LECTURE 14 : Randomized Algorithms

"The assumption of an absolute determinism is the essential foundation of every scientific enquiry."

— Max Planck, 1858–1947

```
main :: IO ()
main =
  do print "Hello World!"
     print "Hello again
```

```
printRandom :: IO ()
printRandom =
  do x <- getRandom
     print (x + 42)
```

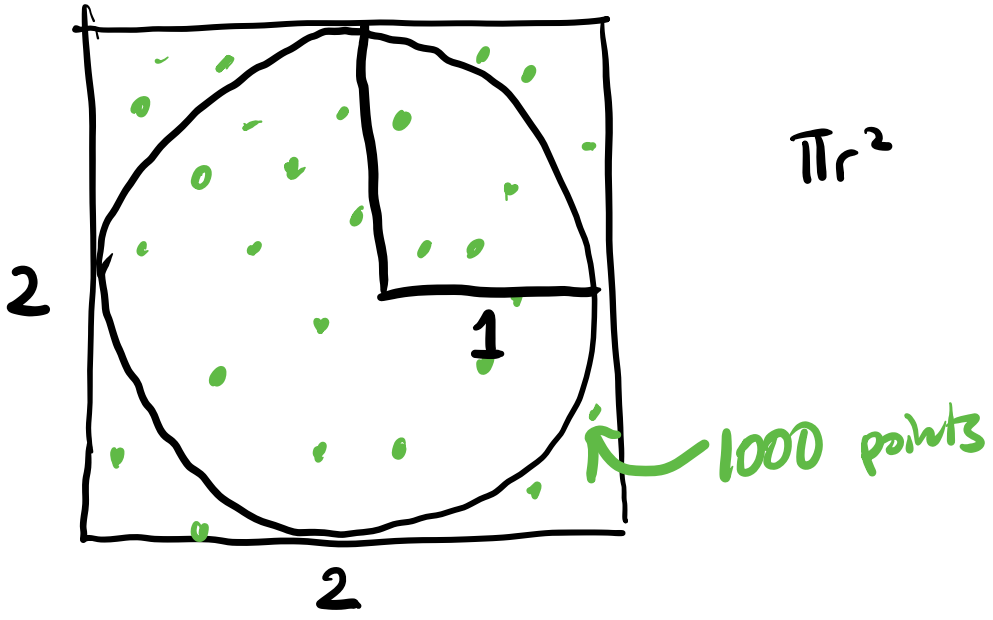
← too specific we use:

```
printRandom :: MonadRandom m
  => m ()
```

↑ IO works, but so do other things.

- Monte Carlo Algorithms — known time, probable answer
- Las Vegas Algorithms — probable time, known answer.

We can use a Monte Carlo algorithm to approximate the value of π.



πr^2

```
montePi :: MonadRandom m => m Double
montePi = loop 1000 0
  where
    loop 0 m = return (4 * fromIntegral m / fromIntegral 1000)
    loop n m = do
      x <- getRandomR (0,1)
      y <- getRandomR (0,1)
      let n' = n-1
      let m' = if inside (x,y)
               then m+1
               else m
      loop n' m'
```

how many iterations are left

how many values are in the circle

gets a random value between (0...1)

```
printPi :: IO ()
printPi = do
  pi <- montePi
  print pi
```

```
montePi >>= λpi → print pi

montePi >>= print
```

fancy, but not needed.

```
> class Monad m ⇒ MonadRandom m where
>     getRandom :: Random a ⇒ m a
>     getRandoms :: Random a ⇒ m [a]
>
>     getRandomR :: Random a ⇒ (a,a) → m a
>     getRandomRs :: Random a ⇒ (a,a) → m [a]
```

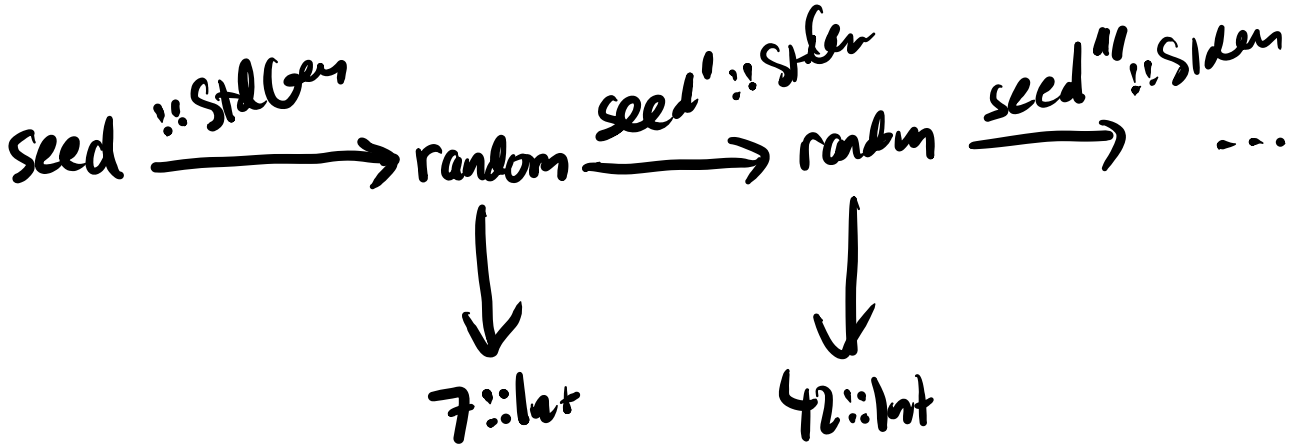
min range, max range

Random functions do not exist. This is a consequence of "the indescernability of identicals":

$$x = y \Rightarrow fx = fy \quad \text{for all functions } f.$$

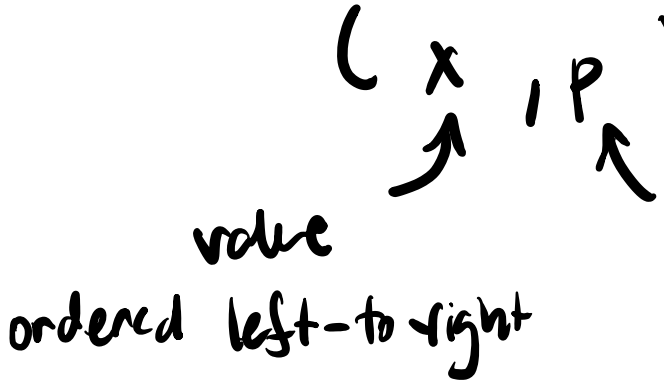
So where do Random values come from?

```
> class Random a where
>     random :: StdGen -> (a, StdGen)
>     randoms :: StdGen -> [a]
>
>     randomR :: (a,a) -> StdGen -> (a, StdGen)
>     randomRs :: (a,a) -> StdGen -> [a]
```

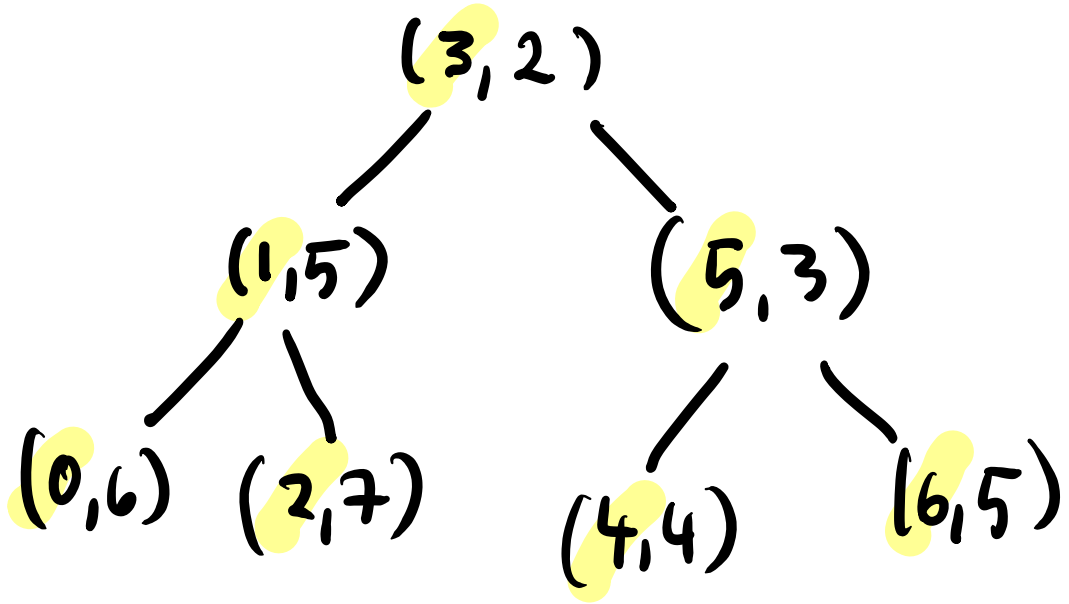


Treaps.

A Treap is both a Tree and a Heap.
This structure contains pairs:



Example:



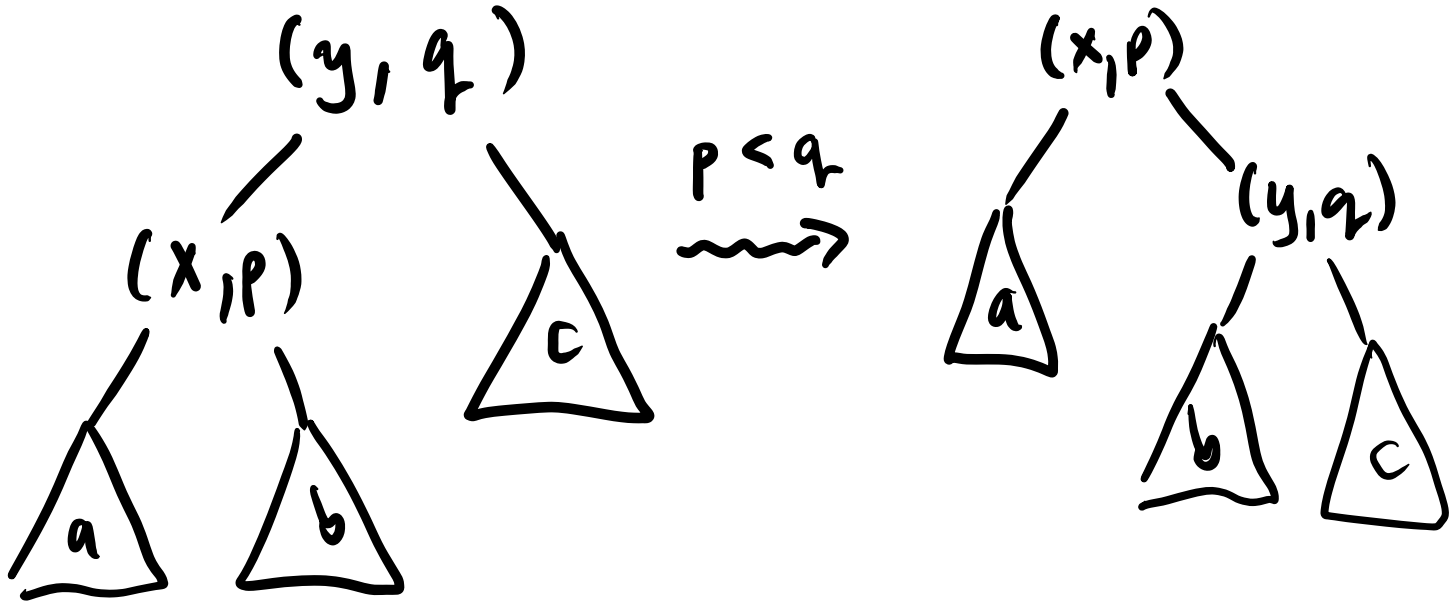
value
priority.

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

```
> member :: Ord a => a -> Treap a -> Bool.
   -- implemented in the "usual" way,
   -- searching based on value.
```

value
priority.

```
>pinsert :: Ord a => a -> Int -> Treap a -> Treap a
>pinsert x p Empty = Node Empty x p Empty
>pinsert x p t@(Node l y q r)
>     | x == y  = t
>     | x < y   = lnode (insert x p l) y q r
>     | x > y   = rnode l y q (insert x p r)
```




```
> 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
>
>   | p < q =
>       Node a x p (Node b y q c)
>
>   | otherwise = Node l y q c
```

Next lecture we will see how to generate random priorities to insert values into the treap.

Also: why delete / merge are easy.