTree a)

> data Diff = MinusOne | Zero | PlusOne.

> insert :: Ord a => a -> DTree a -> DTree a
> insert x t = snd (ins x t)

> where                                  -- True if |t'|>|t| where t'=snd(ins t).
>     ins :: DTree a -> (Bool, DTree a)
>     ins DTip = (True, DNode Zero DTip x DTip)
>     ins x (DNode b lt y rt)
>       | x == y = (False, t)
>       | x<y = case ins x lt of
>           (False, lt') -> (False, DNode b lt' y rt)
>           (True, lt') -> case b of
>             MinusOne -> (False, DNode Zero lt' y rt)
>             Zero -> (True, DNode PlusOne lt' y rt)
>
```





Exercise 5.2

Write a function *fromOrdList* :: $[a] \rightarrow$ *HTree a* which builds an AVL tree from a sorted list in worst case linear time.

```
> fromList :: [a] -> HTree a
> fromList xs = fst ( go (length xs) xs )
>   where
>     go :: Int -> [a] -> (HTree a, [a])
>     go 0 xs = (HTip, xs)
>     go n xs = (hnode' lt x rt, xs'')
>       where
>         (lt, xs') = go m xs
>         (rt, x:xs'') = go (n-m-1) xs'
>                        ^ pivot
>         m = n `div` 2.
```

Exercise 5.3

The implementation of *member* presented in lectures uses $2d$ comparisons in the worst case, where d is the depth of the tree. Reimplement *member* to perform at most $d + 1$ comparisons, using only $(\leqslant)$ to compare elements.

```
> member :: Int -> HTree Int -> Bool
> member x HTip = False
> member x (HNode _ lt p rt)
>     | x <= p = go p x lt
>     | otherwise = member x rt
>   where
>     go y x HTip = y == x
>     go y x (HNode _ lt p rt)
>       | x <= p = go p x lt
>       | otherwise = go y x rt
```

Exercise 5.4

We can define a kind of *map* on search trees as follows:

$$
\begin{aligned}
&mapt :: (a \to b) \to HTree\ a \to HTree\ b \\
&mapt\ f\ HTip = HTip \\
&mapt\ f\ (HNode\ b\ ls\ x\ rs) = HNode\ b\ (mapt\ f\ ls)\ (f\ x)\ (mapt\ f\ rs)
\end{aligned}
$$

Describe the minimal condition to which f must adhere in order for the output of $mapt\ f$ to be a valid search tree, given valid input.

Define a function $mapt' :: Ord\ b \Rightarrow (a \to b) \to Tree\ a \to Tree\ b$ which does not require f to adhere to your condition in order to produce a valid tree.

$$x < y \implies f\,x < f\,y$$

```
mapt' :: (a -> b) -> HTree a -> HTree a
mapt' f = fromList . map f . toList
```

Exercise 5.5

There is a monoid instance on pairs of lists. Define a monoid instance for *Ordering* such that *uncurry compare* on pairs of lists *of the same length* is a monoid homomorphism.

$$\begin{aligned}&\textbf{instance } \mathit{Monoid}\ ([a],[a]) \textbf{ where}\\ &\quad \epsilon \qquad\qquad\quad = ([\,],[\,])\\ &\quad (xl,xr)\diamond(yl,yr) = (xl +\!\!+ yl, xr +\!\!+ yr)\end{aligned}$$

Here is the type of *compare*:

$$\mathit{compare} :: \mathit{Ord}\ a \Rightarrow a \to a \to \mathit{Ordering}$$

The result of *compare* x y returns *LT*, *EQ* or *GT* depending on whether $x < y$, $x \equiv y$, or $x > y$.

$$\textbf{data } \mathit{Ordering} = \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT}$$

ucompare = Ord a ⇒
uncurry compare :: ([a],[a]) → Ordering

ucompare (x,y)

monoid homomorphism:

h(x ◇1 y) = h x ◇2 h y

h ε1 = ε2

instance Monoid Ordering where
ε = EQ
EQ ◇ y = y

x ◇ _ = x

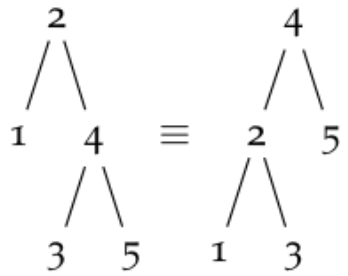
Exercise 5.6

Consider a data type representing a set, implemented using AVL trees. The trees will contain no duplicates. Write an *Eq* instance for this type. $(\equiv)$ should be linear, and should not differentiate between trees which are balanced differently.

Given sets *xs* and *ys*, *f* is defined to be *well behaved* if it obeys:

$$xs \equiv ys \Rightarrow f\ xs \equiv f\ ys$$

Discuss whether the functions *member x* or *insert x* are well behaved. Give other examples of well behaved and non-well behaved functions.




```
> hTreeToList :: HTree a → [a]
> hTreeToList t = go t []
>     go HTip xs = xs
>     go (HNode _ lt x rt) xs
>       = go lt (x : go rt xs)
```