## Binary Search Trees (AVL Trees)

- records height

```
type Height = Int
data HTree a = HTip | HNode Height (HTree a) a (HTree a)
```

Constructor:

```
hnode :: HTree a -> a -> HTree a -> HTree a
hnode lt x rt = HNode h lt x rt
  where
    h = (height lt ⊔ height rt) + 1
height :: HTree a -> Int
height HTip = 0
height (HNode h _ _ _) = h
```

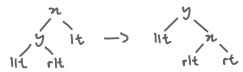
While insert, need to make sure height between two sibling trees have a max height difference of 1 !!

```
insert :: Ord a => a -> HTree a -> HTree a
insert x HTip = hnode HTip x HTip
insert x t@(HNode _ lt y rt)
  | x == y = t
  | x < y  = balancel (insert x lt) y rt
  | otherwise = balancer lt y (insert x rt)
```

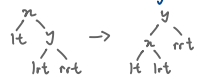
```
balancel :: HTree a -> a -> HTree a -> HTree a
balancel lt x rt
  | height lt - height rt ≤ 1 = hnode lt x rt
  | otherwise                 = case lt of
      HNode _ llt y rlt
        | height llt ≥ height rlt -> rotr (hnode lt x rt)
        | otherwise               -> rotr (hnode (rotl lt) x rt)
```

```
balancer :: HTree a -> a -> HTree a -> HTree a
balancer lt x rt
  | height rt - height lt ≤ 1 = hnode lt x rt
  | otherwise                 = case rt of
      HNode _ lrt y rrt
        | height lrt ≥ height rrt = rotl (hnode lt x (rotr rt))
        | otherwise               = rotl (hnode lt x rt)
```

```
rotr :: HTree a -> HTree a
rotr (HNode _ (HNode _ llt y rlt) x rt)
  = hnode llt y (hnode rlt x rt)
```



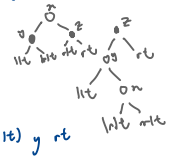
```
rotl :: HTree a -> HTree a
rotl (HNode _ lt x (HNode _ lrt y rrt))
  = hnode (hnode lt x lrt) y rrt
```



## Red-Black Tree

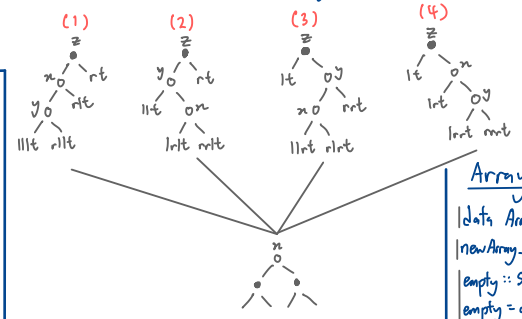
```
data Colour = R | B
data RBTree a = E | N Colour (RBTree a) a (RBTree a)
```

```
insert :: Ord a => a -> RBTree a -> RBTree a
insert x t = blacken (go t)
  where
    go :: RBTree a -> RBTree a
    go E = N R E x E
    go t@(N c lt y rt)
      | x < y = balance c (go lt) y rt
      | x == y = t
      | x > y = balance c lt y (go rt)
```



```
blacken :: RBTree a -> RBTree a
blacken (N R lt x rt) = N B lt x rt   (just blackens the root node)
blacken t             = t
```

```
balance :: Colour -> RBTree a -> a -> RBTree a -> RBTree a
balance B (N R (N R llt y rlt) x rlt) z rt   (1)
  = N R (N B llt y rlt) x (N B rlt z rt)
balance B (N R llt y (N R lrlt x rrlt)) z rt   (2)
  = N R (N B llt y lrlt) x (N B rrlt z rt)
balance B lt z (N R (N R llrt x rlrt) y rrt)   (3)
  = N R (N B lt z llrt) x (N B rlrt y rrt)
balance B lt z (N R lrt x (N R lrrt y rrrt))   (4)
  = N R (N B lt z lrt) x (N B lrrt y rrrt)
```




* Cost of insert : - O(n log(n))
  - if list sorted : O(n) amortized

## Random

```
montePi :: Double
montePi = loop (mkStdGen 42) samples 0
  where
    loop :: StdGen -> Int -> Int -> Double
    loop seed 0 m = 4 * fromIntegral m / fromIntegral samples
    loop seed n m =
      let (x, seed') = randomR (0,1) seed
          (y, seed'') = randomR (0,1) seed'
          m' = if inside (x,y) then m+1 else m
          n' = n-1
      in loop seed'' n' m'

samples :: Int
samples = 10000
```

## Dynamic Programming

```
array :: Ix i => (i,i) -> [(i,a)] -> Array i a

tabulate :: Ix i => (i,i) -> (i -> a) -> Array i a
tabulate (u,v) f = array (u,v)
    [(i, f i) | i <- range (u,v)]
```

```
table :: Int -> Array Int Integer
table n = array (0,n) [(0,0), (1,1),
    (2, table!0 + table!1),
    (3, table!2 + table!2),
    ...]
```

## Treaps

```
data Treap a = Empty | Node (Treap a) a Int (Treap a)
```

(left child, value, priority, right child)

```
insert :: Ord a => a -> Int -> Treap a -> Treap a
insert x p Empty = Node Empty x p Empty
insert x p (Node a y q b)
  | x < y = lnode (insert x p a) q b
  | x == y = Node a y q b
  | x > y = rnode a y q (insert x p b)
```

```
lnode :: Treap a -> a -> Int -> Treap a -> Treap a
lnode Empty y q c = Node Empty y q c
lnode l@(Node a x p b) y q c
  | q ≤ p     = Node l y q c
  | otherwise = Node a x p (Node b y q c)
```

```
rnode :: Treap a -> a -> Int -> Treap a -> Treap a
rnode a x p Empty = Node a x p Empty
rnode a x p r@(Node b y q c)
  | p ≤ q = Node a x p r
  | otherwise = Node (Node a x p b) y q c
```

## Array Resizing

```
data ArrayList s a = ArrayList (STRef s Int) (STRef s Int) (STRef s (STArray s Int a))
newArray_ :: Ix i => (i,i) -> ST s (STArray s i a)

empty :: ST s (ArrayList s a)
empty = do pn <- newSTRef 0
           pm <- newSTRef m
           axs <- newArray_ (0, m-1)
           paxs <- newSTRef axs
           return (ArrayList pn pm paxs)
  where m = 8
```

```
toList :: ArrayList s a -> ST s [a]
toList (ArrayList rn rm raxs) = do
    n <- readSTRef rn
    m <- readSTRef rm
    axs <- readSTRef raxs
    sequence [readArray axs i |
                i <- [m-n .. m-1]]
```

```
reverse :: [Int] -> [Int]
reverse xs = runST $ do
    pxs <- empty
    sequence [insert x pxs | x <- xs]
    toList pxs
```

```
insert :: a -> (ArrayList s a) -> ST s ()
insert x (ArrayList pn pm paxs) = do
    n <- readSTRef pn
    m <- readSTRef pm
    axs <- readSTRef paxs
    writeSTRef pn (n+1)
    if n<m
      then do
        writeArray axs (m-n-1) x
      else do
        let m' = 2*m
        writeSTRef pm m'
        axs' <- newArray_ (0,m'-1)
        writeSTRef paxs axs'
        sequence [do x' <- readArray axs i
                     writeArray axs' (m+i) x'
                  | i <- [0..m-1]]
        writeArray axs' (m-1) x
```

## Mutable Datastructure

```
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()

runST :: (forall s. ST s a) -> a
```

```
fib' :: Int -> Integer
fib' n = runST $ do
    rx <- newSTRef 0
    ry <- newSTRef 1
    let loop 0 = do
          x <- readSTRef rx
          return x
        loop n = do
          x <- readSTRef rx
          y <- readSTRef ry
          writeSTRef rx y
          writeSTRef ry (x+y)
          loop (n-1)
    loop n
```

## ordered list -> balanced tree in O(n)

```
fromOrdList :: [a] -> HTree a
fromOrdList xs = fst (go (length xs) xs)
  where
    go :: Int -> [a] -> (HTree a, [a])
    go 0 xs = (HTip, xs)
    go n xs =
      let
        m = n `div` 2
        (lhs, x:xs2) = go m xs
        (rhs, xs3)   = go (n-m-1) xs2
      in (HNode (1+ max (height lhs) (height rhs))
                lhs x rhs, xs3)
    height HTip = 0
    height (HNode h _ _ _) = h
```

## Maximise SplitDiff

```
splitDiff :: [Int] -> Int -> Int
splitDiff xs i = sum lhs - sum rhs
  where (lhs, rhs) = splitAt i xs

maximise xs = maximumBy (comparing (splitDiff xs))
              [0..(length xs - 1)]
```

⇓

```
maximise xs = maximumBy (comparing (table!))
              [0..(length xs - 1)]
  where
    table = tabulate (0, length xs - 1) memo
    memo 0 = -sum xs
    memo i = table!(i-1) + xs!(i-1) * 2
    xs = fromList xs
```

## Random Access List

```
newtype RAList a = RAList [Tree a]
instance List RAList where
toList (RAList ts) = (concat . map toList) ts

RAList (t:ts) !! k
  | isEmpty t = RAList ts !! k
  | k < m = t !! k
  | otherwise = RAList ts !! (k-m)
    where m = length t

cons x xs = RAList (consTrees (Leaf x) xs)
where
  consTrees :: Tree a -> RAList a -> [Tree a]
  consTrees t (RAList []) = [t]
  consTrees t (RAList (Tip:ts)) = t : ts
  consTrees t (RAList (t':ts)) = Tip : consTrees
                                 (fork t t') (RAList ts)

tail = snd . split where
  split :: RAList a -> (Tree a, RAList a)
  split (RAList [t]) = (t, RAList [])
  split (RAList (Tip:ts)) = (t, RAList (t':ts'))
    where
      (Node t t', RAList ts') = split (RAList ts)
  split (RAList (t:ts)) = (t, RAList (Tip:ts))
```

```
newArray :: Ix i => (i,i) -> a -> ST s (STArray s i a)
readArray :: Ix i => STArray s i a -> i -> ST s a
writeArray :: Ix i => STArray s i a -> i -> a -> ST s (STArray s i a)
```

## Quicksort

```
swap :: STArray s Int a -> Int -> Int -> ST s ()
swap axs i j = do
    x <- readArray axs i
    y <- readArray axs j
    writeArray axs i y
    writeArray axs j x
```

```
qsort :: Ord a => [a] -> [a]
qsort xs = runST $ do
    axs <- newListArray (0,n) xs
    qsort axs 0 n
    getElems axs
  where n = length xs - 1
```

```
partition :: Ord a => STArray s Int a
          -> Int -> Int -> ST s Int
partition axs p q = do
  x <- readArray axs p
  let loop i j
        | i > j = do swap axs p j
                     return j
        | otherwise = do
            u <- readArray axs i
            if u < x
              then do loop (i+1) j
              else do swap axs i j
                      loop i (j-1)
  loop (p+1) q
```

```
qsort :: Ord a => STArray s Int a -> Int -> Int -> ST s ()
qsort axs i j
  | i ≥ j     = return ()
  | otherwise = do
      k <- partition axs i j
      qsort axs i (k-1)
      qsort axs (k+1) j
```

```
insert :: Int → [Int] → [Int]
insert x [ ] = [x]
insert x (y : ys)
  | x ⩽ y     = x : y : ys
  | otherwise = y : insert x ys
```

$$
\begin{aligned}
T_{insert}\ n &= 1 + T_{insert}\ (n-1) \\
&= 1 + 1 + T_{insert}\ (n-2) \\
&= \ldots \\
&= 1 + 1 + \ldots + T_{insert}\ (n-n) \\
&= 1 + n
\end{aligned}
$$

```
msort :: [Int] → [Int]
msort [ ]   = [ ]
msort [x]   = [x]
msort xs    = merge (msort us) (msort vs)
  where (us, vs) = splitAt (n `div` 2) xs
        n = length xs

merge :: [Int] → [Int] → [Int]
merge [ ] ys       = ys
merge xs [ ]       = xs
merge (x : xs) (y : ys)
  | x ⩽ y     = x : merge xs (y : ys)
  | otherwise = y : merge (x : xs) ys
```

$$
\begin{aligned}
T_{msort}(0) &= 1 \\
T_{msort}(1) &= 1 \\
T_{msort}(n) &= T_{length}(n) + T_{splitAt}(\tfrac{n}{2}) + T_{merge}(\tfrac{n}{2}) + 2 \times T_{msort}(\tfrac{n}{2})
\end{aligned}
$$

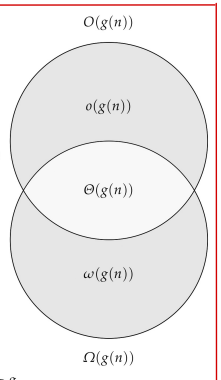
O(n log n)

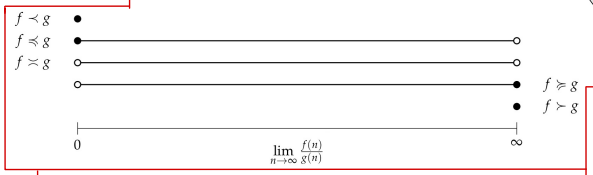
```
qsort :: [Int] → [Int]
qsort [ ]       = [ ]
qsort [x]       = [x]
qsort (x : xs)  = qsort us ++ [x] ++ qsort vs
  where (us, vs) = partition (<x) xs

partition :: (a → Bool) → [a] → ([a], [a])
partition p xs = (filter p xs, filter (¬ ∘ p) xs)
```

$$
\begin{aligned}
T_{qsort}(n) &= T_{partition}(n-1) + T_{++}(n-1) + T_{qsort}(n-1) + T_{qsort}(0) \\
&= c \times n + T_{qsort}(n-1) \\
&= c \times n + c \times (n-1) + \ldots + c \times 1 \\
&= c \times \frac{n \times (n+1)}{2} \\
&= c \times \frac{n^2 + n}{2}
\end{aligned}
$$

O(n²)

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




$f \prec g \iff g \succ f$ ($\prec, \succ$ are converse)

transitive:

$f \prec g \wedge g \prec h \Rightarrow f \prec h$

$f \preccurlyeq g \wedge g \preccurlyeq h \Rightarrow f \preccurlyeq h$

$$1 \prec \log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec n^3 \prec n! \prec n^n$$

$$
\begin{aligned}
o(g(n)) &= \{\ f \mid \forall \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) < \delta g(n)\ \} \\
O(g(n)) &= \{\ f \mid \exists \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) \leqslant \delta g(n)\ \} \\
\Theta(g(n)) &= O(g(n)) \cap \Omega(g(n)) \\
\Omega(g(n)) &= \{\ f \mid \exists \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) \geqslant \delta g(n)\ \} \\
\omega(g(n)) &= \{\ f \mid \forall \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) > \delta g(n)\ \}
\end{aligned}
$$

```
fib :: Int → Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n − 1) + fib (n − 2)
```

O(2ⁿ)

$$T_{fib}(n) \leqslant 1 + 2 \times T_{fib}(n-1)$$

```
fib' :: Int → Integer
fib' n = table ! n
  where
    table :: Array Int Integer
    table = tabulate (0, n) memo
    memo 0 = 0
    memo 1 = 1
    memo n = table ! (n − 1) + table ! (n − 2)
```

O(n)

$$T_{fib'}(n) = 1 + T_{table}(n) + T_!(n)$$

```
data Deque a = Deque [a] [a]

instance List Deque where
  toList :: Deque a → [a]
  toList (Deque xs sy) = xs ++ reverse sy
  fromList xs = Deque ys (reverse zs)
    where (ys, zs) = splitAt (length xs `div` 2) xs
  tail :: Deque a → Deque a
  tail (Deque [ ]  [ ])  = error "tail: empty list"
  tail (Deque [ ]  sy)   = empty
  tail (Deque [x]  sy)   = fromList (reverse sy)
  tail (Deque (x : xs) sy) = Deque xs sy
  cons :: a → Deque a → Deque a
  cons x (Deque xs [ ]) = Deque [x] xs
  cons x (Deque xs sy)  = Deque (x : xs) sy
  snoc :: Deque a → a → Deque a
  snoc (Deque [ ] sv) x = Deque sv [x]
  snoc (Deque us sv) x = Deque us (x : sv)
```

e ::= x – variables
| k – constants
| f e₁ . . eₙ – applications
| if e then e₁ else e₂ – conditionals

**Applicative order**: leftmost innermost reducible expr when evaluating (args 1st)
**Normal order**: leftmost outermost reducible expr when evaluating (func 1st)
**Primitive cost**: $T(f)x_1 \ldots x_n = 0$
**Variable cost**: $T(x) = 0$
**Application cost**: $T(fe_1 \ldots e_n) = T(f)e_1 \ldots e_n + T(e_1) + \ldots + T(e_n)$
**Conditional cost**: $T(\text{if p then } e_1 \text{ else } e_2) = T(p) +$ if p then $T(e_1)$ else $T(e_2)$

We set $A_{op}(xs) = 2$ for each *op* and use the same cost function $C$ and size function $S$ as in the lecture:

$$C_{cons}(xs) = 1 \qquad C_{snoc}(xs) = 1$$

$$C_{tail}(Deque\ xs\ sy) = \textbf{if}\ length\ xs > 1\ \textbf{then}\ 1\ \textbf{else}\ length\ sy$$

$$S(Deque\ xs\ sy) = |length\ xs - length\ sy|$$

so it remains to show

$$C_{op_i}(xs_i) \leqslant A_{op_i}(xs_i) + S(xs_i) - S(xs_{i+1})$$

for $op_i \in \{cons, snoc, tail\}$. The case for *tail* can be found in the lecture notes. For $op_i = snoc$, we have the definition

```
snoc :: Deque a → a → Deque a
snoc (Deque [ ] sv) x = Deque sv [x]
snoc (Deque us sv) x = Deque us (x : sv)
```

and $C_{snoc}(xs) = 1$ and $A_{snoc}(xs) = 2$. If $xs_i$ matches the first pattern *Deque* [ ] *sv* of *snoc*,

$$
\begin{aligned}
& A_{snoc}(xs_i) + S(xs_i) - S(xs_{i+1}) \\
&= 2 - S(Deque\ sv\ [x]) + S(Deque\ [\,]\ sv) \\
&= 2 - |length\ sv - 1| + |length\ sv| \\
&\geqslant \quad \{|a-b| \leqslant |a| + |b|\} \\
&\quad 2 - 1 - |length\ sv| + |length\ sv| \\
&= 2 - 1 = 1 = C_{snoc}(xs_i)
\end{aligned}
$$

If $xs_i$ matches the second pattern *Deque us sv* of *snoc*,

$$
\begin{aligned}
& A_{snoc}(xs_i) + S(xs_i) - S(xs_{i+1}) \\
&= 2 - S(Deque\ us\ (x:sv)) + S(Deque\ us\ sv) \\
&= 2 - |length\ sv + 1 - length\ us| + |length\ sv - length\ us| \\
&\geqslant \quad \{|a+b| \leqslant |a| + |b|\} \\
&\quad 2 - 1 - |length\ sv - length\ us| + |length\ sv - length\ us| \\
&= 2 - 1 = 1 = C_{snoc}(xs_i)
\end{aligned}
$$

The case for *cons* is completely symmetric.

```
instance List DList where
  -- toList: O(n)
  toList :: DList a → [a]
  toList (DList dxs) = dxs [ ]
  -- fromList: O(1)
  fromList :: [a] → DList a
  fromList xs = DList (xs++)
  -- (++): O(1)
  (++) :: DList a → DList a → DList a
  DList dxs ++ DList fys = DList (dxs ∘ fys)
  -- empty: O(1)
  empty :: DList a
  empty = DList (λxs → xs)
  -- single x: O(1)
  single :: a → DList a
  single x = DList (x:)
  -- cons x xs: O(1)
  cons :: a → DList a → DList a
  cons x (DList dxs) = DList ((x:) ∘ dxs)
  -- snoc xs x: O(1)
  snoc :: DList a → a → DList a
  snoc (DList dxs) x = DList (dxs ∘ (x:))
  -- head xs: O(n), where n = length xs
  head :: DList a → a
  head xs = head (toList xs)
  -- tail xs: O(n), where n = length xs
  tail :: DList a → DList a
  tail xs = fromList (tail (toList xs))
  -- init xs: O(n), where n = length xs
  init :: DList a → DList a
  init xs = fromList (init (toList xs))
  -- last xs: O(n), where n = length xs
  last :: DList a → a
  last xs = last (toList xs)
  -- isEmpty xs: O(n), where n = length xs
  isEmpty :: DList a → Bool
  isEmpty xs = isEmpty (toList xs)
  -- isSingle xs: O(n), where n = length xs
  isSingle :: DList a → Bool
  isSingle xs = isSingle (toList xs)
  -- length xs: O(n), where n = length xs
  length :: DList a → Int
  length xs = length (toList xs)
  -- xs !! i: O(n), where n = length xs
  (!!) :: DList a → Int → a
  xs !! n = toList xs !! n
```

O(n) inter2 :: [Int] → [Int] → Int
inter2 xs ys = length xs + length ys

O(n²) inter2 :: [Int] → [Int] → Int
inter2 (x:xs) ys
  | xs == [ ] = 0
  | otherwise = length ys + inter2 xs ys

**Abstract Lists**

**1. Default**

```
instance List list where
... - toList, fromList, length, (++), (:)
normalise = fromList . toList
foldr f k [] = k
foldr f k (x:xs) = f x (foldr f k xs)
foldl f k [] = k
foldl f k (x:xs) = foldl f (f k x ) xs
-- top-down approach
foldt [x] = x
foldt f xs = f (foldt f ys) (foldt f zs)
  where (ys, zs) = splitAt (length xs`div` 2) xs
-- bottom-up approach
foldt f [x] = x
foldt f xs = foldt f (pMap f xs)
  where
    pMap f (x1 : x2 : xs) = f x1 x2 : pMap f xs
    pMap xs = xs
```

**2. Tree Lists** -- only useful if ordered/balanced

```
data Tree a = Tip | Leaf a | Fork (Tree a) (Tree a)
fromTree :: Tree a -> [a]  -- Bottom Up
fromTree t = helper t []
 where helper :: Tree a -> [a] -> [a]
 helper (Leaf x) xs = (x:xs)
 helper (Node l r) xs = helper l (helper r xs)
```