LIST INDUCTION

David Kauchak
CS52 – Spring 2015

Admin

Assignment 6

Assignment 4 grading
- 100 vs. 300 stack size
- name missing

Today

http://lavonhardison.com/tag/repetition/

List induction

1. State what you're trying to prove!
2. State and prove the base case (often empty list)
3. Assume it's true for sublists – inductive hypothesis
4. Show that it holds for the full list

List fact

len (map f lst) = len lst

What does this say?
Does it make sense?

List induction

```
fun len []       = 0
  | len (x::xs) = 1 + len xs
fun map f []       = []
  | map f (x::xs) = (f x) :: (map f xs);
```

Facts

Prove: len (map f lst) = len lst

1. State what you're trying to prove!
2. State and prove the base case (often empty list)
3. Assume it's true for sublists – inductive hypothesis
4. Show that it holds for the full list

Base case: lst = []

Want to prove: len (map f []) = len []

Proof?

Prove: len (map f lst) = len lst

```
fun len []       = 0
  | len (x::xs) = 1 + len xs
fun map f []       = []
  | map f (x::xs) = (f x) :: (map f xs);
```

Facts

Base case: lst = []

Want to prove: len (map f []) = len []

definition of map!

Prove: len (map f lst) = len lst

```
fun len []       = 0
  | len (x::xs) = 1 + len xs
fun map f []       = []
  | map f (x::xs) = (f x) :: (map f xs);
```

Inductive hypothesis: len (map f xs) = len xs

Want to prove: len (map f (x::xs)) = len (x::xs)

Proof?

Prove: len (map f lst) = len lst

```
fun len []       = 0
  | len (x::xs) = 1 + len xs
fun map f []       = []
  | map f (x::xs) = (f x) :: (map f xs);
```

Inductive hypothesis: len (map f xs) = len xs

Want to prove: len (map f (x::xs)) = len (x::xs)

len (map f (x::xs)) =	len ((f x) :: (map f xs))	definition of map
	= 1 + len (map f xs)	definition of len
	= 1 + len xs	inductive hypothesis
	= len (x::xs)	definition of len

Done!

```
fun len []       = 0
  | len (x::xs) = 1 + len xs
fun map f []       = []
  | map f (x::xs) = (f x) :: (map f xs);
```

Some list "facts"

1. `[]@v1 = v1`
2. `u1@[] = u1`
3. `(u1@v1)@w1 = u1@(v1@w1)`
4. `[u]@us = u::us`

Another list fact

len (xlst @ ylst) = len xlst + len ylst

What does this say?
Does it make sense?

```
1. []@v1 = v1
2. u1@[] = u1
3. (u1@v1)@w1 = u1@(v1@w1)
4. [u]@us = u::us
```

use induction on xlst

Prove: len (xlst @ ylst) = len xlst + len ylst

1. State what you're trying to prove!
2. State and prove the base case (often empty list)
3. Assume it's true for smaller lists – inductive hypothesis
4. Show that it holds for the current list

Base case: xlst = []

Want to prove: len ([] @ ylst) = len [] + len ylst

Proof?

Prove: len (xlst @ ylst) = len xlst + len ylst

```
1. []@v1 = v1
2. u1@[] = u1
3. (u1@v1)@w1 = u1@(v1@w1)
4. [u]@us = u::us
```

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

Base case: xlst = []

Want to prove: len ([] @ ylst) = len [] + len ylst

len ([] @ ylst) = … = len [] + len ylist

1. start with left hand side
2. show a set of justified steps that derive the right hand size

Prove: len (xlst @ ylst) = len xlst + len ylst

```
1. []@v1 = v1
2. u1@[] = u1
3. (u1@v1)@w1 = u1@(v1@w1)
4. [u]@us = u::us
```

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

Base case: xlst = []

Want to prove: len ([] @ ylst) = len [] + len ylst

len ([] @ ylst) = len ylst — fact 1

= 0 + len ylst — math

= len [] + len ylst — definition of len

Prove: len (xlst @ ylst) = len xlst + len ylst

1. []@v1 = v1
2. u1@[] = u1
3. (u1@v1)@w1 = u1@(v1@w1)
4. [u]@us = u::us

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

Inductive hypothesis: len (xs @ ylst) = len xs + len ylst

Want to prove: len ((x::xs) @ ylst) = len (x::xs) + len ylst

Prove: len (xlst @ ylst) = len xlst + len ylst

Prove: len (xlst @ ylst) = len xlst + len ylst

1. []@v1 = v1
2. u1@[] = u1
3. (u1@v1)@w1 = u1@(v1@w1)
4. [u]@us = u::us

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

Want to prove: len ((x::xs) @ ylst) = len (x::xs) + len ylst

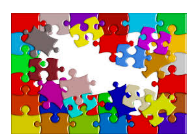
len ((x::xs) @ ylst) =  = len (x::xs) + len ylst

1. []@v1 = v1
2. u1@[] = u1
3. (u1@v1)@w1 = u1@(v1@w1)
4. [u]@us = u::us

len (xs @ ylst) = len xs + len ylst

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

Want to prove: len ((x::xs) @ ylst) = len (x::xs) + len ylst

len ((x::xs) @ ylst) =  = len (x::xs) + len ylst

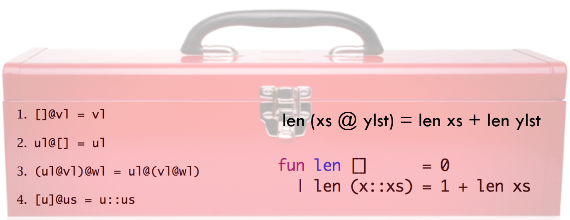
1. []@v1 = v1
2. u1@[] = u1
3. (u1@v1)@w1 = u1@(v1@w1)
4. [u]@us = u::us

len (xs @ ylst) = len xs + len ylst

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

Want to prove: len ((x::xs) @ ylst) = len (x::xs) + len ylst

len ((x::xs) @ ylst) = ? = len (x::xs) + len ylst



Inductive hypothesis: len (xs @ ylst) = len xs + len ylst

Want to prove: len ((x::xs) @ ylst) = len (x::xs) + len ylst

len ((x::xs) @ ylst)	= len (([x]@xs) @ ylst)	fact 4
	= len ([x] @ (xs @ ylst))	fact 3
	= len (x :: (xs @ ylst))	fact 4
	= 1 + len (xs @ ylst)	definition of len
	= 1 + len xs + len ylst	inductive hypothesis
	= len (x::xs) + len ylst	definition of len

1. []@vl = vl
2. ul@[] = ul
3. (ul@vl)@wl = ul@(vl@wl)
4. [u]@us = u::us

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

Blast from the past

```
fun cart []      _  = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

What does the anonymous function do?

Blast from the past

```
fun cart []      _  = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

Takes a value, x, and creates a tuple with u as the first element and x as the second

Blast from the past

```
fun cart []       _  = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

What does the map part of this function do?

Blast from the past

```
fun cart []       _  = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

For each element in vl, creates a tuple (pair) with u as the first element and an element of vl as the second

Blast from the past

```
fun cart []       _  = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

What is the type signature?
What does this function do?

Blast from the past

```
fun cart []       _  = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

4. [2 points] Write a function `cartesian` that takes two lists and forms a list of all the ordered pairs, with one element from the first list and one from the second. For example, `cartesian [1,3,5] [2,4]` will return `[(1,2),(1,4),(3,2),(3,4),(5,2),(5,4)]`.

```
cartesian : 'a list -> 'b list -> ('a * 'b) list
```

Blast from the past

Name the actor and movie

Blast from the past

A property of cart

```
fun cart []      _  = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

len(cart ul vl) = (len ul) * (len vl)

What does this say?
Does it make sense?

A property of cart

```
fun cart []      _  = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

Prove: len(cart ul vl) = (len ul) * (len vl)

Proof by induction. Which variable, ul or vl?

Base case: ulst = []

Want to prove: len (cart [] vl) = (len []) * (len vl)

Proof?

Prove: len(cart ul vl) = (len ul) * (len vl)

1. []@vl = vl
2. ul@[] = ul
3. (ul@vl)@wl = ul@(vl@wl)
4. [u]@us = u::us

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

```
fun cart [] _ = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

Base case: ulst = []

Want to prove: len (cart [] vl) = (len []) * (len vl)

len (cart [] vl) = len []	definition of cart
= 0	definition of len
= 0 * (len vl)	math
= (len []) * (len vl)	definition of len

Prove: len(cart ul vl) = (len ul) * (len vl)

1. []@vl = vl
2. ul@[] = ul
3. (ul@vl)@wl = ul@(vl@wl)
4. [u]@us = u::us

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

```
fun cart [] _ = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

Inductive hypothesis: len (cart us vl) = (len us) * (len vl)

Want to prove: len (cart (u::us) vl) = (len (u::us)) * (len vl)

Prove: len(cart ul vl) = (len ul) * (len vl)

Prove: len(cart ul vl) = (len ul) * (len vl)

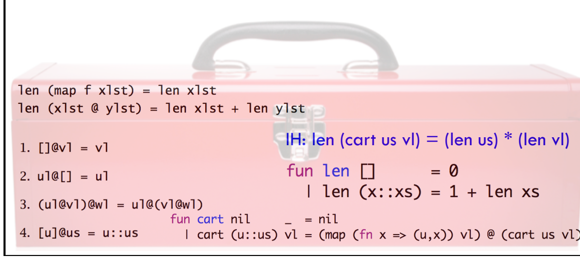
1. []@vl = vl
2. ul@[] = ul
3. (ul@vl)@wl = ul@(vl@wl)
4. [u]@us = u::us

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

```
fun cart [] _ = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

Want to prove: len (cart (u::us) vl) = (len (u::us)) * (len vl)

len (cart (u::us) vl) = ? = (len (u::us)) * (len vl)



len (map f xlst) = len xlst
len (xlst @ ylst) = len xlst + len ylst

1. []@vl = vl
2. ul@[] = ul
3. (ul@vl)@wl = ul@(vl@wl)
4. [u]@us = u::us

IH: len (cart us vl) = (len us) * (len vl)

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

```
fun cart nil _ = nil
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

Want to prove: len (cart (u::us) vl) = (len (u::us)) * (len vl)

len (cart (u::us) vl) = len (map (fn x => (u,x)) vl) @ (cart us vl)) — definition of cart
= len (map (fn x => (u,x)) vl)) + len (cart us vl) — "@" fact
= len (vl) + len(cart us vl) — "map" fact
= len (vl) + (len us) * (len vl) — inductive hypothesis
= (1 + (len us)) * (len vl) — math
= (len (u::us)) * (len vl) — definition of len

```
len (map f xlst) = len xlst
len (xlst @ ylst) = len xlst + len ylst
```

1. []@vl = vl
2. ul@[] = ul
3. (ul@vl)@wl = ul@(vl@wl)
4. [u]@us = u::us

IH: len (cart us vl) = (len us) * (len vl)

```
fun len []      = 0
  | len (x::xs) = 1 + len xs
```

```
fun cart []      _  = []
  | cart (u::us) vl = (map (fn x => (u,x)) vl) @ (cart us vl);
```

Quick refresher: datatypes

```
datatype direction = North | South | East | West;

datatype student = Firstyear of string |
                   Sophomore of string |
                   Junior of string |
                   Senior of string;

datatype cs52int = Pos of int list |
                   Zero |
                   Neg of int list;
```

Recursive datatype

```
datatype 'a binTree =
           Empty
         | Node of 'a binTree * 'a * 'a binTree;
```

- Defines a type variable for use in the datatype constructors
- Still just defines a new type called "binTree"

Recursive datatype

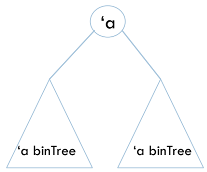
```
datatype 'a binTree =
           Empty
         | Node of 'a binTree * 'a * 'a binTree;
```

What is this?

Recursive datatype

```
datatype 'a binTree =
           Empty
         | Node of 'a binTree * 'a * 'a binTree;
```

Binary Tree!



Recursive datatype

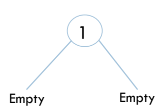
```
datatype 'a binTree =
           Empty
         | Node of 'a binTree * 'a * 'a binTree;
```

Node(Empty, 1, Empty); What does this look like?

Recursive datatype

```
datatype 'a binTree =
           Empty
         | Node of 'a binTree * 'a * 'a binTree;
```

Node(Empty, 1, Empty);



Recursive datatype

```
datatype 'a binTree =
           Empty
         | Node of 'a binTree * 'a * 'a binTree;
```

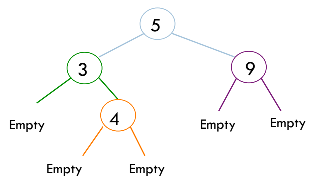
Node(Node(Empty, 3, Node(Empty, 4, Empty)), 5, Node(Empty, 9, Empty));

What does this look like?

Recursive datatype

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Node(Node(Empty, 3, Node(Empty, 4, Empty)), 5, Node(Empty, 9, Empty));



Recursive datatype

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

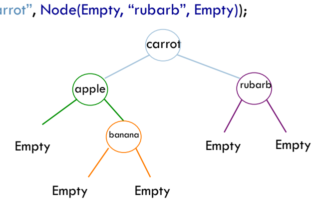
Node(Node(Empty, “apple”, Node(Empty, “banana”, Empty)),
“carrot”,
Node(Empty, “rubarb”, Empty));

What does this look like?

Recursive datatype

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Node(Node(Empty, “apple”, Node(Empty, “banana”, Empty)),
“carrot”, Node(Empty, “rubarb”, Empty));



Facts about binary trees

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Counting elements in a tree N( ):

N(Empty) =

How many Nodes (i.e. values) are in an empty binary tree?

Facts about binary trees

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Counting elements in a tree N( ):

N(Empty) = 0

Facts about binary trees

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Counting elements in a tree N( ):

N(Empty) = 0

N(Node(u, elt, v)) =

How many Nodes (i.e. values) are in non-empty binary tree (stated recursively)?

Facts about binary trees

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Counting elements in a tree N( ):

N(Empty) = 0

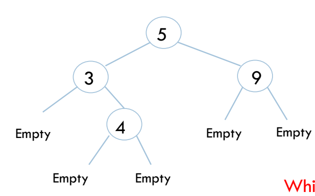
N(Node(u, elt, v)) = 1 + N(u) + N(v)

One element stored in this node plus the nodes in the left tree and the nodes in the right tree

Leaves

A "*leaf*" is a Node at the bottom of the tree, i.e. Node(Empty, elt, Empty)

Node(Node(Empty, 3, Node(Empty, 4, Empty)), 5, Node(Empty, 9, Empty));

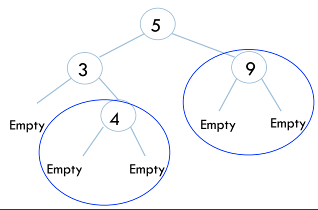


Which are the leaves?

Leaves

A "*leaf*" is a Node at the bottom of the tree, i.e.
Node(Empty, elt, Empty)

Node(Node(Empty, 3, Node(Empty, 4, Empty)), 5, Node(Empty, 9, Empty));



Facts about binary trees

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Counting *leaves* in a tree L( ):

L(Empty) =

L((Empty, elt, Empty) =

L(Node(u, elt, v) =

?

Facts about binary trees

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Counting *leaves* in a tree L( ) :

L(Empty) = 0

L((Empty, elt, Empty) = 1

L(Node(u, elt, v) = L(u) + L(v)

Facts about binary trees

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Counting *Emptys* in a tree E( ):

E(Empty) =

E(Node(u, elt, v) =

?

Facts about binary trees

```
datatype 'a binTree =
        Empty
      | Node of 'a binTree * 'a * 'a binTree;
```

Counting *Emptys* in a tree E( ):

E(Empty) = 1

E(Node(u, elt, v) = E(u) + E(v)

Notation summarized

- N( ): number of elements/values in the tree
- L( ): number of leaves in the tree
- E( ): number of Empty nodes in the tree

Tree induction

1. State what you're trying to prove!
2. State and prove the base case(s) (often Empty and/or Leaf)
3. Assume it's true for smaller subtrees – inductive hypothesis
4. Show that it holds for the full tree

N(t) = E(t) - 1

What is this saying in English?

N(Empty) = 0
N(Node(u, elt, v)) = 1 + N(u) + N(v)

N: number of nodes
L: number of leaves
E: number of Emptys

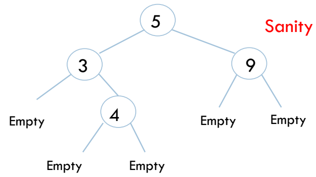
E(Empty) = 1
E(Node(u, elt, v) = E(u) + E(v)

L(Empty) = 0
L((Empty, elt, Empty) = 1
L(Node(u, elt, v) = L(u) + L(v)

N(t) = E(t) - 1

Number of nodes/values is equal to number of Emptys minus one

Sanity check: is it right here?



N(Empty)	= 0
N(Node(u, elt, v))	= 1 + N(u) + N(v)

N: number of nodes
L: number of leaves
E: number of Emptys

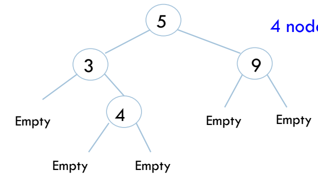
E(Empty)	= 1
E(Node(u, elt, v)	= E(u) + E(v)

L(Empty)	= 0
L((Empty, elt, Empty)	= 1
L(Node(u, elt, v)	= L(u) + L(v)

N(t) = E(t) - 1

Number of nodes/values is equal to number of Emptys minus one

4 nodes = 5 Emptys - 1



N(Empty)	= 0
N(Node(u, elt, v))	= 1 + N(u) + N(v)

N: number of nodes
L: number of leaves
E: number of Emptys

E(Empty)	= 1
E(Node(u, elt, v)	= E(u) + E(v)

L(Empty)	= 0
L((Empty, elt, Empty)	= 1
L(Node(u, elt, v)	= L(u) + L(v)

Base case: t = Empty

Want to prove: N(Empty) = E(Empty) - 1

Proof?

Prove: N(t) = E(t) - 1

N(Empty)	= 0
N(Node(u, elt, v))	= 1 + N(u) + N(v)

N: number of nodes
L: number of leaves
E: number of Emptys

E(Empty)	= 1
E(Node(u, elt, v)	= E(u) + E(v)

L(Empty)	= 0
L((Empty, elt, Empty)	= 1
L(Node(u, elt, v)	= L(u) + L(v)

Base case: t = Empty

Want to prove: N(Empty) = E(Empty) - 1

N(Empty) = 0

E(Empty) = 1 - 1 = 0

Prove: N(t) = E(t) - 1

N(Empty)	= 0
N(Node(u, elt, v))	= 1 + N(u) + N(v)

N: number of nodes
L: number of leaves
E: number of Emptys

E(Empty)	= 1
E(Node(u, elt, v)	= E(u) + E(v)

L(Empty)	= 0
L((Empty, elt, Empty)	= 1
L(Node(u, elt, v)	= L(u) + L(v)

Inductive hypotheses: N(u) = E(u) - 1
N(v) = E(v) - 1 (Relation holds for any subtree)

Want to prove: N(Node(u, elt, v)) = E(Node(u, elt, v)) - 1

Prove: N(t) = E(t) - 1

N(Empty) = 0
N(Node(u, elt, v)) = 1 + N(u) + N(v)

N: number of nodes
L: number of leaves
E: number of Emptys

E(Empty) = 1
E(Node(u, elt, v) = E(u) + E(v)

L(Empty) = 0
L((Empty, elt, Empty) = 1
L(Node(u, elt, v) = L(u) + L(v)

Want to prove: N(Node(u, elt, v)) = E(Node(u, elt, v)) - 1

N(Node(u, elt, v)) = ? = E(Node(u, elt, v)) - 1

N(u) = E(u) - 1
N(v) = E(v) - 1

N(Empty) = 0
N(Node(u, elt, v)) = 1 + N(u) + N(v)

N: number of nodes
L: number of leaves
E: number of Emptys

E(Empty) = 1
E(Node(u, elt, v) = E(u) + E(v)

L(Empty) = 0
L((Empty, elt, Empty) = 1
L(Node(u, elt, v) = L(u) + L(v)

Want to prove: N(Node(u, elt, v)) = E(Node(u, elt, v)) - 1

N(Node(u, elt, v)) = 1 + N(u) + N(v) "N" fact
= 1 + E(u) - 1 + E(v) - 1 inductive hypothesis
= E(u) + E(v) - 1 math
= E(Node(u, elt, v)) - 1 "E" fact

N(u) = E(u) - 1
N(v) = E(v) - 1

N(Empty) = 0
N(Node(u, elt, v)) = 1 + N(u) + N(v)

N: number of nodes
L: number of leaves
E: number of Emptys

E(Empty) = 1
E(Node(u, elt, v) = E(u) + E(v)

L(Empty) = 0
L((Empty, elt, Empty) = 1
L(Node(u, elt, v) = L(u) + L(v)

Other interesting tree facts

N(t) = E(t) - 1

N(Empty) = 0
N(Node(u, elt, v)) = 1 + N(u) + N(v)

N: number of nodes
L: number of leaves
E: number of Emptys

E(Empty) = 1
E(Node(u, elt, v) = E(u) + E(v)

L(Empty) = 0
L((Empty, elt, Empty) = 1
L(Node(u, elt, v) = L(u) + L(v)

Summary of induction proofs

Numbers: $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Recurrence relations:

$$count_0(k) = \frac{k(k+1)}{2} \qquad count_1(k) = 2^{k+1} - k - 2$$

Code equivalence:

fibrec(n) = fibiter(n)

Induction on lists:

len (map f xlst) = len xlst   len (xlst @ ylst) = len xlst + len ylst

len(cart ul vl) = (len ul) * (len vl)

Induction on trees:

N(t) = E(t) - 1

Be careful!

Outline for a "good" proof by induction

1. Prove: *what_to_prove*

2. Base case: *the_base_case(s)*
   step by step proof
   with each step clearly justified

3. Assuming: *the_inductive_hypothesis*
4. Show: *what_you're_trying_to_prove*
   step by step proof
   with each step clearly justified