COMP50001: Algorithm Design & Analysis

Sheet 6 (Week 7)

Exercise 6.1

Notice that the definition of *montePi* only generates random x and y values between 0 and 1, thus only hitting points in one quadrant of the square. Explain why this measure approximates $\pi/4$.

```
montePi :: Double
montePi = loop (mkStdGen 42) 0 0 where
  loop seed m n
    | n ≡ 100000 =
      4 * fromIntegral m / fromIntegral n
    | otherwise    = loop seed'' m' n'
    where n'  = n + 1
          m'  = if inside (x, y) then m + 1 else m
          (x, seed')  = randomR (0, 1) seed
          (y, seed'') = randomR (0, 1) seed'
inside :: (Double, Double) → Bool
inside (x, y) = x * x + y * y ⩽ 1
```

Exercise 6.2

Devise a randomized Monte Carlo algorithm to find the value of $\sqrt{2}$. It need not be efficient. *Hint: Consider the ratio of values between* 0 *and* 2 *that are less than* $\sqrt{2}$.

Exercise 6.3

The *insertBTree′* function does not produce correct results when the same element is inserted more than once, since it always increments the size of the tree even when no new elements are added. Discuss how the implementation can be changed without affecting asymptotic complexity.

```
insertBTree' :: Ord a ⇒ a → RBTree a → RBTree a
insertBTree' x (RBTree seed n t)
  | p ≡ 0      = RBTree seed' (n + 1)
                   (insertRoot x t)
  | otherwise = RBTree seed' (n + 1)
                   (insertBTree x t)
  where
    (p, seed') = randomR (0, n) seed
```

```
data Treap a = Empty | Node (Treap a) a Int (Treap a)
insert :: Ord a ⇒ a → Int → Treap a → Treap a
insert x p Empty = Node Empty x p Empty
insert x p (Node lt y q rt)
  | x < y = lnode (insert x p lt) y q rt
  | x ≡ y = Node lt y q rt
  | x > y = rnode lt y q (insert x p rt)
lnode :: Treap a → a → Int → Treap a → Treap a
lnode Empty y q rt = Node Empty y q rt
lnode lt@(Node llt x p lrt) y q rt
  | q ⩽ p      = Node lt y q rt
  | otherwise = Node llt x p (Node lrt y q rt)
rnode :: Treap a → a → Int → Treap a → Treap a
rnode lt x p Empty = Node lt x p Empty
rnode lt x p rt@(Node rlt y q rrt)
  | p ⩽ q      = Node lt x p rt
  | otherwise = Node (Node lt x p rlt) y q rrt
```

Exercise 6.4

Prove that for a set of integers, each paired with a priority

$$S = \{(x_i, p_i) \mid 1 \leqslant i \leqslant n\}$$

where $x_i, p_i :: Int$, if $x_i \neq x_j$ and $p_i \neq p_j$ for any $i \neq j$, then there is a unique $t :: Treap\ Int$ such that t satisfies the invariant of treaps and *nodes t* contains the same set of elements as S, where

```
nodes :: Treap a → [(a, Int)]
nodes Empty             = []
nodes (Node lt x p rt) = (x, p) : nodes lt ++ nodes rt
```

Argue that as a consequence inserting the elements $\{(x_i, p_i)\}$ into a treap in different orders gives rise to the same treap, assuming all elements and priorities are distinct.

Exercise 6.5

Given $lt, rt :: Treap\ a$ such that $x \leqslant y$ for any $(x, \_)$ in *nodes lt* and any $(y, \_)$ in *nodes rt*, prove that *merge lt rt* satisfies the invariants of treaps.

```
merge :: Treap a → Treap a → Treap a
merge Empty rt = rt
merge lt Empty = lt
merge lt@(Node llt x p lrt) rt@(Node rlt y q rrt)
  | p < q      = Node llt x p (merge lrt rt)
  | otherwise = Node (merge lt rlt) y q rrt
```

Exercise 6.6

Implement a $split :: Ord\ a \Rightarrow Treap\ a \to a \to (Treap\ a, Treap\ a)$ such that *split t x* computes (lt, rt) where lt contains exactly

$$filter\ (\lambda(y, \_) \to y < x)\ (nodes\ t)$$

as nodes and rt contains exactly

$$filter\ (\lambda(y, \_) \to y \geqslant x)\ (nodes\ t)$$

It should run in $O(depth\ t)$ time.

merge is a right-inverse of *split*:

$$uncurry\ merge\ (split\ t\ x) = t$$

$depth :: Treap\ a \to Int$
$depth\ Empty = 0$
$depth\ (Node\ lt\ x\ p\ rt) = 1 + max\ (depth\ lt)\ (depth\ rt)$

Exercise 6.7

Implement insertion and deletion for treaps using only *merge* and *split*

$insert' :: Ord\ a \Rightarrow a \to Int \to Treap\ a \to Treap\ a$
$delete' :: Ord\ a \Rightarrow a \to a \to Treap\ a \to Treap\ a$

such that $delete'\ x\ y\ t$ removes all elements in interval $[x, y)$ from t. In this exercise, we allow duplicate elements in a treap.

Exercise 6.8

Unordered lists, i.e. the *List* type class, can be efficiently implemented by a variant of treaps in which the *indices* of a list $0, \dots, length\ xs - 1$ are used as the ordered keys (the x in $Node\ lt\ x\ p\ rt$) of a treap and the list elements are stored as *payloads* in treap nodes. Define

$\textbf{data}\ TList\ a = EmptyT \mid NodeT\ (TList\ a)\ Int\ a\ Int\ (TList\ a)$

with invariants that any $NodeT\ lt\ s\ x\ p\ rt$ satisfies

$$s = length\ lt + length\ rt + 1$$

and the heap invariant that $p \leqslant priority\ lt$ and $p \leqslant priority\ rt$ whenever lt or rt is not empty. Note that the indices are *not* stored in the nodes. The list represented by a $TList\ a$ is

$toList :: TList\ a \to [a]$
$toList\ EmptyT = [\,]$
$toList\ (NodeT\ lt\ \_\ x\ \_\ rt) = toList\ lt +\!\!+ [x] +\!\!+ toList\ rt$

These following functions are useful in this exercise:

$length :: TList\ a \to Int$
$length\ EmptyT = 0$
$length\ (NodeT\ \_\ s\ \_\ \_\ \_) = s$

$payload :: TList\ a \to a$
$payload\ EmptyT = error\ \texttt{"empty"}$
$payload\ (NodeT\ \_\ \_\ x\ \_\ \_) = x$

$priority :: TList\ a \to Int$
$priority\ EmptyT = error\ \texttt{"empty"}$
$priority\ (NodeT\ \_\ \_\ \_\ p\ \_) = p$

1. Implement $single :: MonadRandom\ m \Rightarrow a \to m\ (TList\ a)$ that creates a singleton list from an element with a random priority generated using the *MonadRandom* interface.
2. Implement $(!!) :: TList\ a \to Int \to a$ such that $xs\ !!\ n = toList\ xs\ !!\ n$ for any $xs :: TList\ a$ and n. The function should run in $O(depth\ t)$ time.
3. Implement $(+\!\!+) :: TList\ a \to TList\ a \to TList\ a$ in a way similar to $merge :: Treap\ a \to Treap\ a \to Treap\ a$ (see Exercise 6.5). The time complexity of $xs +\!\!+ ys$ should be in $O(depth\ xs + depth\ ys)$. Explain why the indices of the elements are not stored in *TList* nodes.
4. Implement $splitAt :: Int \to TList\ a \to (TList\ a, TList\ a)$ in a way similar to *split* in Exercise 6.6 such that if $(xs, ys) = split\ n\ zs$ then $xs +\!\!+ ys = zs$ and $length\ xs = n$. The time complexity of $split\ n\ xs$ should be in $O(depth\ xs)$.

Same as treaps, the expected depth of $t :: TList\ a$ is $log\ (length\ t)$ when the priorities are random.

$depth :: TList\ a \to Int$
$depth\ EmptyT = 0$
$depth\ (NodeT\ l\ \_\ \_\ \_\ r) = 1 + max\ (depth\ l)\ (depth\ r)$

Both *tail* and *init* are special cases of *splitAt*.

Solutions to the Exercises

Solution 6.1

The ratio between a circle with radius 1 and a square with sides of length 2 is the same as the ratio between a quarter circle with radius 1 and a square with sides of length 1. This can easily be show with some algebra:

$$\pi : 2 \times 2 \Leftrightarrow \pi/4 : 4/4$$

Solution 6.2

A possible way of sampling is

```
root2 :: Double
root2 = loop (mkStdGen 42) 0 0 where
  loop seed m n
    | n ≡ 100000 = 9 * m / fromIntegral n
    | otherwise   = loop seed′ m′ n′
    where n′ = n + 1
          m′ = if inside x then m + 1 else m
          (x, seed′) = randomR (0, 9) seed
  inside :: Double → Bool
  inside x = x * x ⩽ 2
```

Solution 6.3

There are two potential solutions. The first is to redefine *insertBTRee* and *insertRoot* so that they return a flag indicating whether a value was actually inserted. This can then be inspected in *insertBTree′* and the counter can be incremented when appropriate.

A second solution is to store the size in the *BNode* constructor, and to use smart constructors that increment the size only when a value has indeed been inserted.

Solution 6.4

Prove by induction on the size of S.

1. If $n = 0$, the unique choice of t is *Empty*.

2. If $n > 0$, suppose t is treap with $\mathit{nodes}\ t = S$. Because treap t is a heap in the priorities of the nodes, the root node of t must have the highest priority. By the assumption that all priorities are distinct, the node with the highest priority is unique. Let the root be $(x_r, p_r) \in S$ for some r. The left subtree of the root must contain the nodes

$$S_l = \{(x_i, p_i) \mid 1 \leqslant i \leqslant n,\ x_i < x_r\}$$

The assumption that all x_i are distinct guarantees that $S = S_l \cup S_r \cup \{(x_r, p_r)\}$

and the right subtree must contain the nodes

$$S_r = \{(x_i, p_i) \mid 1 \leqslant i \leqslant n,\ x_i > x_r\}$$

because the treap t is also a binary search tree in terms of the keys x_i. Since the sizes of S_l and S_r are strictly smaller than S, by the inductive hypothesis, the left and right subtrees are uniquely determined. Thus t is unique.

The order of insertions into a treap does not matter because of the uniqueness of the treap containing the set of nodes.

Solution 6.5

Suppose lt and rt satisfy the invariants of treaps. If lt or rt is empty, $merge\ lt\ rt$ must satisfy the invariants because

$$merge\ Empty\ rt = rt \quad \text{and} \quad merge\ lt\ Empty = lt$$

Otherwise lt is some $Node\ llt\ x\ p\ lrt$ and rt is $Node\ rlt\ y\ q\ rrt$. If $p < q$ (priority p is higher than q), $merge\ lt\ rt$ is

$$Node\ llt\ x\ p\ (merge\ lrt\ rt) \tag{1}$$

Because lrt is strictly smaller than lt, we can assume that $merge\ lrt\ rt$ is a treap from the inductive hypothesis. To see that the result of merging (1) satisfies the BST invariant of treaps, notice that every (z, r) in $nodes\ \ (merge\ \ lrt\ \ rg)$ comes from either rt or lrt. In either case, $z \leqslant x$ by the assumption in the exercise or the BST invariant of lt. The result (1) also respects the heap invariant of treaps because if (z, r) is from lrt, then $r \geqslant p$ by the heap invariant lt of (lrt is the right subtree of $lt\ =\ Node\ \ llt\ \ x\ \ p\ \ lrt$), and if (z, r) is from rt, then $r \geqslant q \geqslant p$ (q is the priority of the root node of rt). The symmetric case $p \geqslant q$ is similar.

Solution 6.6

This can be done by

$$
\begin{array}{l}
split :: Ord\ a \Rightarrow Treap\ a \rightarrow a \rightarrow (Treap\ a, Treap\ a) \\
split\ Empty\ \_ = (Empty, Empty) \\
split\ (Node\ lt\ y\ p\ rt)\ x \\
\quad \mid y < x = \mathbf{let}\ (rlt, rrt) \quad\ \ = split\ rt\ x\ \mathbf{in}\ (Node\ lt\ y\ p\ rlt, rrt) \\
\quad \mid otherwise = \mathbf{let}\ (llt, lrt) = split\ lt\ x\ \mathbf{in}\ (llt, Node\ lrt\ y\ p\ rt)
\end{array}
$$

Solution 6.7

Both $insert'$ and $delete'$ can be straightforwardly expressed as $merge$ and $split$:

$$
\begin{array}{l}
insert' :: Ord\ a \Rightarrow a \rightarrow Int \rightarrow Treap\ a \rightarrow Treap\ a \\
insert'\ x\ p\ t = merge\ lt\ (merge\ (Node\ Empty\ x\ p\ Empty)\ rt)
\end{array}
$$

$$
\begin{aligned}
&\quad \textbf{where } (lt, rt) = split\ t\ x \\
&delete' :: Ord\ a \Rightarrow a \to a \to Treap\ a \to Treap\ a \\
&delete'\ x\ y\ t = merge\ lt\ rrt\ \textbf{where} \\
&\quad (lt, rt) \quad = split\ t\ x \\
&\quad (rlt, rrt) = split\ rt\ y
\end{aligned}
$$

Solution 6.8

Define a smart constructor to be used throughout this exercise:

$$
\begin{aligned}
&node :: TList\ a \to a \to Int \to TList\ a \to TList\ a \\
&node\ lt\ x\ p\ rt = NodeT\ lt\ (length\ lt + 1 + length\ rt)\ x\ p\ rt
\end{aligned}
$$

1. A random priority can be generated by the *getRandom* function from the *MonadRandom* interface:

$$
\begin{aligned}
&single :: MonadRandom\ m \Rightarrow a \to m\ (TList\ a) \\
&single\ a = \textbf{do}\ p \leftarrow getRandom \\
&\qquad\qquad\quad return\ (node\ EmptyT\ a\ p\ EmptyT)
\end{aligned}
$$

2. Lookup can be done as follows:

$$
\begin{aligned}
&(!!) :: TList\ a \to Int \to a \\
&EmptyT\ !!\ n = error\ \texttt{"out of bounds"} \\
&(NodeT\ lt\ \_\ x\ \_\ rt)\ !!\ n \\
&\quad \mid n < length\ lt\ = lt\ !!\ n \\
&\quad \mid n \equiv length\ lt = x \\
&\quad \mid otherwise \quad = rt\ !!\ (n - length\ lt - 1)
\end{aligned}
$$

   It clearly runs in $O(depth\ \ t)$ for input t because each recursion descends into a subtree.

3. It can be done as follows:

$$
\begin{aligned}
&(+\!\!+) :: TList\ a \to TList\ a \to TList\ a \\
&xs +\!\!+ EmptyT = xs \\
&EmptyT +\!\!+ ys = ys \\
&lt@(NodeT\ llt\ \_\ x\ p\ lrt) +\!\!+ rt@(NodeT\ rlt\ \_\ y\ q\ rrt) \\
&\quad \mid p < q \qquad = node\ llt\ x\ p\ (lrt +\!\!+ rt) \\
&\quad \mid otherwise = node\ (lt +\!\!+ rlt)\ y\ q\ rrt
\end{aligned}
$$

   We do not store the indices of nodes explicitly because when two *TList*'s are concatenated using $xs +\!\!+ ys$, the indices of *all* elements in the second list are shifted by *length xs*. Thus it would need $\Omega(length\ ys)$ time to update these indices if they are explicitly stored, which is too expensive. Instead, the indices in *TList* are implicitly calculated using the second field of *NodeT* when needed.

4. It can be done as follows:

```
splitAt :: Int → TList a → (TList a, TList a)
splitAt _ EmptyT = (EmptyT, EmptyT)
splitAt n (NodeT lt _ x p rt)
  | length lt < n = let (rlt, rrt) = splitAt (n − length lt − 1) rt in (node lt x p rlt, rrt)
  | otherwise     = let (llt, lrt) = splitAt n lt in (llt, node lrt x p rt)
```