Informatics 1

Functional Programming Lectures 11 and 12

Abstract Types

Don Sannella

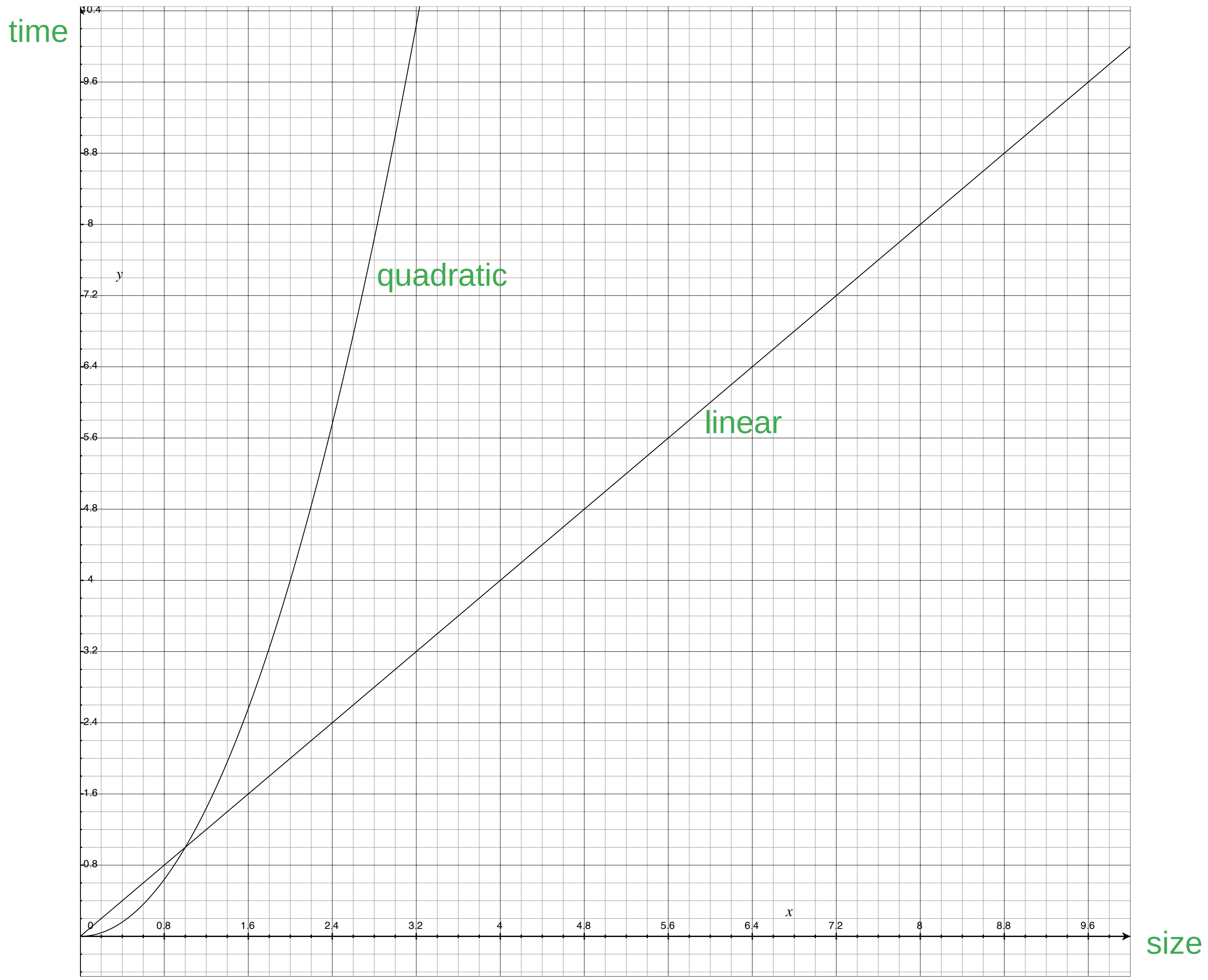
University of Edinburgh

Part I

Complexity

Premature optimisation is the root of all evil. Get it right, and make it clear.
But sometimes you do need things to run fast, or at least not really really slowly.
Especially when processing LOTS of data - millions or billions of items.
This lecture is about data abstraction, a way of separating getting things right from making them run fast.
First, let's look at the difference between fast programs and slow programs, concentrating on what happens for BIG inputs.
How long does it take to check if an item is in a list of n elements? Depends on how fast the computer is, and how big n is.
Best case: 1 step, because it's at the front of the list.
Worst case: n steps, because it's at the end of the list, or not in the list.
Average case: n/2 steps if it's there, n steps if not.

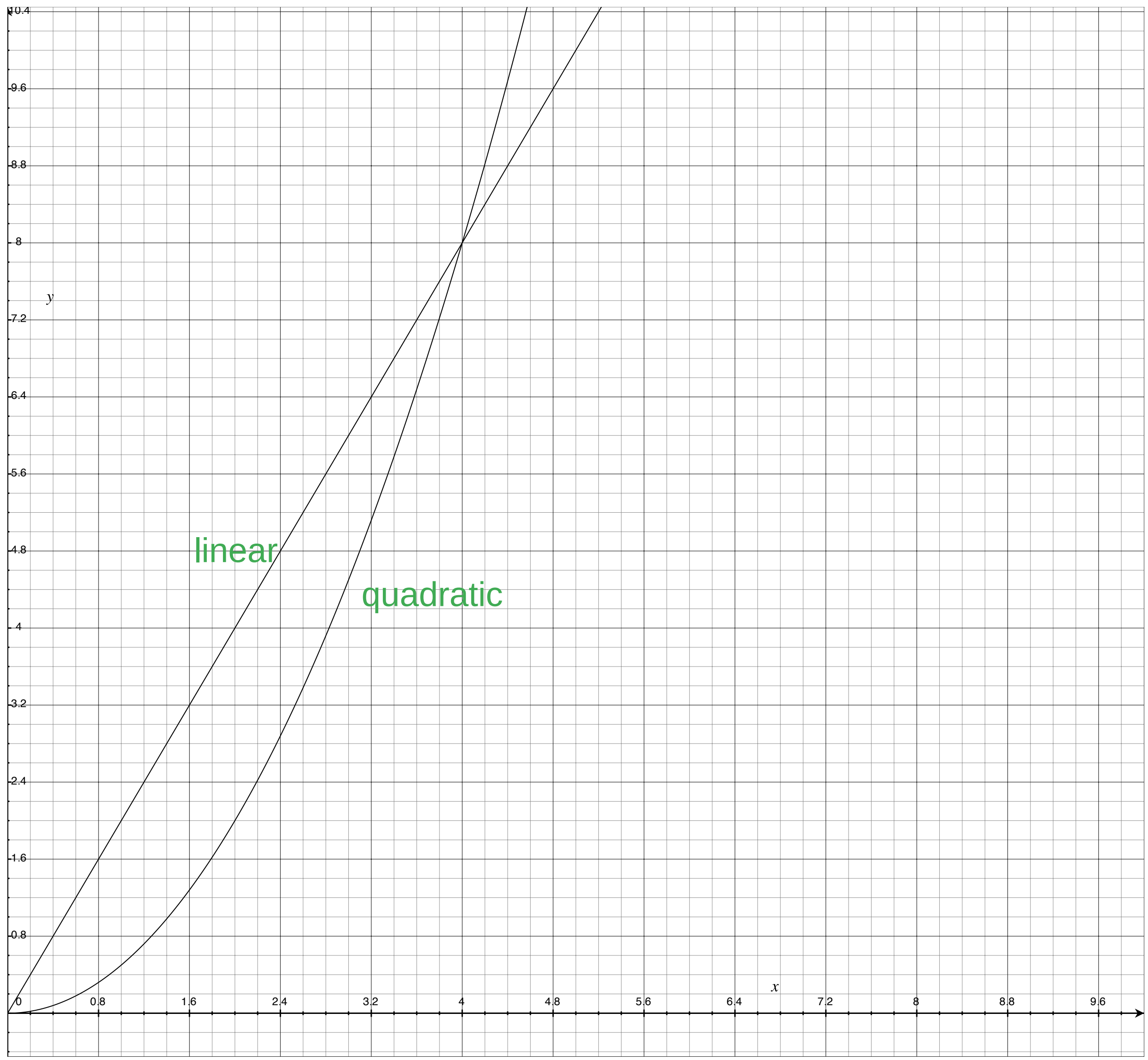
$t = n$ vs $t = n^2$



Here's what run time of n steps looks like ("linear") and how it compares with n^2 steps ("quadratic").
So n is faster than n^2 for n>1.

$t = 2n$ vs $t = 0.5n^2$

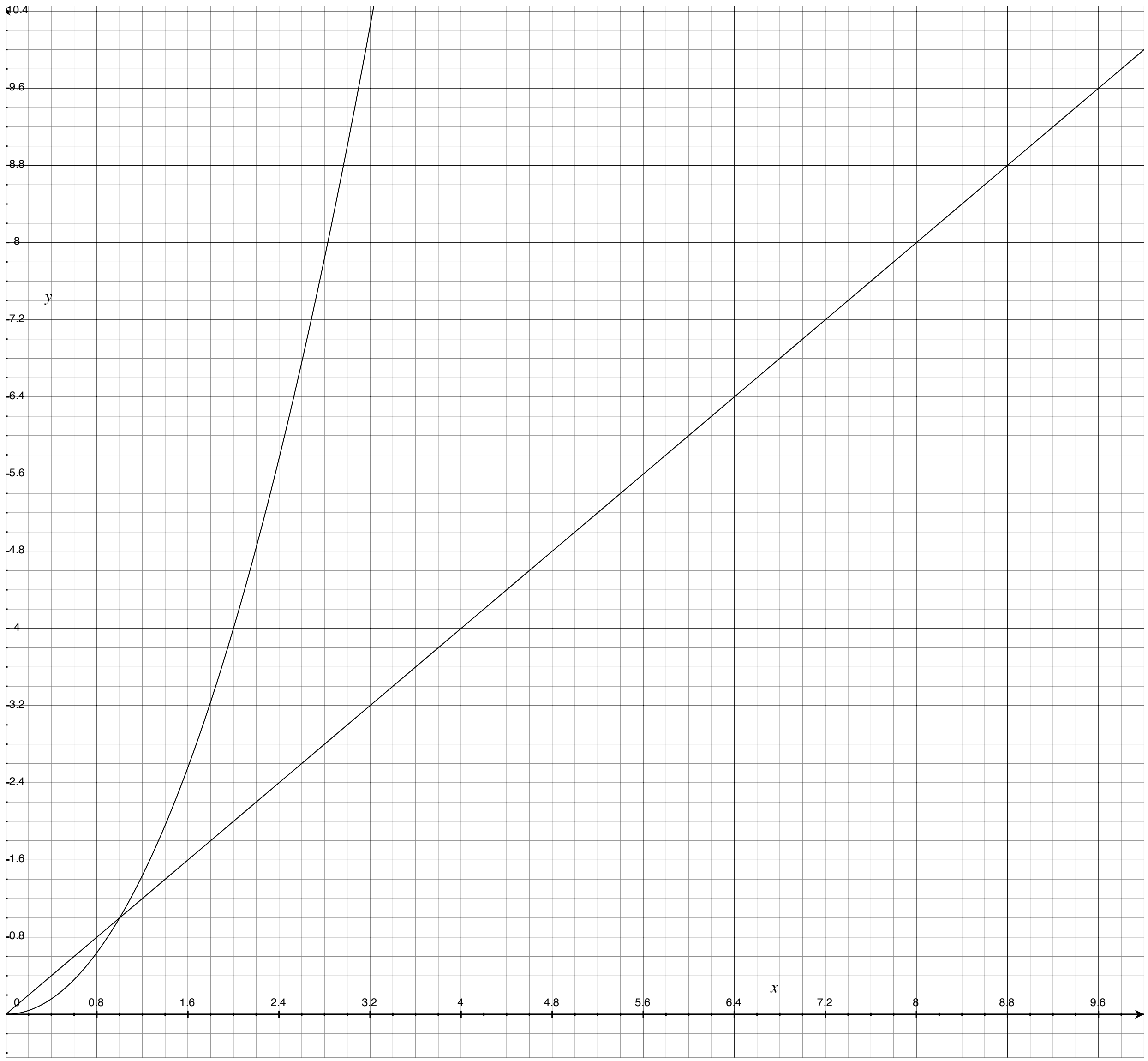
But what about a really fast quadratic algorithm (say 0.5n^2)
versus a really slow linear algorithm (say 2n)?



n is better for n>4: 2*4 = 0.5*4^2 = 8.
cn is always better than dn^2, for any c,d, for big enough n. For small n, who cares?
cn = dn^2 for n >= c/d.
That's why we care about linear versus quadratic and not about c and d.

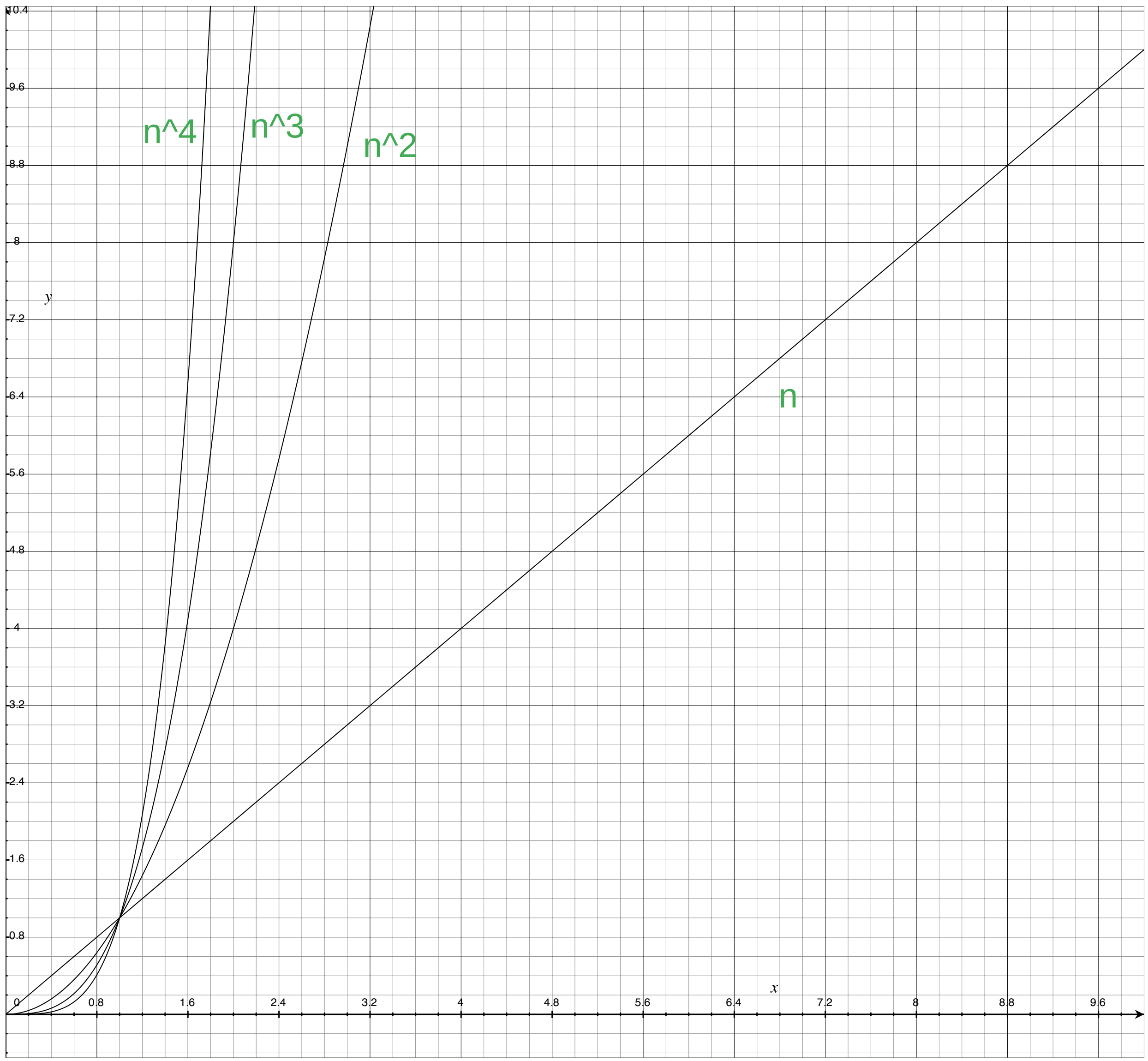
$O(n)$ vs $O(n^2)$

O-notation captures the idea that multiplicative and additive factors don't matter.
f is O(n) means f(x) <= cx for x>m for some c,m
f is O(n^n) means f(x) <= cx^2 for x>m for some c,m
etc.



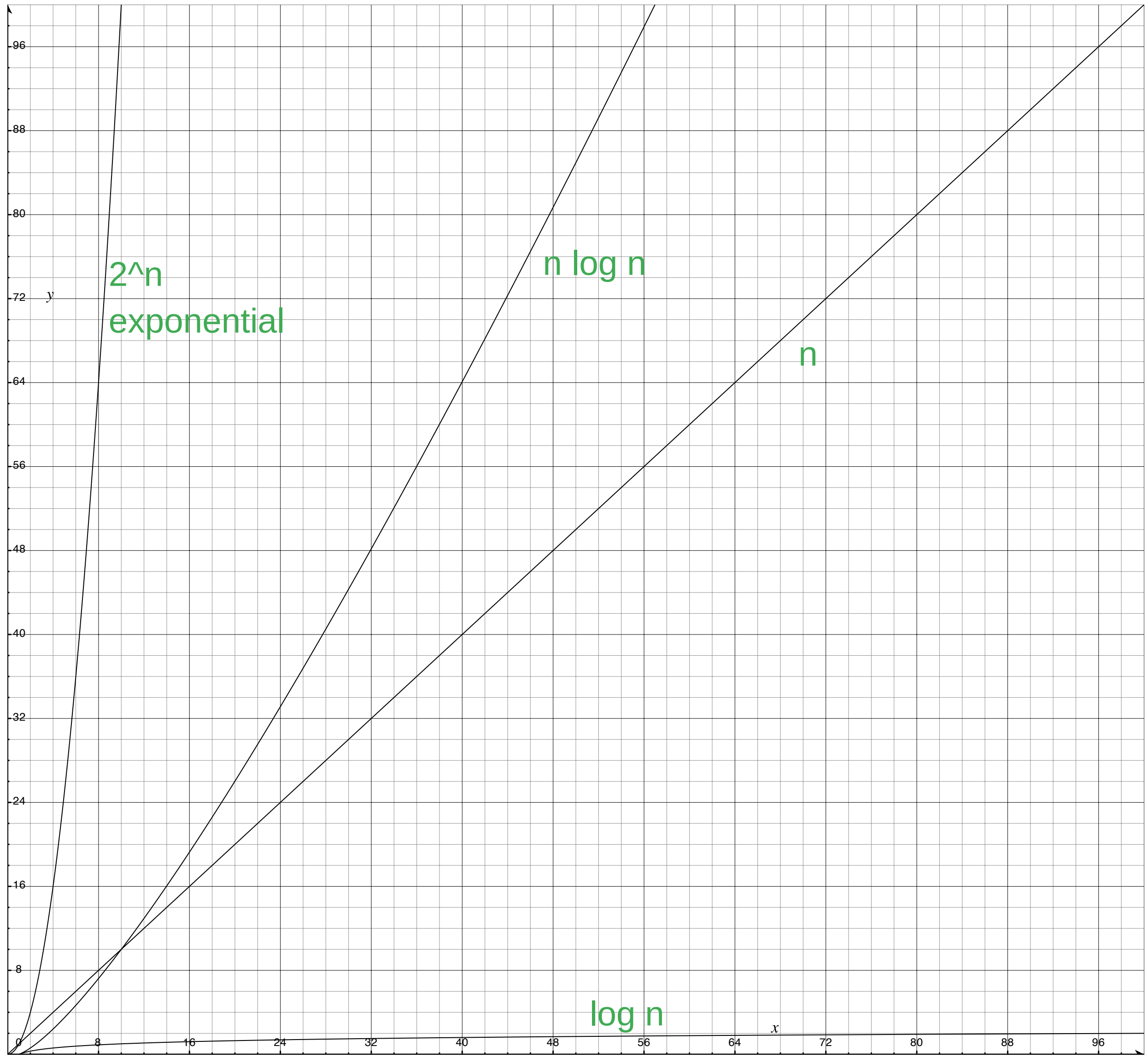
You can show that O(n^2 + n) = O(n^2), O(n^3 + n^2 + n) = O(n^3), O(n+b) = O(n) etc.
You only care about the degree of the polynomial - that's why we say linear, quadratic etc.

$$O(n), O(n^2), O(n^3), O(n^4)$$



O(n^4) is usually too slow. O(n^3) is maybe tolerable. O(n^2) is okay. O(n) is great.
For really big data sets, you need O(n) or better.

$O(\log n), O(n), O(n \log n), O(2^n)$



Logarithms arise naturally in "divide and conquer" algorithms.
Exponential (2^n) is really bad - intractable. E.g. building truth tables - add one variable, table doubles in size.
Logarithmic (log n) is really great - 1000->1000000 takes twice as long.
Many sorting algorithms are n log n.

Part II

Sets as lists without abstraction

We're now going to look at several different ways of implementing sets, and compare them using O-notation.
The easiest way is using a list, so we'll start with that.
"Without abstraction" will be explained later.

ListUnabs.hs (1)

A module gives a name to a program unit, saying what it exports (list of names) and what it needs to do its work (imports).

```
module ListUnabs
  (Set,empty,insert,set,element,equal,check) where
import Test.QuickCheck
```

We're going to look at a series of modules that all export the same names, but have different implementations of data. Here, sets are represented as lists.

```
type Set a = [a]
```

```
empty :: Set a
empty =  []
```

Empty set is empty list.

```
insert :: a -> Set a -> Set a
insert x xs  =  x:xs
```

Inserting an element is just : (cons) - adding new element to the beginning of the list.
Could instead add it in the middle or end - doesn't matter. O(1)

```
set :: [a] -> Set a
set xs  =  xs
```

Convert a list into a set: don't need to do anything, it is a set already. O(1)

ListUnabs.hs (2)

```
element :: Eq a => a -> Set a -> Bool
x `element` xs  =  x `elem` xs

equal :: Eq a => Set a -> Set a -> Bool
xs `equal` ys  =  xs `subset` ys && ys `subset` xs
  where
  xs `subset` ys  =  and [ x `elem` ys | x <- xs ]
```

To test if an item is in a set, just use built-in elem function on lists. Looks through list from the beginning, stopping when it finds item or runs out of elements. So O(n).

To check equality, we can't just compare the underlying lists for equality:

insert 1 (insert 2 empty) = [1,2]

insert 2 (insert 1 empty) = [2,1]

insert 1 (insert 2 (insert 1 empty)) = [1,2,1]

but we want to regard these as the same set - order of insertion isn't supposed to matter, for sets. So we define subset (xs `subset` ys if each element in xs is also in ys) and then xs and ys have the same elements if xs `subset` ys and vice versa.

Equality is O(n^2): for every of n elements in xs, need to check if it is in ys - which is O(n) - and vice versa. (Actually O(nm), if xs has length n and ys has length m.)

ListUnabs.hs (3)

```
prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_element

-- Prelude ListUnabs> check
-- +++ OK, passed 100 tests.
```

ListUnabsTest.hs

```haskell
module ListUnabsTest where
import ListUnabs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]
```

How would I use this module? I might make sets out of [1,2,...,n] and [n,n-1,...,1] and check to see if they are equal.

```haskell
breakAbstraction :: Set a -> a
breakAbstraction =  head
```

Because set = list, all list function can be applied to sets!

```haskell
-- not a function!
-- head (set [1,2,3]) == 1 /= 3 == head (set [3,2,1])
```

But head isn't a function on sets: set [1,2,3] `equal` set [3,2,1] but head(set [1,2,3]) /= head(set [3,2,1])
It isn't enough to write documentation saying "please don't apply head to sets". We need a better solution.

This is called "breaking the abstraction".

Part III

Sets as *ordered* lists without abstraction

A different way to represent a set is as an ordered list without duplicates. Then

```
insert 1 (insert 2 empty) = [1,2]
insert 2 (insert 1 empty) = [1,2]
insert 1 (insert 2 (insert 1 empty)) = [1,2]
```

So equality checking should be easier.

OrderedListUnabs.hs (1)

```haskell
module OrderedListUnabs
  (Set,empty,insert,set,element,equal,check) where

import Data.List(nub,sort)
import Test.QuickCheck

type Set a = [a]

invariant :: Ord a => Set a -> Bool
invariant xs  =
  and [ x < y | (x,y) <- zip xs (tail xs) ]
```

Module heading as before, but I need some extra imports.

Same type definition as before.

But now I have an invariant: I insist that adjacent elements are always in ascending order.
And since < rather than <=, there are no duplicates.

OrderedListUnabs.hs (2)

```
empty :: Set a
empty =  []

insert :: Ord a => a -> Set a -> Set a
insert x []                 =  [x]
insert x (y:ys) | x < y     =  x : y : ys
                | x == y    =  y : ys
                | x > y     =  y : insert x ys
```

Adding an element to a set is harder then before - we need to put it in the right place. O(n)

```
set :: Ord a => [a] -> Set a
set xs  =  nub (sort xs)
```

Making a list into a set.
One way is to sort it and then remove duplicates, which is O(n log n) provided Haskell uses a good sorting algorithm.
Another way is to insert each item in the list into a set, starting with the empty set:
set xs = foldr insert empty xs
but that is slower, O(n^2).

OrderedListUnabs.hs (3)

```
element :: Ord a => a -> Set a -> Bool
x `element` []                   =   False
x `element` (y:ys) | x < y       =   False
                   | x == y      =   True
                   | x > y       =   x `element` ys
```

To check membership: because the list is in order, we can stop when we get to a bigger element.
Still O(n), even though faster than for unordered lists.

```
equal :: Eq a => Set a -> Set a -> Bool
xs `equal` ys   =   xs == ys
```

Equality: just use list equality.
O(n), much better than unordered lists.

OrderedListUnabs.hs (4)

```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

Prelude OrderedListUnabs> check
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

OrderedListUnabsTest.hs

```haskell
module OrderedListUnabsTest where
import OrderedListUnabs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]

breakAbstraction :: Set a -> a
breakAbstraction =  head
-- now it's a function
-- head (set [1,2,3]) == 1 == head (set [3,2,1])
```

Head is a function now because it always returns the smallest element.

```haskell
badtest :: Int -> Bool
badtest n =
  s `equal` t
  where
  s = [1,2..n]      -- no call to set!
  t = [n,n-1..1]    -- no call to set!
```

But now I can break the abstraction by using equal on lists that don't satisfy the invariant - nothing to stop me.

Could also check membership in t - will get the wrong answer

Membership and equality rely on the invariant.

Part IV

Sets as ordered trees without abstraction

We can do better!
It's common to represent sets as trees.
If done properly, we can make membership O(log n) rather than O(n).

TreeUnabs.hs (1)

```
module TreeUnabs
  (Set(Nil,Node),empty,insert,set,element,equal,check) where
import Test.QuickCheck

data  Set a  =  Nil | Node (Set a) a (Set a)
```

A set is a tree: either empty (Nil) or a node with a left subtree, a data value, and a right subtree.

```
list :: Set a -> [a]
list Nil              =  []
list (Node l x r)   =  list l ++ [x] ++ list r
```

We can convert a tree to a list by appending all of the node labels in order. "Inorder traversal".

```
invariant :: Ord a => Set a -> Bool
invariant Nil    =  True
invariant (Node l x r)   =
  invariant l && invariant r &&
  and [ y < x | y <- list l ] &&
  and [ y > x | y <- list r ]
```

The invariant says that, at every node, all the values in the left subtree are less than the node label, and all the values in the right subtree are greater than the node label.

TreeUnabs.hs (2)

```
empty :: Set a
empty  =  Nil

insert :: Ord a => a -> Set a -> Set a
insert x Nil  =  Node Nil x Nil
insert x (Node l y r)
  | x == y     =  Node l y r
  | x < y      =  Node (insert x l) y r
  | x > y      =  Node l y (insert x r)
```

Inserting an element needs to put it in the right place.
We use the node labels to find the right place.

```
set :: Ord a => [a] -> Set a
set  =  foldr insert empty
```

We can convert a list to a set by inserting each of its elements, starting with the empty tree.

TreeUnabs.hs (3)

```
element :: Ord a => a -> Set a -> Bool
x `element` Nil   =  False
x `element` (Node l y r)
  | x == y      =  True
  | x < y       =  x `element` l
  | x > y       =  x `element` r
```

To check if x is an element, use the node labels to find the right place to look.
At each node we can ignore a subtree, because of the invariant - we know that x can't be there!
So at each node we can ignore about half of the remaining elements, if the tree is balanced. O(log n).

```
equal :: Ord a => Set a -> Set a -> Bool
s `equal` t  =  list s == list t
```

Equality is O(n): convert to a list in O(n), then check for equality in O(n).

TreeUnabs.hs (4)

```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

-- Prelude TreeUnabs> check
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
```

TreeUnabsTest.hs

```
module TreeUnabsTest where
import TreeUnabs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]

badtest :: Bool
badtest =
  s `equal` t
  where
  s = set [1,2,3]
  t = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
  -- breaks the invariant!
```

Works very well for balanced trees. But trees may not be balanced.
set [1,2,...,n] will be very unbalanced, for instance.
x `element` set [1,2,...,n] is O(n), not O(log n).

Again, we can break the abstraction by building a tree that doesn't respect the invariant - gets equal to give the wrong answer.

Part V

Sets as *balanced* trees without abstraction

If we are clever, we can make sure that trees are always balanced: AVL trees
Invented 1962 by Adelson-Velskii and Landis.
First example you're seeing of a clever data structure - there are LOTS of others, see Inf2B.

We're going to ensure that at each node, the depths of the left and right subtrees differ by at most 1.
It's impossible to do better than that, unless the tree has exactly 2^d - 1 elements.

BalancedTreeUnabs.hs (1)

```
module BalancedTreeUnabs
  (Set(Nil,Node),empty,insert,set,element,equal,check) where
import Test.QuickCheck

type  Depth  =  Int
data  Set a  =  Nil | Node (Set a) a (Set a) Depth
```

Same data representation, but I keep track of the depth at each node.

```
node :: Set a -> a -> Set a -> Set a
node l x r  =  Node l x r (1 + (depth l `max` depth r))
```

When I build a node, I need to calculate its depth.

```
depth :: Set a -> Int
depth Nil  =  0
depth (Node _ _ _ d) = d
```

BalancedTreeUnabs.hs (2)

```
list :: Set a -> [a]
list Nil                  =  []
list (Node l x r _)  =  list l ++ [x] ++ list r
```

I can turn a tree into a list as before.

```
invariant :: Ord a => Set a -> Bool
invariant Nil   =  True
invariant (Node l x r d)  =
  invariant l && invariant r &&
  and [ y < x | y <- list l ] &&
  and [ y > x | y <- list r ] &&
  abs (depth l - depth r) <= 1 &&
  d == 1 + (depth l `max` depth r)
```

The invariant is the same as before, plus the balance property.
Also, the depth component of each node should be accurate.

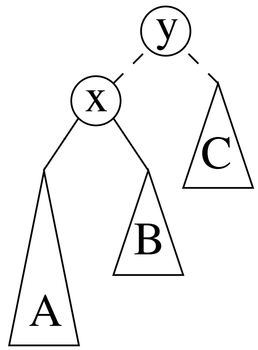
BalancedTreeUnabs.hs (3)

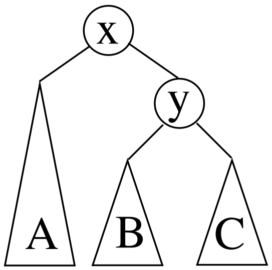
```
empty :: Set a
empty  =  Nil

insert :: Ord a => a -> Set a -> Set a
insert x Nil  =  node empty x empty
insert x (Node l y r _)
  | x == y      =  node l y r
  | x < y       =  rebalance (node (insert x l) y r)
  | x > y       =  rebalance (node l y (insert x r))

set :: Ord a => [a] -> Set a
set  =  foldr insert empty
```

Inserting is just as before, except that after inserting I need to rebalance. Rebalancing is the tricky part.

Rebalancing

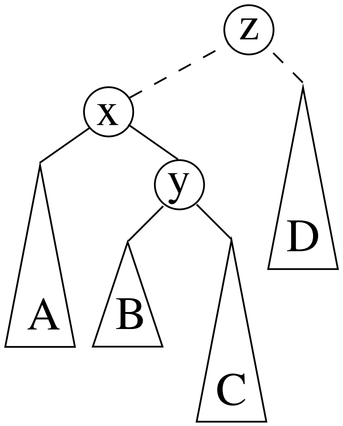
Rebalancing is best understood by using these pictures.

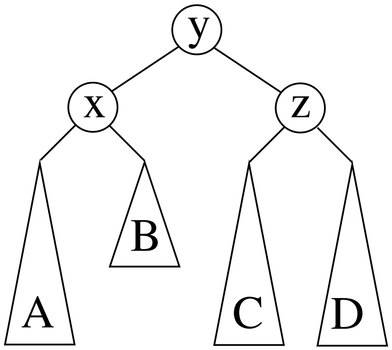





```
Node (Node a x b) y c    -->    Node a x (Node b y c)
```

A is more than 1 longer than C: rearrange, retaining the order AxByC






```
Node (Node a x (Node b y c) z d)
            --> Node (Node a x b) y (Node c z d)
```

C is more than 1 longer than D: rearrange, retaining the order AxByCzD.
These, plus symmetric variants, are the only two cases.

BalancedTreeUnabs.hs (4)

```
rebalance :: Set a -> Set a
rebalance (Node (Node a x b _) y c _)
  | depth a >= depth b && depth a > depth c
  = node a x (node b y c)
rebalance (Node a x (Node b y c _) _)
  | depth c >= depth b && depth c > depth a
  = node (node a x b) y c
rebalance (Node (Node a x (Node b y c _) _) z d _)
  | depth (node b y c) > depth d
  = node (node a x b) y (node c z d)
rebalance (Node a x (Node (Node b y c _) z d _) _)
  | depth (node b y c) > depth a
  = node (node a x b) y (node c z d)
rebalance a  =  a
```

Here's the code - easy to understand if you look at the pictures.
There are 5 cases - the 2 we've seen, plus symmetric variants, plus the case where no rebalancing is required.

BalancedTreeUnabs.hs (5)

```
element :: Ord a => a -> Set a -> Bool
x `element` Nil  =  False
x `element` (Node l y r _)
  | x == y      =  True
  | x < y       =  x `element` l
  | x > y       =  x `element` r

equal :: Ord a => Set a -> Set a -> Bool
s `equal` t  =  list s == list t
```

Element test as before.
Now O(log n), because the tree is balanced.

Equality as before, O(n).

BalancedTreeUnabs.hs (6)

```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

-- Prelude BalancedTreeUnabs> check
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
```

BalancedTreeUnabsTest.hs

```
module BalancedTreeUnabsTest where
import BalancedTreeUnabs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]

badtest :: Bool
badtest =
  s `equal` t
  where
  s = set [1,2,3]
  t = (Node Nil 1 (Node Nil 2 (Node Nil 3 Nil 1) 2) 3)
  -- breaks the invariant!
```

We can still break the invariant.

Part VI

Complexity, revisited

Summary

	`insert`	`set`	`element`	`equal`
`List`	$O(1)$	$O(1)$	$O(n)$	$O(n^2)$
`OrderedList`	$O(n)$	$O(n \log n)$	$O(n)$	$O(n)$
`Tree`	$O(\log n)^*$ $O(n)^\dagger$	$O(n \log n)^*$ $O(n^2)^\dagger$	$O(\log n)^*$ $O(n)^\dagger$	$O(n)$
`BalancedTree`	$O(\log n)$	$O(n \log n)$	$O(\log n)$	$O(n)$

$*$ average case / $\dagger$ worst case

Here is a summary: considering insertion, creating of a set from a list, element testing, and equality.

Balanced tree is the best.
Actually, you need to consider the mix of operations
List might be best if you know that you will be doing lots of insertions and almost no element testing or equality.

Part VII

Data Abstraction

How do we keep people from breaking our abstraction?
It's easy - we use data constructors, and are very careful about who gets to use them.

ListAbs.hs (1)

```
module ListAbs
  (Set,empty,insert,set,element,equal,check) where
import Test.QuickCheck

data  Set a  =  MkSet [a]
```

We need to include a constructor: MkSet

```
empty :: Set a
empty =  MkSet []
```

empty uses the constructor

```
insert :: a -> Set a -> Set a
insert x (MkSet xs)  =  MkSet (x:xs)
```

insert needs to extract the list, add an element, then turn the result back into a set.

```
set :: [a] -> Set a
set xs  =  MkSet xs
```

set is just MkSet

ListAbs.hs (2)

```
element :: Eq a => a -> Set a -> Bool
x `element` (MkSet xs)  =  x `elem` xs
```

Just as before, once the list has been extracted from the set

```
equal :: Eq a => Set a -> Set a -> Bool
MkSet xs `equal` MkSet ys  =
  xs `subset` ys && ys `subset` xs
  where
  xs `subset` ys  =  and [ x `elem` ys | x <- xs ]
```

Ditto for equal

It seems a little tedious and pointless, all this unpacking and re-packing using MkSet.
But wait a minute.

ListAbs.hs (3)

```
prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_element

-- Prelude ListAbs> check
-- +++ OK, passed 100 tests.
```

ListAbsTest.hs

```
module ListAbsTest where
import ListAbs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]

-- Following no longer type checks!
-- breakAbstraction :: Set a -> a
-- breakAbstraction =  head
```

Now we can't break the abstraction: head works on lists, not on sets!
But wait a minute: somebody could just extract the list from a set, using pattern matching with MkSet.
We prevent that by not exporting MkSet - it's only available inside the module.
It's mine, you can't have it.

Hiding—the secret of abstraction

```
module ListAbs(Set,empty,insert,set,element,equal)

> ghci ListAbs.hs
Ok, modules loaded: SetList, MainList.
*ListAbs> let s0 = set [2,7,1,8,2,8]
*ListAbs> let MkSet xs = s0 in xs
Not in scope: data constructor ‘MkSet’
```

vs.

```
module ListUnhidden(Set(MkSet),empty,insert,element,equal)

> ghci ListUnhidden.hs
*ListUnhidden> let s0 = set [2,7,1,8,2,8]
*ListUnhidden> let MkSet xs = s0 in xs
[2,7,1,8,2,8]
*ListUnhidden> head xs
```

In the module heading, I exported Set but not MkSet. If I do export MkSet, then it can be used to break the abstraction.

By not exporting MkSet, you can guarantee that nobody can break your abstraction.
The only way that people can get access to the representation is via the functions provided by the module.

Hiding—the secret of abstraction

```
module TreeAbs(Set,empty,insert,set,element,equal)

> ghci TreeAbs.hs
Ok, modules loaded: SetList, MainList.
*TreeAbs> let s0 = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
Not in scope: data constructor 'Node', 'Nil'
```

VS.

```
module TreeUnabs(Set(Node,Nil),empty,insert,element,equal)

> ghci TreeUnabs.hs
*SetList> let s0 = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
*SetList> invariant s0
False
```

For trees, it's exactly the same: I don't export Nil and Node, so I can't build a tree that violates the invariant.
This makes the constructors accessible only inside the module, making the abstraction unbreakable.
That's the secret to protecting the abstraction and having control over the representation.

Preserving the invariant

```haskell
module TreeAbsInvariantTest where
import TreeAbs

prop_invariant_empty = invariant empty

prop_invariant_insert x s =
  invariant s ==> invariant (insert x s)

prop_invariant_set xs = invariant (set xs)

check =
  quickCheck prop_invariant_empty >>
  quickCheck prop_invariant_insert >>
  quickCheck prop_invariant_set

-- Prelude TreeAbsInvariantTest> check
-- +++ OK, passed 1 tests.
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
```

You can ensure that the invariant holds by checking that it holds for all functions in the module that produce values of type Set.

A function like insert, which takes a Set as argument, needs to PRESERVE the invariant.

In this case, set will satisfy the invariant since it just combines empty and insert.

It's mine!

The constructors are mine. You can't have them. That's the secret of data abstraction.

Then you can separate getting things right from making them fast. How? Create data abstractions in modules, and protect the abstraction. If you pick an inefficient representation, find a better one that provides the same "interface" (functions/types it exports). You can then replace the bad representation by the efficient one, without changing anything else!
Protection of the abstraction means that the rest of the program CAN'T depend on details that might change.