From the Prelude:

data [a] = [] | (:) a [a]
From the Prelude:

```
data [a] = [] | (:) a [a]
    deriving Eq
```

The result of deriving `Eq`:

```
instance Eq a => Eq [a] where
    []   == []  = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _     == _    = False
```

Defined explicitly:

```
instance Show a => Show [a] where
    show xs = "[" ++ concat cs ++ "]"
    where cs = Data.List.intersperse "," (map show xs)
```

From the Prelude:

```
data [a] = [] | (:) a [a]
    deriving Eq
```

The result of deriving `Eq`:

```
instance Eq a => Eq [a] where
    []   == []  = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _     == _    = False
```

Defined explicitly:

```
instance Show a => Show [a] where
    show xs = "[" ++ concat cs ++ "]"
    where cs = Data.List.intersperse "," (map show xs)
```

```
data Tree a = Empty | Node a (Tree a) (Tree a)
    deriving (Eq, Show)
```
data Tree a = Empty | Node a (Tree a) (Tree a)  
   deriving (Eq, Show)

Some trees:
Empty
Node 1 Empty Empty
Node 1 (Node 2 Empty Empty) Empty
Node 1 Empty (Node 2 Empty Empty)
Node 1 (Node 2 Empty Empty) (Node 3 Empty Empty)
find :: a -> Tree a -> Bool

find _ Empty = False
find x (Node a l r)
  | x < a = find x l
  | a < x = find x r
  | otherwise =
find :: Ord a => a -> Tree a -> Bool
find_ Empty = False
find x (Node a l r)
  | x < a = find x l
  | a < x = find x r
  | otherwise = True

-- assumption: < is a linear ordering
find :: Ord a => a -> Tree a -> Bool
find_ Empty = False
find x (Node a l r)
  | x < a = find x l
  | a < x = find x r
  | otherwise = True

insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
  | x < a = Node a (insert x l) r
  | a < x = Node a l (insert x r)
  | otherwise =
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
  | x < a = Node a (insert x l) r
  | a < x = Node a l (insert x r)
  | otherwise = Node a l r

Example
insert 6 (Node 5 Empty (Node 7 Empty Empty))

insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
  | x < a = Node a (insert x l) r
  | a < x = Node a l (insert x r)
  | otherwise = Node a l r

Example
insert 6 (Node 5 Empty (Node 7 Empty Empty))
= Node 5 Empty (insert 6 (Node 7 Empty Empty))

QuickCheck for Tree

import Control.Monad
import Test.QuickCheck

-- for QuickCheck: test data generator for Trees
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized tree
    where
tree 0 = return Empty
tree n | n > 0 =
    oneof [return Empty,
           liftM3 Node arbitrary (tree (n `div` 2))
                      (tree (n `div` 2))]
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
    | x < a = Node a (insert x l) r
    | a < x = Node a l (insert x r)
    | otherwise = Node a l r

Example
insert 6 (Node 5 Empty (Node 7 Empty Empty))
= Node 5 Empty (insert 6 (Node 7 Empty Empty))
= Node 5 Empty (Node 7 (insert 6 Empty) Empty)
= Node 5 Empty (Node 7 (Node 6 Empty Empty) Empty)

import Control.Monad
import Test.QuickCheck
-- for QuickCheck: test data generator for Trees
instance Arbitrary a => Arbitrary (Tree a) where
    arbitrary = sized tree
    where
        tree 0 = return Empty
        tree n | n > 0 =
            oneof [return Empty,
                liftM3 Node arbitrary (tree (n `div` 2))
                (tree (n `div` 2))]

prop_find_insert x y t =
    find x (insert y t) == ???

prop_find_insert x y t =
    find x (insert y t) == (x == y || find x t)
prop_find_insert :: Int -> Int -> Tree Int -> Bool
prop_find_insert x y t = find x (insert y t) == (x == y || find x t)

(prop_not optimal for QuickCheck)

Edit distance (see Thompson)

Problem: how to get from one word to another, with a minimal number of "edits".
Example: from "fish" to "chips"
Applications: DNA Analysis,
Edit distance (see Thompson)

Problem: how to get from one word to another, with a minimal number of “edits”.
Example: from "fish" to "chips"
Applications: DNA Analysis, Unix diff command

data Edit = Change Char 
  | Copy
  | Delete
  | Insert Char
  deriving (Eq, Show)

transform :: String -> String -> [Edit]

data Edit = Change Char 
  | Copy
  | Delete
  | Insert Char
  deriving (Eq, Show)

transform [] ys =
```haskell
data Edit = Change Char  
| Copy  
| Delete  
| Insert Char  
    deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys = map Insert ys
transform xs [] = replicate (length xs) Delete
transform (x:xs) (y:ys)
    | x == y     = Copy : transform xs ys
```

```haskell
data Edit = Change Char  
| Copy  
| Delete  
| Insert Char  
    deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys = map Insert ys
transform xs [] = replicate (length xs) Delete
transform (x:xs) (y:ys)
    | x == y     = Copy : transform xs ys
    | otherwise  = best
```
Example: What is the edit distance from "trittin" to "tarantino"?
data Edit = Change Char
    | Copy
    | Delete
    | Insert Char
deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys = map Insert ys
transform xs [] = replicate (length xs) Delete
transform (x:xs) (y:ys)
    | x == y      = Copy : transform xs ys
    | otherwise   = best [Change y : transform xs ys,
                         Delete : transform xs (y:ys),
                         Insert y : transform (x:xs) ys]

Example: What is the edit distance from "trittin" to "tarantino"?
transform "trittin" "tarantino" = ?

Complexity of transform: time $O(3^{m+n})$

Example: What is the edit distance from "trittin" to "tarantino"?
transform "trittin" "tarantino" = ?

Complexity of transform: time $O(3^{m+n})$

The edit distance problem can be solved in time $O(mn)$ with dynamic programming
8.2 The general case

data \( T \ a_1 \ldots a_p = \]
\( C_1 \ t_{11} \ldots t_{1k_1} \mid \]
\( \vdots \]
\( C_n \ t_{n1} \ldots t_{nk_n} \]
defines the **constructors**
\( C_1 :: t_{11} \rightarrow \ldots t_{1k_1} \rightarrow T \ a_1 \ldots a_p \]
\( \vdots \]
\( C_n :: t_{n1} \rightarrow \ldots t_{nk_n} \rightarrow T \ a_1 \ldots a_p \]

---

```
data Edit = Change Char
    | Copy
    | Delete
    | Insert Char
    deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys = map Insert ys
transform xs [] = replicate (length xs) Delete
transform (x:xs) (y:ys)
    | x == y = Copy : transform xs ys
    | otherwise = best [Change y : transform xs ys,
                        Delete : transform xs (y:ys),
                        Insert y : transform (x:xs) ys]
```
Patterns are expressions that consist only of constructors and variables (which must not occur twice):
A pattern can be
- a variable (incl. \_)
- a literal like 1, 'a', "xyz", ...

3.3 Case study: boolean formulas

Patterns are expressions that consist only of constructors and variables (which must not occur twice):
A pattern can be
- a variable (incl. \_)
- a literal like 1, 'a', "xyz", ...
- a tuple \( \langle p_1, \ldots, p_n \rangle \) where each \( p_i \) is a pattern
- a constructor pattern \( C \ p_1 \ldots \ p_n \) where
  \( C \) is a data constructor (incl. True, False, [], and ())
  and each \( p_i \) is a pattern
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
  | x < a = Node a (insert x l) r
  | a < x = Node a l (insert x r)
  | otherwise = Node a l r

Example
insert 6 (Node 5 Empty (Node 7 Empty Empty))