Generating random tests

Beijing 2023
Autumn School

John Hughes

CHALMERS QuviQ

arbitrary

shrink

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

Given **a**, *construct a list of* ***simpler*** *test cases which* **a** *may shrink to*

The ***generator*** *type; generates a random* **a**

The Greedy Shrinking Search

*Placing **large** shrinking steps early makes for efficient shrinking*

- Given a failing test case **`x`**,
  - Replace **`x`** by the *first* element **`y`** of **`shrink x`** for which the test fails, and repeat
  - If no element of **`shrink x`** makes the test fail, report **`x`** as the shrunk test case

*QuickCheck runs the test for **every** element of **`shrink x`** before reporting the failure. Better not be too long!*

Type modifiers

- The *type* determines generation and shrinking

- We can *change* generation and shrinking by using a newtype

```
newtype NonEmptyList a = NonEmpty [a]
```

`Arbitrary` *instance generates and shrinks to non-empty lists*

```
newtype Positive a     = Positive a
```

`Arbitrary` *instances for numeric types generate and shrink to positive numbers*

Testing square root

```
prop_Sqrt :: Double -> _
prop_Sqrt x =
  x >= 0 ==> sq (sqrt x) =~= x

sq x = x*x

prop_Sqrt' :: (NonNegative Double) -> _
prop_Sqrt' (NonNegative x) =
  sq (sqrt x) =~= x
```

Approximately equal to

The precondition discards half the tests!

The test is written very similarly

Only non-negative test cases are generated

Define an **`Arbitrary`** instance…

- To make your own types usable in properties

- To specify *different* generation and shrinking for an existing type

The **`Gen`** type: useful combinators

- **`choose (m,n)`**
  Generate a value uniformly in the range **`m`** to **`n`**

- **`elements xs`**
  Generate an element of **`xs`**, chosen uniformly

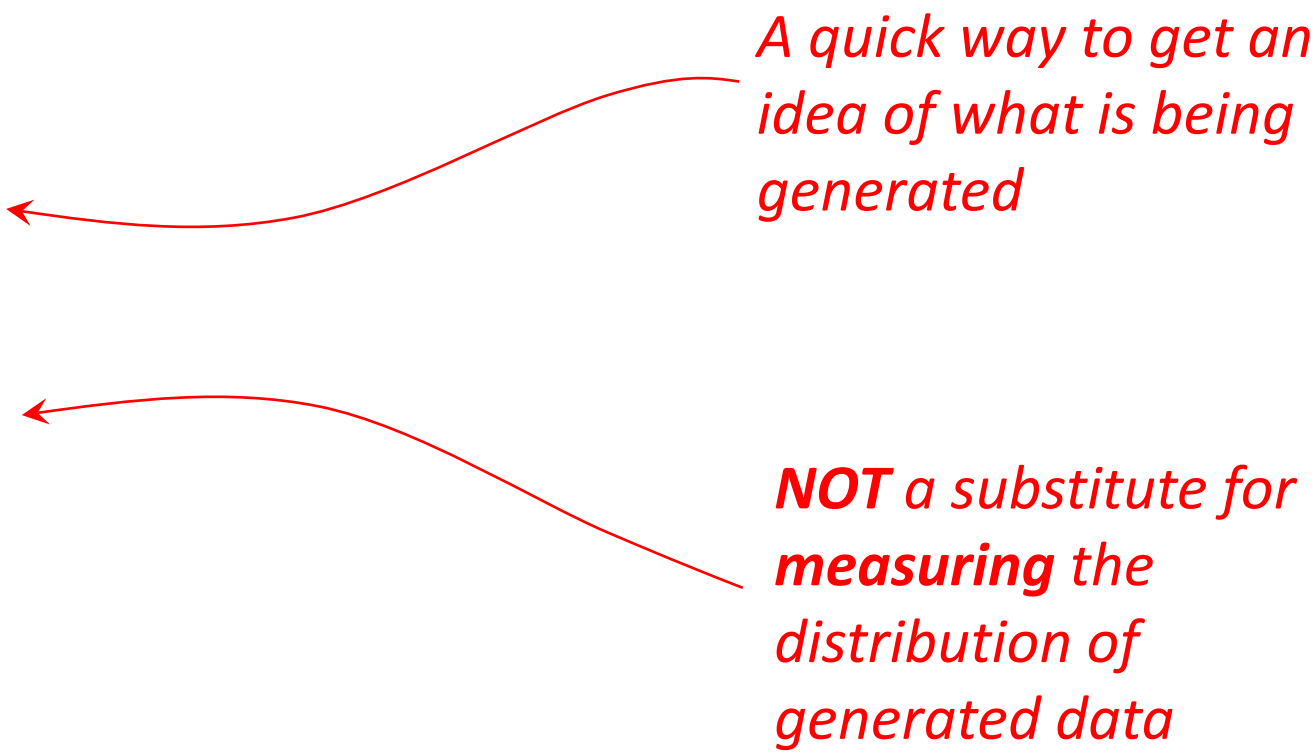
- **`oneof [gen1, …, genN]`**
  A uniform choice between generators

- **`frequency [(weight1,gen1),…,(weightN,genN)]`**
  A weighted choice between generators (with integer weights)

monadic
\+ *operators;*
do-*syntax*

Sampling generators

```
*Sqrt> sample (elements "hello")
'l'
'h'
'o'
'e'
'e'
'l'
'e'
'e'
'l'
'e'
'l'
```



Sampling at a type

```
Test.QuickCheck> sample (arbitrary :: Gen [Int])
[]
[0,0]
[-3,-3,3,0]
[5,5]
[5,-5,5]
[-2,-4,0,-6,-8,-4]
[-1,-8]
[-13]
[-7]
[10,-9,-5,6,14,0]
[]
```

Example: ordered list insertion

```
propInsert :: Int -> [Int] -> Bool
propInsert x xs = ordered (insert x xs)
  where ordered xs = sort xs == xs
```

```
*Examples> quickCheck propInsert
*** Failed! Falsified (after 6 tests and 6 shrinks):
0
[1,0]

*Examples> insert 0 [1,0]
[0,1,0]
```

Example: using a precondition

```
propInsert :: Int -> [Int] -> Property
propInsert x xs =
  ordered xs ==> ordered (insert x xs)
```

```
*Examples> quickCheck propInsert
*** Gave up! Passed only 71 tests; 1000 discarded
tests.
```

Example: using a generator

```
propInsert :: Int -> Ordered Int -> Bool
propInsert x (Ordered xs) =
  ordered (insert x xs)

data Ordered a = Ordered [a] deriving (Eq, Show)

instance (Ord a, Arbitrary a) =>
           Arbitrary (Ordered a) where
  arbitrary = do xs <- arbitrary
                 return (Ordered (sort xs))
  shrink (Ordered xs) =
    [Ordered ys | ys <- shrink xs, ordered ys]

*Examples> quickCheck propInsert
+++ OK, passed 100 tests.
```

Recall BSTs and their invariant

```
valid :: Ord k => BST k v -> Bool
valid Leaf = True
valid (Branch l k v r) =
  valid l && valid r &&
  all (<k) (keys l) && all (>k) (keys r)
```

```
prop_NilValid           = valid (nil :: Tree)
prop_InsertValid k v t = valid (insert k v t)
prop_DeleteValid k t    = valid (delete k t)
prop_UnionValid t t'    = valid (union t t')
```

A generator and shrinker for BSTs

```
instance (Ord k, Arbitrary k, Arbitrary v) =>
    Arbitrary (BST k v) where

  -- generator
  arbitrary =
    do kvs <- arbitrary
       return $
         foldr (\(k,v) t -> insert k v t)
               nil
               (kvs :: [(k,v)])

  -- shrinker
  shrink = genericShrink
```

Generate by inserting random key-value pairs

Shrink using a generic QuickCheck mechanism

insert

```
=== prop_InsertValid from BSTSpec.hs:19 ===
*** Failed! Falsified (after 6 tests and 8 shrinks):
0
0
Branch Leaf 0 0 Leaf

=== prop_DeleteValid from BSTSpec.hs:22 ===
*** Failed! Falsified (after 8 tests and 7 shrinks):
0
Branch Leaf 1 0 (Branch Leaf 0 0 Leaf)

=== prop_UnionValid from BSTSpec.hs:25 ===
*** Failed! Falsified (after 7 tests and 9 shrinks):
Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)
Leaf
```

insert

```
=== prop_InsertValid from BSTSpec.hs:19 ===
*** Failed! Falsified (after 6 tests and 8 shrinks):
0
0
Branch Leaf 0 0 Leaf

=== prop_DeleteValid from BSTSpec.hs:22 ===
*** Failed! Falsified (after 8 tests and 7 shrinks):
0
Branch Leaf 1 0 (Branch Leaf 0 0 Leaf)

=== prop_UnionValid from BSTSpec.hs:25 ===
*** Failed! Falsified (after 7 tests and 9 shrinks):
Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)
Leaf
```

Testing our tests

```
prop_ArbitraryValid t = valid t
```

```
prop_ShrinkValid t =
  all valid (shrink t)
```

```
Branch Leaf 0 0 (Branch Leaf 0 1 Leaf)
➔ Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)
```

Testing our tests

```
prop_ArbitraryValid t = valid t
```

```
prop_ShrinkValid t =
  all valid (shrink t)
```

```
Branch Leaf 0 0 (Branch Leaf 0 1 Leaf)
➔ Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)
```

Testing our tests

```
prop_ArbitraryValid t = valid t
```

```
prop_ShrinkValid t =
  all valid (shrink t)
```

```
Branch Leaf 0 0 (Branch Leaf 0 1 Leaf)
```

➔ `Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)`

Testing our tests

```
prop_ArbitraryValid t = valid t
```

```
prop_ShrinkValid t =
  valid t ==> all valid (shrink t)
```

```
Branch Leaf 0 0 (Branch Leaf 1 0 Leaf)
➔ Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)
```

Fixing shrinking

```
shrink = genericShrink.filter isValid.
```

Completeness: can every tree be constructed just using **`insert`**?

```
insertions Leaf = []
insertions (Branch l k v r) =
   (k,v):insertions l++insertions r

prop_InsertComplete t =
  t
  ===
  foldl (\t (k,v)-> insert k v t) nil (insertions t)

prop_InsertCompleteForDelete k t =
  prop_InsertComplete (delete k t)

prop_InsertCompleteForUnion t t' =
  prop_InsertComplete (union t t')
```

Generating trees the hard way

```
data BST k v = Leaf
             | Branch (BST k v) k v (BST k v)
  deriving (Eq, Show, Generic)
```

```
instance (Arbitrary k, Arbitrary v) =>
           Arbitrary (BST k v) where

  arbitrary = oneof [return Leaf,
                     do (l,k,v,r) <- arbitrary
                        return (Branch l k v r)]

  shrink = …
```

*Examples> sample (arbitrary :: Gen (BST Int Int))

Leaf

Branch (Branch (Branch (Branch (Branch Leaf 1 2 Leaf) 0 1 (Branch (Branch Leaf 2 2 (Branch Leaf (-1) 2 Leaf)) (-1) 0 Leaf)) (-1) (-2) Leaf) (-2) 1 Leaf) 0 (-1) (Branch Leaf (-1) 1 (Branch Leaf 2 (-1) (Branch (Branch Leaf 2 1 Leaf) (-1) 1 (Branch Leaf (-1) 1 (Branch Leaf 0 1 (Branch Leaf 0 (-2) Leaf))))))

Branch (Branch Leaf (-2) (-1) Leaf) (-1) 4 (Branch Leaf (-2) (-4) Leaf)

Branch (Branch (Branch Leaf (-1) (-2) Leaf) (-1) (-5) (Branch (Branch (Branch Leaf 6 (-4) (Branch Leaf 6 3 Leaf)) 6 (-2) Leaf) 3 3 (Branch (Branch (Branch (Branch (Branch (Branch Leaf 4 (-2) (Branch (Branch (Branch (Branch Leaf 0 (-6) Leaf) (-1) 5 (Branch Leaf (-1) (-1) (Branch Leaf (-4) (-6) Leaf))) 1 1 (Branch Leaf 1 3 (Branch Leaf 1 (-1) Leaf))) 2 (-1) Leaf)) (-5) 3 (Branch (Branch (Branch (Branch (Branch Leaf 4 5 Leaf) 6 5 (Branch (Branch Leaf 6 (-3) Leaf) 3 (-1) Leaf)) (-5) 1 (Branch Leaf (-5) (-6) (Branch Leaf 5 1 (Branch Leaf 6 (-4) (Branch (Branch Leaf 3 5 (Branch Leaf (-3) 2 Leaf)) (-6) (-2) Leaf))))) 0 (-3) (Branch Leaf 4 (-2) (Branch Leaf 3 6 Leaf))) (-2) 5 Leaf)) (-6) (-1) Leaf) (-4) (-5) Leaf) (-2) (-1) (Branch (Branch Leaf 6 (-4) (Branch (Branch Leaf (-2) (-2) (Branch (Branch Leaf 2 (-5) Leaf) (-4) (-1) (Branch (Branch (Branch Leaf 0 5 Leaf) (-2) (-1) Leaf) 6 3 Leaf))) (-2) (-6) (Branch Leaf 2 1 (Branch (Branch (Branch Leaf (-2) 0 (Branch Leaf (-1) (-1) Leaf)) (-3) 1 Leaf) 6 (-2) Leaf)))) (-5) 2 (Branch Leaf (-3) 3 (Branch Leaf (-1) (-6) Leaf)))) 5 (-1) Leaf))) (-2) 3 Leaf

Leaf

Leaf

Branch (Branch (Branch (Branch Leaf (-12) (-5) (Branch (Branch (Branch (Branch Leaf (-3) 12 Leaf) (-1) 3 Leaf) 6 (-1) Leaf) (-11) 1 (Branch (Branch Leaf (-9) (-8) Leaf) 9 9 Leaf))) 3 (-6) Leaf) (-11) 2 (Branch (Branch (Branch Leaf (-4) 10 Leaf) 8 0 Leaf) 6 2 (Branch (Branch Leaf 0 6 Leaf) 4 8 Leaf))) 12 12 (Branch (Branch (Branch (Branch Leaf 0 7 Leaf) (-11) (-7) Leaf) (-6) 2 Leaf) 9 12 (Branch (Branch (Branch Leaf 1 (-10) Leaf) (-11) 5 Leaf) (-11) 5 (Branch Leaf 5 (-9) (Branch Leaf (-9) 4 (Branch Leaf (-6) (-6) Leaf)))))

Leaf

Leaf

Leaf

Leaf

Way too large test cases

Way too many leaves

Reducing leaves

```
instance (Arbitrary k, Arbitrary v) =>
         Arbitrary (BST k v) where
  arbitrary = frequency
                [(1,return Leaf),
                 (2,do (l,k,v,r) <- arbitrary
                       return (Branch l k v r))]
```

```
*Examples> sample (arbitrary :: Gen (BST Int Int))
Leaf
Branch Leaf 1 1 Leaf
Branch (Branch (Branch (Branch (Branch (Branch Leaf 4 1 Leaf) (-2) 3 (Branch (Branch Leaf
0 4 (Branch Leaf 3 3 Leaf)) 1 2 (Branch (Branch Leaf 1 (-4) (Branch (Branch (Branch (Branch Leaf (-4) (-1) Leaf) (-2) 2 (Branch Leaf (-3) 0 (Branch (Branch (Branch (Branch (Branch (Branch (Branch
(Branch (Branch Leaf 4 (-2) (Branch (Branch (Branch (Branch (Branch (Branch Leaf (-1) (-1) (Branch (Branch (Branch (Branch (Branch Leaf (-2) 3 Leaf) (-2) (-4) (Branch Leaf 4 2 (Branch (Branch (Branch
(Branch (Branch Leaf (-1) 4 Leaf) 4 0 Leaf) 0 (-4) Leaf) 2 (-3) (Branch (Branch (Branch (Branch (Branch (Branch (Branch Leaf (-2) 3 Leaf) 3 3 (Branch (Branch (Branch Leaf (-2) 3 (Branch
Leaf 0 2 (Branch Leaf 1 (-1) Leaf))) 0 (-2) Leaf) (-2) (-1) Leaf)) 0 (-4) Leaf) (-1) 2 (Branch Leaf (-3) 1 (Branch (Branch Leaf 2 (-1) (Branch Leaf 3 2 Leaf)) (-2) (-1) (Branch Leaf 3 (-1) (Branch
(Branch Leaf (-3) (-2) (Branch (Branch (Branch Leaf (-4) 1 (Branch (Branch (Branch (Branch Leaf 2 (-4) (Branch Leaf (-1) 0 (Branch Leaf 2 (-2) (Branch Leaf (-4) 3 (Branch Leaf (-2) (-2) (Branch
(Branch (Branch Leaf 1 2 (Branch (Branch Leaf (-1) (-1) (Branch (Branch Leaf 3 (-3) (Branch Leaf 0 2 (Branch (Branch (Branch (Branch (Branch Leaf 1 0 Leaf) 2 (-4) (Branch Leaf (-3) 2 (Branch (Branch
(Branch (Branch Leaf 4 1 (Branch (Branch Leaf 4 0 Leaf) 1 1 (Branch (Branch (Branch Leaf 3 (-3) (Branch (Branch (Branch Leaf (-3) 0 (Branch Leaf 0 3 (Branch (Branch Leaf 3 2 (Branch (Branch Leaf (-1)
(-3) (Branch (Branch (Branch (Branch (Branch (Branch Leaf (-3) 3 (Branch (Branch (Branch (Branch Leaf 0 (-1) (Branch (Branch Leaf 1 (-4) (Branch (Branch (Branch (Branch (Branch Leaf 1 1 (Branch (Branch (Branch
(Branch (Branch (Branch (Branch (Branch Leaf 2 (-2) (Branch (Branch (Branch Leaf 1 (-4) (Branch Leaf 3 (-2) (Branch (Branch Leaf 0 (-2) Leaf) 0 4 (Branch (Branch (Branch (Branch Leaf (-4) (-1)
(Branch Leaf 2 (-3) (Branch (Branch Leaf (-1) 1 Leaf) 3 (-3) (Branch (Branch Leaf (-1) 2 (Branch (Branch (Branch (Branch (Branch Leaf (-1) (-1) (Branch Leaf 0 0 (Branch (Branch Leaf (-1) (-1)
(Branch (Branch Leaf (-3) (Branch (Branch (Branch (Branch (Branch Leaf 2 (-1) Leaf) (-2) (-1) (Branch (Branch (Branch (Branch Leaf 2 (Branch (Branch Leaf (-2) 2) Leaf) 3 4
Leaf)) (-4) (-1) Leaf) (-2) (-1) Leaf) 1 (-1) (Branch Leaf (-3) (-2) Leaf)) 1 (Branch Leaf (-2) (-3) Leaf)) 3 (-4) (Branch Leaf 2 3 Leaf)) (-2) 0 (Branch (Branch Leaf 2 2 Leaf) 0 2 (Branch (Branch
(Branch Leaf (-4) 4 Leaf) 4 (-2) Leaf) 1 1 Leaf)) (-2) 2 Leaf) (-1) (Branch Leaf (-1) Leaf) (-4) Leaf)) 4 (-2) (Branch (Branch (Branch (Branch Leaf 1 (-4) (Branch (Branch Leaf (-1) 0 (Branch
(Branch Leaf 0 (-3) (Branch (Branch (Branch (Branch (Branch (Branch (Branch (Branch Leaf 0 4 (Branch Leaf (-4) 1 (Branch Leaf 3 (-3) (Branch Leaf 3 (-2) (Branch (Branch (Branch Leaf (-3) 0 (Branch Leaf
(-4) 2 (Branch Leaf (-3) (-3) Leaf))) (-3) (-2) Leaf) (-4) (-1) (Branch (Branch (Branch (Branch (Branch Leaf 4 3 Leaf) 3 0 (Branch (Branch (Branch (Branch (Branch (Branch (Branch (Branch Leaf 0
(-1) (Branch (Branch Leaf (-1) (-3) Leaf) 0 (-3) (Branch (Branch Leaf (-3) (-3) (Branch (Branch Leaf (-3) 4 Leaf) 2 2 (Branch (Branch Leaf (-3) 1 (Branch (Branch (Branch Leaf (-1) 0 (Branch (Branch
(Branch (Branch (Branch (Branch Leaf 1 4 (Branch (Branch (Branch Leaf 3 3 (Branch Leaf (-1) (-3) (Branch (Branch (Branch (Branch Leaf (-3) (-2) (Branch (Branch (Branch Leaf 2 (-3) Leaf) 0 (-3)
Leaf) 0 3 (Branch (Branch (Branch (Branch (Branch Leaf 0 4 (Branch (Branch (Branch Leaf 3 3 (Branch (Branch Leaf (-2) (-4) Leaf) 3 (-2) (Branch Leaf 3 (-3) Leaf))) (-3) (-1) (Branch (Branch (Branch
(Branch Leaf 3 0 (Branch (Branch (Branch (Branch (Branch (Branch (Branch Leaf (-2) (-3) Leaf) (-3) 0 (Branch (Branch (Branch (Branch Leaf 3 (-1) (Branch (Branch Leaf 3 4 (Branch (Branch
(Branch (Branch Leaf (-4) 2 Leaf) 2 (-2) Leaf) 0 (-2) Leaf) 1 (-3) (Branch Leaf 1 4 (Branch (Branch (Branch (Branch (Branch (Branch (Branch (Branch (Branch (Branch (Branch Leaf
3 4 (Branch (Branch (Branch (Branch (Branch (Branch (Branch Leaf (-1) 2 (Branch (Branch (Branch Leaf 1 4 Leaf) 3 4 (Branch Leaf 0 2 Leaf)) 1 (-2) (Branch (Branch (Branch (Branch Leaf 0 0 Leaf)
(-3) 4 Leaf) 1 4 Leaf) 4 2 (Branch (Branch (Branch (Branch (Branch Leaf (-2) 2 (Branch (Branch (Branch (Branch Leaf (-4) 2 (Branch (Branch (Branch (Branch (Branch (Branch Leaf (-3) 4 (Branch
(Branch Leaf 4 4 (Branch (Branch (Branch (Branch (Branch Leaf 0 (-1) Leaf) (-2) (-4) Leaf) (-2) 4 Leaf) (-1) (-2) Leaf) 4 2 (Branch Leaf (-1) (-3) Leaf))) 3 1 (Branch (Branch Leaf (-2) 3 Leaf) 0 (-1)
```

50% probability of termination

Sized generators

```
instance (Arbitrary k, Arbitrary v) =>
           Arbitrary (BST k v) where
  arbitrary = sized tree
    where
      tree 0 = return Leaf
      tree n = frequency
                 [(1,return Leaf),
                  (2,do (k,v) <- arbitrary
                        l <- child
                        r <- child
                        return (Branch l k v r))]
        where child = tree ((n-1) `div` 2)
```

sized gen generates a **gen n**, for some 0 <= **n** <= 100.

```
*Examples> sample (arbitrary :: Gen (BST Int Int))
Leaf
Branch Leaf 0 (-1) Leaf
Leaf
Branch Leaf 0 2 Leaf
Branch (Branch Leaf (-5) 3 Leaf) 2 (-8) (Branch Leaf 4 2
(Branch Leaf 1 (-4) Leaf))
Branch (Branch (Branch Leaf (-6) (-1) Leaf) 9 9 Leaf) 9 (-
5) Leaf
Branch (Branch Leaf (-12) 2 Leaf) (-10) 10 (Branch (Branch
Leaf 0 (-2) Leaf) 4 (-5) (Branch Leaf (-4) 1 Leaf))
Branch Leaf 4 14 Leaf
Leaf
Branch (Branch Leaf 11 (-4) (Branch (Branch Leaf 17 (-10)
Leaf) (-3) 15 (Branch Leaf 4 (-12) Leaf))) 6 (-10) (Branch
(Branch (Branch Leaf (-6) 11 Leaf) 3 4 (Branch Leaf (-16)
(-7) Leaf)) (-5) (-8) Leaf)
Branch (Branch (Branch (Branch Leaf 6 4 Leaf) 8 (-7)
(Branch Leaf (-16) (-3) Leaf)) 8 (-7) Leaf) 10 12 Leaf
```

BUT...

```
*Examples> quickCheck prop_ArbitraryValid
*** Failed! Falsified (after 4 tests):
Branch (Branch Leaf 3 1 Leaf) 2 0 Leaf
```

One way to fix this…

Upper and lower bounds on the keys (if present)

Generate a key within the bounds (if possible)

```
instance (Ord k, Arbitrary k, Arbitrary v) =>
           Arbitrary (BST k v) where

  arbitrary = sized (tree Nothing Nothing)
    where
      tree lb ub 0 = return Leaf
      tree lb ub n =
        frequency [(1,return Leaf),
                   (2,do maybe_k <- between lb ub
                         case maybe_k of
                           Nothing -> return Leaf
                           Just k -> do
                             v <- arbitrary
                             l <- child lb (Just k)
                             r <- child (Just k) ub
                             return (Branch l k v r))]
        where child lb ub = tree lb ub ((n-1) `div` 2)
              …
```

…and between

Can't use **choose** *because that only works for numbers*

```
between lb ub =
  arbitrary
    `suchThatMaybe` (\x ->
      (lb==Nothing || lb < Just x) &&
      (ub==Nothing || Just x < ub))
```

suchThatMaybe tries repeatedly to generate a value satisfying the predicate, but may fail and generate **Nothing**

Maybe values are compared by comparing their components

```
*Examples> quickCheck prop_ArbitraryValid
+++ OK, passed 100 tests.

*Examples> sample (arbitrary :: Gen (BST Int Int))
Leaf
Branch Leaf (-1) (-1) Leaf
Leaf
Branch (Branch Leaf (-2) 3 Leaf) (-1) 4 (Branch
Leaf 0 (-1) Leaf)
Branch (Branch Leaf (-3) 4 Leaf) (-2) (-6) Leaf
Branch Leaf (-10) (-3) Leaf
Branch Leaf 10 (-3) (Branch Leaf 14 0 (Branch Leaf
19 12 Leaf))
Branch Leaf (-10) (-8) (Branch (Branch Leaf (-6) 6
Leaf) 14 5 Leaf)
Leaf
Branch (Branch (Branch (Branch Leaf (-10) (-10)
Leaf) (-1) (-18) Leaf) 2 (-10) Leaf) 12 (-18)
(Branch Leaf 15 8 (Branch Leaf 17 (-17) (Branch
Leaf 20 (-8) Leaf)))
Leaf
```

Are our tests effective?

- What size are the trees we generate?

```
size Leaf = 0
size (Branch l _ _ r) = size l + 1 + size r
```

```
prop_MeasureSize :: BST Int Int -> Property
prop_MeasureSize t = collect (size t) True
```

Data to collect

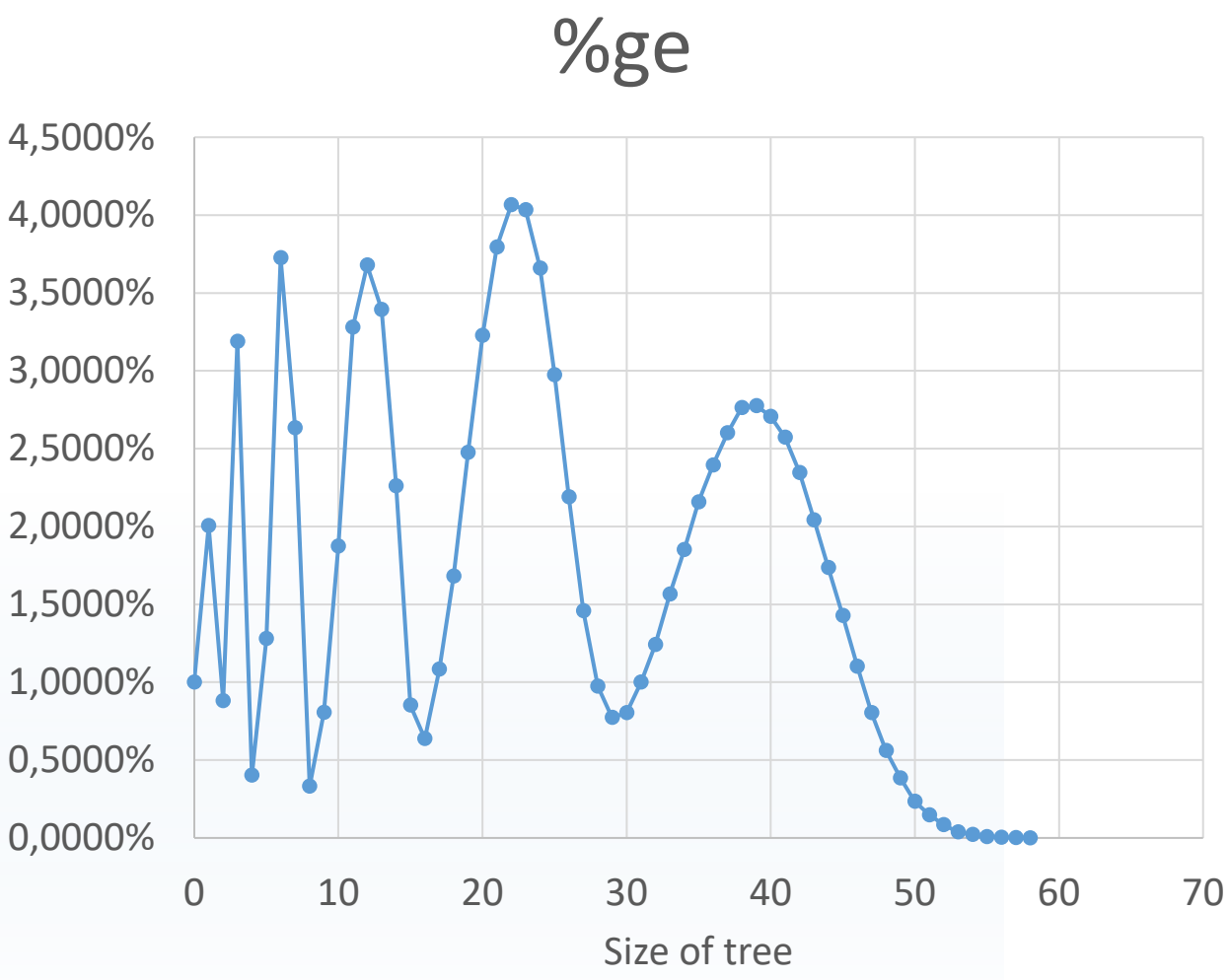
The rest of the property

```
*Examples> quickCheck prop_MeasureSize
+++ OK, passed 100 tests:
 9% 21
 6% 6
 5% 11
 5% 42
 4% 23
 4% 3
 4% 47
 3% 10
 3% 17
 3% 22
 3% 24
 3% 38
 2% 1
 2% 14
 2% 16
 2% 20
 2% 29
```

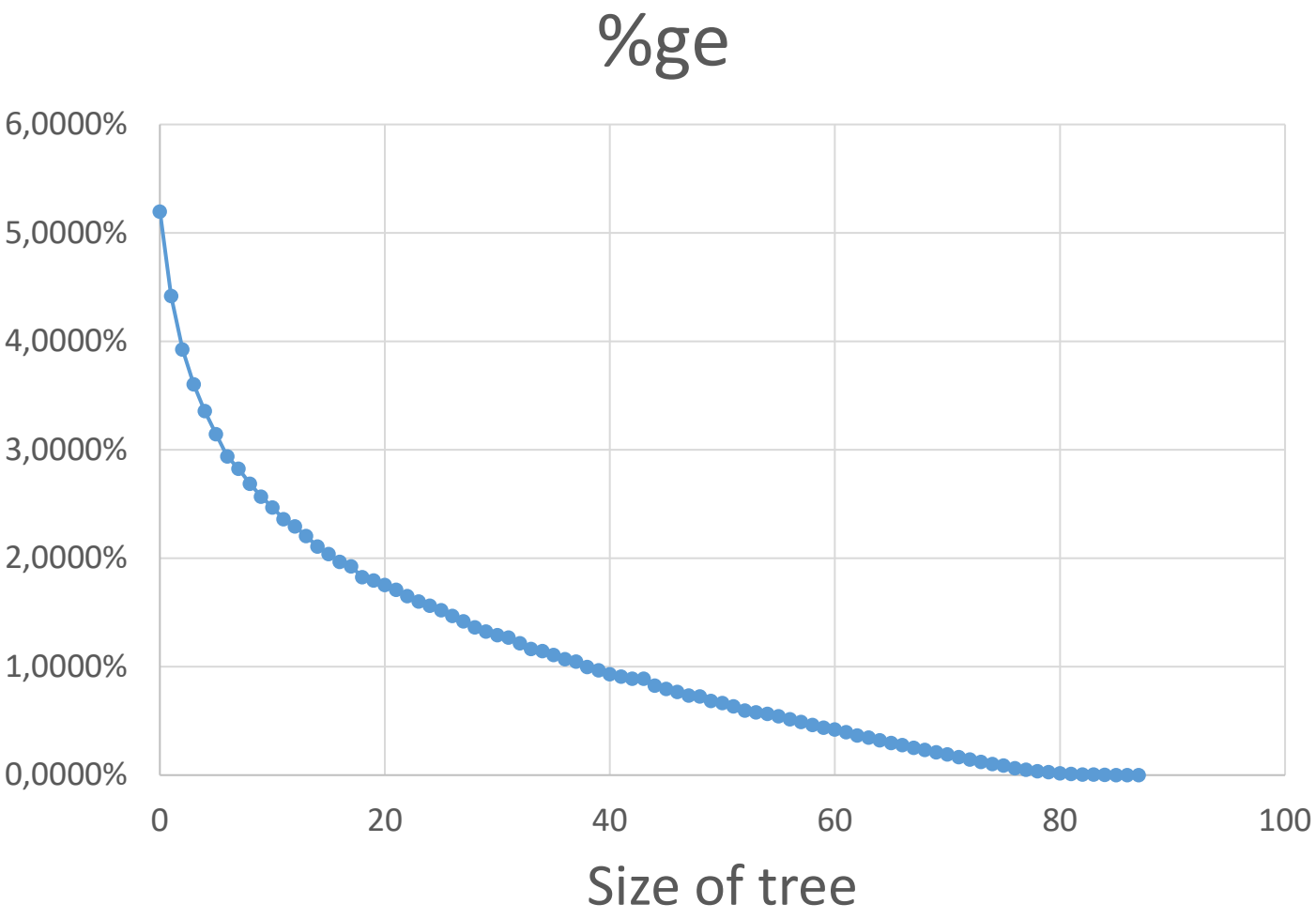
Way too small for reliable statistics

Decent looking variety of sizes, not too small, not too big

```
*Examples> quickCheck . withMaxSuccess 1000000 $ prop_MeasureSize
+++ OK, passed 1000000 tests:
 4.0675% 22
 4.0348% 23
 3.7954% 21
 3.7257% 6
 3.6792% 12
 3.6602% 24
 3.3938% 13
 3.2810% 11
 3.2282% 20
 3.1886% 3
 2.9750% 25
 2.7764% 39
 2.7644% 38
 2.7062% 40
 2.6339% 7
 2.6007% 37
 2.5724% 41
 2.4761% 19
 2.3949% 36
 2.3465% 42
```



Generator based on insertion



Are our tests effective?

```
prop_NilValid              = valid (nil :: Tree)
prop_InsertValid k v t = valid (insert k v t)
prop_DeleteValid k t   = valid (delete k t)
prop_UnionValid t t'   = valid (union t t')
```

- What *unit tests* would you write?
- How often is **`k`** present in **`t`**?
- How often *should* **`k`** be present in **`t`**?

About half the time?

Let's find out…

```
prop_MeasureMembership
  :: Int -> BST Int Int -> Property
prop_MeasureMembership k t = label l True
  where l | k `elem` keys t = "present"
          | otherwise       = "absent"
```

```
*Examples> quickCheck . withMaxSuccess 100000 $
              prop_MeasureMembership
+++ OK, passed 100000 tests:
79.258% absent
20.742% present
```

It depends on the type of keys…

```
prop_MeasureMembership
  :: String -> BST String Int -> Property
prop_MeasureMembership k t = label l True
  where l | k `elem` keys t = "present"
          | otherwise       = "absent"

*Examples> quickCheck . withMaxSuccess 100000 $
prop_MeasureMembership
+++ OK, passed 100000 tests:
98.558% absent
 1.442% present

*Examples> quickCheck . withMaxSuccess 100000 $
prop_MeasureMembership
+++ OK, passed 100000 tests; 16228 discarded:
99.987% absent
 0.013% present
```

not (null k) ==>

Draw keys from a smaller set

```
prop_MeasureMembership
  :: Key -> BST Key Int -> Property
prop_MeasureMembership k t = label l True
  where l | k `elem` keys t = "present"
          | otherwise       = "absent"
```

Is this code correct?

```
newtype Key = Key Int deriving (Eq, Ord, Show)

instance Arbitrary Key where
  arbitrary = Key <$> scale (`div` 3) arbitrary
  shrink (Key n) = map Key (shrink n)
```

52.114% absent

47.886% present

labelledExamples

```
*Examples> labelledExamples prop_MeasureMembership
*** Found example of absent
Key 0
Leaf

*** Found example of present
Key 0
Branch Leaf (Key 0) 0 Leaf

+++ OK, passed 100 tests:
50% absent
50% present
```

```
prop_MeasureMembership
  :: Key -> BST Key Int -> Property
prop_MeasureMembership k t = label l True
  where l | t == nil          = "nil"
          | k `atRoot` t      = "at root"
          | k `elem` keys t = "present"
          | otherwise         = "absent"
```

```
*Examples> labelledExamples . withMaxSuccess 100000 $
             prop_MeasureMembership
*** Found example of nil
Key 0
Leaf

*** Found example of at root
Key 0
Branch Leaf (Key 0) 0 Leaf

*** Found example of absent
Key 0
Branch Leaf (Key 1) 0 Leaf

*** Found example of present
Key 0
Branch Leaf (Key (-1)) 0 (Branch Leaf (Key 0) 0 Leaf)

+++ OK, passed 100000 tests:
47.128% absent
41.766% present
 5.907% at root
 5.199% nil
```

These are important test cases too

What if someone changes the generator?

- State *coverage requirements* as a property!

```
prop_CoverMembership
  :: Key -> BST Key Int -> Property
prop_CoverMembership k t =
  cover 5  (t == nil)                          "nil"     $
  cover 5  (k `atRoot` t)                      "at root" $
  cover 40 (k `elem` keys t &&
            not (k `atRoot` t))                "present" $
  cover 40 (k `notElem` keys t &&
            t /= nil)                          "absent"  $
  True
```

Required percentage of test cases

```
*Examples> quickCheck . checkCoverage $
             prop_CoverMembership
+++ OK, passed 51200 tests:
46.666% absent
42.246% present
 5.844% at root
 5.244% nil
```

SEQUENTIAL TESTS OF STATISTICAL HYPOTHESES

By A. Wald

Columbia University

Table of Contents

How often is it OK for a test to fail *when there is no bug?*


Agile Borat on Twitter: "M
Säker
https://twitter.com/AgileBorat/status/661640864880558 09
Home
About
Search Twitter
Agile Borat
@AgileBorat
Follow
My friend Azamat is very good developer, he is always have all unit test green. If unit test is fail, it is remove. Is best practice.
8:35 AM - 5 May 2011
256 Retweets
25 Likes
256
25
Agile Borat
@AgileBorat
Hello, I am Borat! Am Agile Coach, Scrum Master and Product Owner too also.
Joined April 2011
2
1

How often is it ok for a test to fail *when there is no bug?*

Never in the lifetime of the project!

10^{-6}?

10^{-9}?

Summary

- *Generators* and *shrinkers* must be written, for new types or custom distributions
- *Reuse* existing code as far as possible
- Include *tests* for generators and shrinkers
- *Measure* the distribution of tests
- *Label* test cases with unit test ideas
- *Debug* labelling with **`labelledExamples`**
- *Tune* generators to make test effective
- *State* and *test* coverage requirements with **`cover`**/**`checkCoverage`**.