1 Algorithm Cost

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) -- Assume ys is sorted.
  | x <= y     = x : y : ys
  | otherwise = y : insert x ys
```

Cost $T_{\text{insert}}(0) = 1$; $T_{\text{insert}}(n) = 1 + T_{\text{insert}}(n-1)$. By expansion, this is $\mathcal{O}(n)$.

```
isort :: Ord a => [a] -> [a] -- O(n^2)
isort [] = []
isort (x:xs) = insert x (isort xs)
```

$T_{\text{isort}}(n) = 1 + T_{\text{insert}}(n-1) + T_{\text{isort}}(n-1)$. In general:

```
T(k)=T(x)=0 -- Constants, variables
T(f e1 .. en) = T(f) e1 .. en + T(e1) + .. + T(en)
T(p ? e1 : e2) = T(p) + p ? T(e1) : T(e2)
```

$$T(f(g(x))) = T(f)(gx) + T(g)x$$

1.1 Normal Form

Lazy things are in **weak headed NF** and strict things are in **NF**. e is in NF if:

- `e = \x -> e'` and `e'` is in NF.
- `e = x` and `x` is a normal variable.
- `e = f x` where `f` and `x` are normal.

WHNF doesn't need normal lambda bodies.

1.2 Complexity Classes

$f(n) \in o(g(n))$	$f \prec g$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in \mathcal{O}(g(n))$	$f \preccurlyeq g$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \in \Theta(g(n))$	$f \asymp g$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \in \Omega(g(n))$	$f \succcurlyeq g$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
$f(n) \in \omega(g(n))$	$f \succ g$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

We can also define the sets as:

$$f(n) \in o(g(n)) \Leftrightarrow \forall\delta > 0.\exists m > 0.\forall n > m. f(n) < \delta g(n)$$

$$f(n) \in \mathcal{O}(g(n)) \Leftrightarrow \exists\delta > 0.\exists m > 0.\forall n > m. f(n) \leq \delta g(n)$$

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n))$$

$$f(n) \in \Omega(g(n)) \Leftrightarrow \exists\delta > 0.\exists m > 0.\forall n > m. f(n) \geq \delta g(n)$$

$$f(n) \in \omega(g(n)) \Leftrightarrow \forall\delta > 0.\exists m > 0.\forall n > m. f(n) > \delta g(n)$$

2 Lists

```
data [a] where
  []    :: [a] -- O(1)
  (:)   :: a -> [a] -> [a] -- O(1)
  (++) :: [a] -> [a] -> [a] -- O(n), n = length xs
  [] ++ ys       = ys
  (x:xs) ++ ys = x : (xs ++ ys)
```

We can define folds on lists as:

```
concat :: [[a]] -> [a] -- O(mn)
concat [] = []
concat (xs:xss) = xs ++ concat xss
-- foldr f k [a,b,c] = f a (f b (f c k))
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k [] = k
foldr f k (x:xs) = f x (foldr f k xs)
-- foldl f k [a,b,c] = f (f (f k a) b) c
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
xs ++ ys = foldr (:) ys xs
concat xss = foldr (++) [] xss -- O(mn)
concat yys = folfl (++) [] yss -- O(n^2m)
```

If `f` is assoc and `k` is a zero under `f`, then its a **monoid**:

```
-- e.g. Int: <0, +>, <1, *>
-- e.g. [a]: <[], ++>
class Monoid m where
  mempty :: m
  (<>) :: m -> m -> m
  mempty <> x = x
  x <> mempty = x
  (x <> y) <> z = x <> (y <> z)
```

3 Abstract Datatypes

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
values (Leaf x) = [x]
values (Node l r) = values l ++ values r -- O(n^2)
```

To do better than $\mathcal{O}(n^2)$, define a sequence:

```
class Seq seq where
  -- nil, cons, snoc, append, len, toList, fromList
instance Seq [] where --Can we do better than this?
  -- nil, cons, toList, fromList are O(1)
  -- snoc, append, len are O(n)
```

Since `xs ++ (ys ++ zs) = (xs ++) . (ys ++) . (zs ++) []`, we see bracketing has no effect on the result. Hence, we can write:

```
-- Good for construction, bad for processing.
data DList a = DList ([a] -> [a])
instance Seq DList where
  -- nil, cons, snoc, append, fromList are O(1)
  -- len, head, tail, init, last, !!, toList are O(n)
-- Great for optimising:
values' :: Tree a -> [a] -- O(n)
values' = toList . go
  where go :: Tree a -> DList a -- O(n)
        go (Leaf x) = cons x nil
        go (Fork l r) = go l `append` go r
```

4 Divide and Conquer

A **DAC** algorithm *splits problems* into smaller subproblems, solves those into *subsolutions*, and *recombines* them. For example, **merge sort**:

```
splitAt xs n = (take n xs, drop n xs) -- O(n)
splitHalf xs = splitAt xs (length xs `div` 2) -- O(n)
merge :: Ord a => [a] -> [a] -> [a] -- O(m + n)
merge [] ys = ys
merge xs [] = xs
merge xxs@(x:xs) yys@(y:ys)
  | x <= y     = x : merge xs yys
  | otherwise = y : merge xxs ys
msort :: Ord a => [a] -> [a]
msort [] = []
msort [x] = [x]
msort xs = let (us, vs) = splitHalf xs
           in merge (msort us, msort vs)
```

$T_{\text{msort}}(n) = T_{\text{len}}(n) + T_{\text{splitAt}}(\frac{n}{2}) + T_{\text{merge}}(\frac{n}{2}) + 2T_{\text{msort}}(\frac{n}{2}) \in \Theta(n \log n)$. (Best and worst!) **Quicksort**:

```
partition :: (a -> Bool) -> [a] -> ([a], [a])--O(n)
partition p xs = (filter p xs, filter (not . p) xs)
allLess :: Ord a => a -> [a] -> ([a], [a]) -- O(n)
allLess x xs = partition (< x) xs
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = let (us, vs) = allLess x xs
               in (qsort us) ++ [x] ++ (qsort vs)
```

Best case $T_{\text{qsort}}(n) = T_{\text{allLess}}(n-1) + 2T_{\text{qsort}}(\frac{n-1}{2}) + T_{++}(1) + T_{++}(\frac{n-1}{2}) = \Omega(n \log n)$. *Worst case* $T_{\text{qsort}}(n) = T_{\text{allLess}}(n-1) + T_{\text{qsort}}(0) + T_{\text{qsort}}(n-1) + T_{++}(0) + T_{++}(1) = \mathcal{O}(n^2)$. It may be better to take a random elem.

```
-- These two methods are O(N), making log N tree
foldArray f xs = go 0 (n-1)
where (arr, n) = (toArray xs, length xs)
  go i j
  | i == j     = arr ! i
  | otherwise = f (go i mid) (go (mid + 1) j)
  where mid = (i + j) `div` 2
foldMerge _ [x] = x
foldMerge f xs  = foldMerge f (mergePairs xs)
where
  mergePairs (x:y:rest) = f x y : mergePairs rest
  mergePairs [x]         = [x]
  mergePairs []          = []
```

5 Dynamic Programming

Write a bad solution *recursively*, catch *sub-solutions*.

```
-- Example: fibonacci (bad spatial complexity)
fib' :: Int -> Integer -- O(n)
fib' n = table ! n
  where table :: Array Int Integer
        table = tabulate (0, n) memo
        memo :: Int -> Integer
        memo 0 = 0
        memo 1 = 1
        memo i = table ! (i - 1) + table ! (i - 2)

-- Example: edit distance
type Text = Array Int Char
fromString :: String -> Text -- O(n)
fromString cs = listArray (0, length cs - 1) cs

dist'' :: String -> String -> Int
dist'' cs1 cs2 = table ! (m, n)
  where table :: Array (Int, Int) Int
        table = tabulate ((0,0),(m,n)) (uncurry memo)
        memo :: Int -> Int -> Int
        memo 0 j = j
        memo i 0 = i
        memo i j = minimum
          [ table ! (i - 1, j) + 1
          , table ! (i, j - 1) + 1
          , table ! (i-1, j-1) + c1 == c2 ? 0 : 1 ]
        where c1 = cs1 ! (m - i)
              c2 = cs2 ! (n - j)
        m, n = length cs1, length cs2
        str1, str2 = fromString cs1, fromString cs2
```

5.1 Evidence of Work

```
cs = [1, 2, 3, 5, 10, 20, 50, 100, 200]
change :: Pence -> [Pence]
change g = table ! g
where table :: Array Pence [Pence]
      table = tabulate (0, g) memo
      memo :: Pence -> [Pence]
      memo 0 = []
      memo g = minimumBy (compare `on` length)
        [ c : (table ! (g - c)) | c <- cs, c <= g ]
-- To keep track of work done:
change' :: Pence -> [Pence] -- O(n)
change' g = Seq.toList (table ! g)
where t :: Array Pence (LenList Pence)
      t = tabulate (0, g) memo
      memo :: Pence -> LenList Pence
      memo 0 = Seq.nil
      memo g = minimumBy (compare `on` Seq.length)
        [ cons c ( t ! (g - c)) | c <- cs, c <= g ]

-- And for edit distance example:
edits' :: String -> String -> [String]
edits' cs1 cs2 = Seq.toList $ table ! (m, n)
where table :: Array (Int, Int) (LenList String)
  table = tabulate ((0, 0), (m, n)) (uncurry memo)
  memo :: Int -> Int -> LenList String
  memo 0 j = Seq.inits $ drop (n - j) cs2
  memo i 0 = Seq.tails $ drop (m - i) cs1
  memo i j = minimumBy (compare `on` Seq.length)
    [ Seq.cons cs1' (table ! (i - 1, j))
    , Seq.cons cs1' (Seq.map (c2:) (table ! (i,j-1)))
    , (if c1 == c2 then id else Seq.cons cs1')
      (Seq.map (c2 :) (table ! (i-1, j - 1))) ]
  where c1, x2 = str1 (m - i), str2 (n - j)
        cs1' = drop (m - i) cs1
  m, n = length cs1, length cs2
  str1, str2 = fromString cs1, fromString cs2
```

6 Amortised Complexity

A **deque** splits list into two:

```
 -- xs = us ++ reverse sv in > Deque us sv <
data Deque a = Deque Int [a] [a]
toList :: Deque a -> [a] -- O(n)
toList (Deque _ us sv) = us ++ reverse sv
cons :: a -> Deque a -> Deque a -- O(1)
cons u (Deque n us sv) = Deque (n + 1) (u : us) sv
snoc :: Deque a -> a -> Deque a -- O(1)
snoc (Deque n us sv) v = Deque (n + 1) us (v : sv)
fromList :: [a] -> Deque a
fromList xs = Deque n us $ reverse vs
  where n = Seq.length xs
        (us, vs) = splitAd (div n 2) xs

-- However, we want minimal rebalancing:
-- null sv ==> length us <= 1; and vice versa
cons u (Deque n sv []) = Deque (n + 1) [u] sv
cons u (Deque n us sv) = Deque (n + 1) (u : us) sv
snoc (Deque n [] us) v = Deque (n + 1) us [v]
snoc (Deque n us sv) v = Deque (n + 1) us (v : sv)

head (Deque _ [] [v]) = v     -- O(1)
head (Deque _ (u : _) _) = u -- O(1)
last (Deque _ [u] []) = u     -- O(1)
last (Deque _ _ (v : _)) = v -- O(1)

tail (Deque 0 _ _) = undefined  -- O(1)
tail (Deque 1 _ _) = nil -- O(1)
tail (Deque _ [_] sv) = fromList (reverse sv)--O(n)
tail (Deque n (_:us) sv) = Deque (n-1) us sv-- O(1)
init (Deque 0 _ _) = undefined                -- O(1)
init (Deque 1 _ _) = nil                      -- O(1)
init (Deque _ us [_]) = fromList us           -- O(n)
init (Deque n us (_:sv)) = Deque (n-1) us sv-- O(1)
```

`tail` & `init` appear $\mathcal{O}(n)$ but are not. **Amortised complexity** is like a piggy-bank. We overpay and save every time we do something cheap, so that we have pocket money to pay for nice expensive tails. $C_{\text{opi}}(x) \leq A_{\text{opi}}(x) - (\Phi(x') - \Phi(x))$ where:

- $C_{\text{opi}}(x)$ is the cost of an operation `opi` on x.
- $A_{\text{opi}}(x)$ is the amortised cost of an operation `opi`.
- $\Phi(x)$ is a **potential function**. Largest just before something expensive, smallest just after something expensive. *For example*, for `tail` we can use:

```
Phi (Deque n us sv) = max (length us - length sv) 0
```

7 Binary Lists

Natural numbers have a correspondence to lists:

```
data Nat = Z | S Nat
inc n = S n -- cons x xs = x : xs
dec (S n) = n -- tail (_ : xs) = xs
add Z n = n -- [] ++ ys = ys
add (S m) n = S (add m n) -- (x:xs)++ys=x:(xs++ys)
```

Instead of peano, we use **binary**, same for `BinList`:

```
data BList a = BList !Int [Maybe (Bush a)]
data Bush a | L a | F (Bush a) (Bush a)
instance Seq BList where
  nil = BList 0 []                          -- O(1)
  length (BList n _) = n                    -- O(1)
  head = (!! 0)                             -- O(log n)
  last xs = xs !! (length xs - 1)           -- O(log n)
  fromList = foldr cons nil                 -- O(n)
  xs ++ ys = foldr cons ys (toList xs) -- O(m)
  init = fromList . init . toList           -- O(n)
  null = (== 0) . length                    -- O(1)

  cons x (BList n bs) = BList (n+1) (inc (L x) bs)
    where inc :: Bush a -> [Maybe (Bush a)] -> [Maybe
      (Bush a)] -- Amortised ~ O(1)
      inc t []              = [Just t]
      inc t (Nothing:ts) = (Just t):ts
      inc t (Just t':ts) =Nothing:(inc (F t t') ts)

  (!!) :: BList a -> Int -> a -- O(log n)
  BList n ts !! i
    | i < 0 || i >= n = error "Index out of bounds"
    | otherwise          = find ts i 1
  where find :: [Maybe (Bush a)] -> Int -> Int -> a
    -- No values here, we must be further up
    find (Nothing : ts) i szT = find ts i (szT * 2)
    findBush (Just t : ts)
      -- i is inside this bush!
      | i < szT = index t i (szT `div` 2)
      -- this is not the bush we are looking for
      | otherwise = find ts (i - szT) (szT * 2)
    index :: Bush a -> Int -> Int -> a
    index (L x) 0 1 = x
```

```
    index (F lt rt) i szT
      | i < szT = index lt i (szT `div` 2)
      | otherwise = index rt (i-szT) (szT `div` 2)

  tail :: BList a -> BList a -- ~O(1)?
  tail (BList n ts) = BList (n-1) (snd (dec ts))
  where dec :: [Maybe (Bush a)] -> (Bush a, [Maybe (Bush a)])
    dec (Just t : ts) = (t, ts)
    -- Tree returned by recursion is 2x our size
    -- So, break it in half and feed the rest back
    dec (Nothing : ts) = let (f t t', ts') = dec ts
                         in (t, Just t' : ts')
```

8 Random Access Lists

If we have a BList with $2^n - 1$ els, tail will be $\mathcal{O}(n)$. Instead, consider a list with no Nothings, **RAList**. A **NonEmptyTree** contains $2^{n+1} - 1$ els for depth n. Every NETree is paired with its number of els. The invariance is that every el in the list has an increasing size, except for the first two trees, which may be the same size.

```
data NETree = Tip a | Node (NETree a) a (NETree a)
data RAList = RAList !Int [(Int, NETree a)]
instance Seq RAList where
  nil = RAList 0 []                     -- O(1)
  head = (!! 0)                         -- O(log n)
  last xs = xs !! (length xs - 1)       -- O(log n)
  length (RAList n _) = n               -- O(1)
  null = (== 0) . length                -- O(1)
  init = fromList . init . toList       -- O(n)
  fromList = foldr cons nil             -- O(n)
  xs ++ ys = foldr cons ys (toList xs) -- O(m)

cons :: a -> RAList a -> RAList a -- O(1)
cons x (RAList n ((s1, t1) : (s2, t2) : ts))
-- The trees are equal size, so we can combine
|s1==s2 = RAList (n+1) ((s1+s2+1, Node t1 x t2):ts)
-- Otherwise, add a new tip
cons x (RAList n ts) = RAList (n+1) ((1, Tip x):ts)

tail :: RAList a -> RAList a -- O(1)
tail (RAList n ((1, Tip _) : ts)) = RAList (n-1) ts
-- Split & discard top, making 2 trees of same size
-- Next list guaranteed to double the size of curr
-- Which preserves the invariant
tail (RAList n ((s, Node t1 _ t2) : ts)) = RAList (n - 1) ((s', t1) : (s', t2) : ts)
  where s' = s `div` 2

(!!) :: RAList a -> Int -> a -- O(log n)
RAList n ts !! i
  | i < 0 || i >= n = error "Index out of bounds"
  | otherwise       = find ts i
where find :: [(Int, NETree a)] -> Int -> a
  find ((sz, t) : ts) i
    -- This is our tree
    | i < sz    = index t i ((sz - 1) `div` 2)
    -- This is not our tree
    | otherwise = find ts (i - sz)
  index :: NETree a -> Int -> Int -> a -- O(log n)
  index (Tip x) 0 0 = x
  index (Node t1 x t2) i sz
    | i == 0 = x
    | i <= sz = index t1 (i - 1) sz'
    | otherwise = index t2 (i - sz - 1) sz'
    where sz' = (sz - 1) `div` 2

toList :: RAList a -> [a] -- O(n)
toList (RAList _ ts) = toList (foldr ((++) . vals . snd) nil ts)
  where vals :: NETree a -> DList a
  vals (Tip x)       = cons x nil
  vals (Node t1 x t2)=cons x ((vals t1)++(vals t2))
```

9 Posets

Partially ordered sets are good for searching:

```
class Poset set where
  fromList    = foldr insert empty
  singleton x = insert x empty
  union s1 s2 = foldr insert s1 (toList s2)
  diff s1 s2  = foldr delete s1 (toList s2)
  intersection s1 s2 = fromList $ filter (`member` s2) (toList s1)
```

A Poset [] is simple, but Poset Tree is faster:

```
data Tree a = Tip | Node !Int (Tree a) a (Tree a)

node :: Tree a -> a -> Tree a -> Tree a
node l x r = Node (1+max (height l) (height r)) l x r
quicksort :: Ord a => [a] -> [a]
quicksort = toList . fromList @Tree
balanced :: Tree a -> Tree a -> Bool-->THE INVARIANT<
balanced l r = abs (height l - height r) <= 1
rotr :: Tree a -> Tree a -- O(1)
rotr (Node _ (Node _ llt x lrt) y rt)
  = node llt x (node lrt y rt)
rotl :: Tree a -> Tree a -- O(1)
rotl (Node _ lt x (Node _ rlt y rrt))
  = node (node lt x rlt) y rrt
balL :: Tree a -> a -> Tree a -> Tree a -- O(1)
balL lt x rt
  | balanced lt rt          = node lt x rt
  -- Pre: height lt > height rt + 1
  | height llt >= height lrt = rotr $ node lt x rt
  | otherwise          = rotr $ node (rotl lt) x rt
  where Node (_ llt _ lrt) = lt
balR :: Tree a -> a -> Tree a -> Tree a -- O(1)
balR lt x rt
  | balanced lt rt          = node lt x rt
  -- Pre: height lt + 1 < height rt
  | height rrt >= height rlt = rotl $ node lt x rt
  | otherwise          = rotl $ node lt x (rotr rt)
  where Node (_ rlt _ rrt) = rt
glue :: Tree a -> Tree a -> Tree a -- O(log n)
glue Tip rt = rt
glue lt Tip = lt
glue lt@(Node lh llt lx lrt) rt@(Node rh rlt rx rrt)
  | lh < rh   = let (x, rt') = extractMin rlt rx rrt
                in  balL lt x rt'
  | otherwise = let (x, lt') = extractMax llt lx lrt
                in  balR lt' x rt

instance Poset Tree where
  empty = Tip                          -- O(1)
  singleton x = node Tip x Tip         -- O(1)
  height Tip              = 0          -- O(1)
  height (Node h _ _ _) = h            -- O(1)
  toList = Seq.toList . go             -- O(n)
    where go :: Tree a -> DList a
      go Tip = Seq.nil
      go (Node lt x rt) = go lt `append` : x (go rt)

  member :: Ord a => a -> Tree a -> Bool -- O(log n):
  member _ Tip = False --      as the tree is balanced
  member x (Node _ lt y rt) = case compare x y of
    LT -> member x lt
    EQ -> True
    GT -> member x rt
  insert :: Ord a => a -> Tree a -> Tree a --O(log n)
  insert x Tip = singleton x
  insert x t@(Node _ lt y rt) = case compare x y of
    EQ -> t
    LT -> balL (insert x lt) y rt
    GT -> balR lt y (insert x rt)
  delete :: Ord a => a -> Tree a -> Tree a --O(log n)
  delete _ Tip = Tip
  delete x t@(Node _ lt y rt) = case compare x y of
    EQ -> glue lt rt
    LT -> balR (delete x lt) y rt
    GT -> balL lt y (delete x rt)

  -- Find the minimum element in a tree and return the rest of the tree, O(log n)
  extractMin :: Tree a -> a -> Tree a -> (a, Tree a)
  extractMin Tip min rest = (min, rest)
  extractMin (Node _ llt lx lrt) x rt=(m,balR t x rt)
    where (m, t) = extractMin llt lx lrt
  -- Find the maximum element in a tree and return the rest of the tree, O(log n)
  extractMax :: Tree a -> a -> Tree a -> (a, Tree a)
  extractMax rest max Tip = (max, rest)
  extractMax lt x (Node _ rlt rx rrt)=(m,balL lt x t)
    where (m, t) = extractMax rlt rx rrt
  minValue :: Ord a => Tree a -> a -- O(log n)
  minValue (Node _ lt x rt)= fst $ extractMin lt x rt
  maxValue :: Ord a => Tree a -> a -- O(log n)
  maxValue (Node _ lt x rt)= fst $ extractMax lt x rt
```

10 Red Black Trees

RBTrees self balance in $\sim\mathcal{O}(1)$, allowing for more bias. Here:

- Root is **Black**. Every **Red** node has **Black** parent.
- From root to any Tip, ∃ the same num of **Black** nodes.

```
blacken :: RBTree a -> RBTree a--makes R to B in O(1)
blacken (Node R lt x rt) = Node B lt x rt
blacken                  = id

-- Turn black node red and its children black
balance :: Colour -> RBTree a -> a -> RBTree a -> RBTree a -- O(1)
balance B (Node R (Node R a x b) y c) z d = Node R (Node B a x b) y (Node B c z d)
balance B (Node R a x (Node R b y c)) z d = Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R (Node R b y c) z d) = Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R b y (Node R c z d)) = Node R (Node B a x b) y (Node B c z d)
balance c lt x rt = Node c lt x rt

instance Poset RBTree where
  empty = Tip
  singleton x = Node B Tip x Tip

  member :: Ord a => a -> RBTree a -> Bool-- O(log n)
  member x Tip = False
  member x (Node _ lt y rt) = case compare x y of
    EQ -> True
    LT -> member x lt
    GT -> member x rt

  insert :: Ord a => a -> RBTree a -> RBTree a
  insert = blacken . insert'
    where insert' x Tip = Node _ Tip x Tip-- O(log n)
    insert' x (Node c lt y rt) = case compare x y of
      EQ -> Node c lt y rt
      LT -> balance c (insert x lt) y rt
      GT -> balance c lt y (insert x rt)

  delete :: Ord a => a -> RBTree a -> RBTree a--tspmo
  delete x = fromOrdList . List.delete x . toList

  toList :: RBTree a -> [a]
  toList = Seq.toList . go
    where go :: RBTree a -> DList a
    go Tip = nil
    go (Node _ lt x rt) = go lt `append` : x $ go rt

  minValue (Node _ Tip x _) = x
  minValue (Node _ lt _ _) = minValue lt
  maxValue (Node _ _ x Tip) = x
  maxValue (Node _ _ _ rt) = maxValue rt

data Digit a = One a (RbTree a) | Two a (RBTree a) a (RBTree a)

cons :: a -> [Digit a] -> [Digit a]
cons x ds = inc x Tip ds
where inc :: a -> RBTree a -> [Digit a] -> [Digit a]
  inc x t [] = [One x t]
  inc x t (One y t' : ds) = Two x t y t' : ds
  inc x t (Two y1 t1 y2 t2 : ds) = One x t : inc y1 (Node B t1 y2 t2) ds

fromOrdList :: [a] -> RBTree a
fromOrdList = foldl glue Tip . foldr cons []
  where glue :: RBTree a -> Digit a -> RBTree a
    glue t (One x t') = Node B t x t'
    glue t (Two x1 t1 x2 t2) = Node B (Node R t x1 t1) x2 t2
```

When we construct a tree from a sorted list, we notice that when we represent the tree as a list of half trees (generated by taking all the subtrees where the roots are on the left-hand spine), we can see that when the red nodes appear, they are next to a black rooted partial-tree of the same size. Thus we can define the following counting system, where we can count our way through the list of elements and in $\mathcal{O}(n)$ end up with a final "number" which can be turned into a tree.

11 Random Algorithms

- **Las Vegas**: prob of being correct, but takes random time.
- **Monte Carlo**: better answer given more iterations.

```
mkStdGen :: Int -> StdGen
random :: Random a => StdGen -> (a, StdGen)
randomR :: Random a => StdGen -> (a, a) -> (a, StdGen)

inside (x, y) = x^2 + y^2 <= 1

montePi :: Int -> Double
montePi !darts = go (mkStdGen 4) darts 0
where go :: StdGen -> Int -> Int -> Double
  go _ 0 hits = 4 * fromIntegral inside / fromIntegral darts
  go gen n hits
    | inside p  = go seed' (n-1) (hits+1)
    | otherwise = go seed' (n-1) hits
  where (p, seed') = randomR ((0, 0), (1, 1)) gen
```

A **treap** has priority and value. A **randomized treap** has random priorities for pseudo-balancing:

```
data Treap a = Tip | Node Int (Treap a) a (Treap a)
data RTreap a = RTreap STdGen (Treap a)

nodeL :: Int -> Treap a -> a -> Treap a -> Treap a
nodeL p lt@(Node lp llt u lrt) v rt
  | p <= lp   = Node p lt v rt
  | otherwise = Node lp llt u (Node p lrt v rt)
nodeR :: Int -> Treap a -> a -> Treap a -> Treap a
nodeR p lt u rt@(Node rp rlt v rrt)
  | p <= rp   = Node p lt u rt
  | otherwise = Node rp (Node p lt u rlt) v rrt

height (RTreap _ t) = height' t
  where height' Tip = 0
    height' (Node _ lt _ rt) =
      1 + max (height' lt) (height' rt)

instance Poset RTreap where
  empty = RTreap (mkStdGen 42) Tip
  insert :: Ord a => a -> RTreap a -> RTreap a
  insert x (RTreap s t) = RTreap s' (pinsert p x t)
    where (p, s') = random s
  pinsert :: Int -> a -> Treap a -> Treap a
  pinsert p x Tip               = Node p Tip x Tip
  pinsert p x (Node q lt y rt) = case compare x y of
    EQ -> t
    LT -> nodeL q (pinsert p x lt) y rt
    GT -> nodeR q lt y (pinsert p x rt)

  delete::Ord a=>a->RTreap a->RTreap a--O(log(n))?
  delete x (RTreap s t) = RTreap s (delete' x t)
    where delete' x Tip = Tip
    delete' x (Node p lt y rt) = case compare x y of
      LT -> Node p (delete' x lt) y rt
      GT -> Node p lt y (delete' x rt)
      EQ -> glue p lt rt
  glue::Int->Treap a->Treap a->Treap a -- O(log(n))?
  -- p is < the highest-priority of lt and rt
  glue _ Tip rt = rt
  glue _ lt Tip  = lt
  glue p (Node lp llt lx lrt) rt =
    let (max, lt') = maxView lp llt lx lrt
    in Node p lt' max rt
  minView::Int->Treap a->a->Treap a->(a, Treap a)
  minView _ Tip x rt = (x, rt)
  minView p (Node q llt lx lrt) x rt =
    let (min, rest) = minView q llt lx lrt
    in (min, Node p rest x rt)
  maxView::Int->Treap a->a->Treap a->(a, Treap a)
  maxView _ lt x Tip = (x, lt)
  maxView p lt x (Node q rlt rx rrt) =
    let (max, rest) = maxView q rlt rx rrt
    in (max, Node p lt x rest)
```

12 Zippers

A **zipper** is the equivalent of an *iterator*, a sliding window:

```
type LiztZ a = ([a], [a])
type BushZ a = (Bush a, [Either (Bush a) (Bush a)])
```