List and Tree Induction

March 31st / April 2nd, 2020

1 Reasoning About Trees

Some properties of trees and some tree algorithms are quite subtle. It is useful to have a formal approach to mathematically reason about trees and their properties. For example, we might want to verify the correctness of the following claims and solve the puzzle. (Detailed definitions given in the Sec. 6.)

Claim I. A perfect binary tree of height h has $2^{h+1} - 1$ nodes.

Claim II. A full binary tree with n nodes has a longest path of length $n - 1$.

Claim III. A non-empty full binary tree with n internal nodes has $n + 1$ leaves.

Chain Letter Puzzle. Suppose someone starts a chain letter. Each person who receives the letter is asked to send it on to four people. Some people do this, but others do not send any letters. Assume that everyone who forwarded the letter forwarded it to exactly four people and that all the people are distinct. Furthermore, assume that the chain letter ended after 100 people received it but did not send it out. How many people have seen the letter? How many people sent out the letter?

In the next section, we will introduce the powerful principle of tree induction that can be used to prove many properties of trees. This principle is the mathematical side of recursion in the sense that the structure of mathematical proofs about trees will follow the structure of recursive programs on trees. Because tree induction is a bit more complicated than list induction, we will also discuss induction on lists as a warmup.

2 List and Tree Induction Proof Principles

In the accompanying Java implementation, lists and binary trees are defined as follows:

```
abstract class List

    class EmptyL extends List

    class NodeL extends List
      private int x;
      private List xs;
      NodeL (int x, List xs)
```

```
abstract class BinTree

    class Leaf extends BinTree
      private int d;
      Leaf (int d)

    class NodeT extends BinTree
      private int d;
      private BinTree t1, t2;

      NodeT (int d, BinTree t1, BinTree t2)
```

For convenience, we will abbreviate `new EmptyL()` as [] and `new NodeL(x,xs)` as $x : xs$. So for example lists can be constructed by writing expressions like $0 : 1 : 2 : []$ instead of:

`new NodeL(0,new NodeL(1,new NodeL(2,new EmptyL())))`

Similarly, we will abbreviate `new Leaf(d)` as just d and `new NodeT(d,t1,t2)` as $N(d, t_1, t_2)$.

The associated induction principles for lists and trees can then be stated as follows.

List Induction. To prove a property P of lists, there are two obligations:

- prove $P([\,])$, and
- assuming $P(xs)$ holds for arbitrary lists, prove for arbitrary integer x that $P(x : xs)$ also holds.

Tree Induction. To prove a property P of trees, there are two obligations:

- prove $P(d)$ for arbitrary integers d, and
- assuming $P(t_1)$ and $P(t_2)$ hold for arbitrary trees, prove for arbitrary integer d that $P(N(d, t_1, t_2))$ also holds.

3 A Simple List Example

We consider a small recursive program on lists and use list induction to prove two simple properties about it. In the accompanying Java code, `append` is defined as follows:

```
abstract List append (List ys);

// in EmptyL
List append (List ys) {
    return ys;
}

// in NodeL
List append (List ys) {
    return new NodeL(x, xs.append(ys));
}
```

In our shorthand notation, the program would be written as follows:

$$\begin{aligned} [\,].append(ys) &= ys && \text{(A.1)} \\ (x : xs).append(ys) &= x : (xs.append(ys)) && \text{(A.2)} \end{aligned}$$

where we have labeled the the two code fragments for future reference.

Proof Example I.

Now let us prove a very simple property. For arbitrary lists `zs`, we have `zs.append(new EmptyL())` is identical to `zs`. In our shorthand notation, the property to prove would be:

$$zs.append([\,]) = zs$$

By construction, an arbitrary list zs is either empty or of the form $x : xs$ for some arbitrary x and xs. Following the principle of list induction, our proof obligation then splits into two obligations:

- Prove $[\,].append([\,]) = [\,]$ which is called the *base case*.
- Assuming that for arbitrary xs, we have $xs.append([\,]) = xs$, then prove $(x : xs).append([\,]) = (x : xs)$. The assumption that $xs.append([\,]) = xs$ is called the *inductive hypothesis* (IH) and the full proof obligation is called the *inductive case*.

Base case. We would like to prove $[\,].append([\,]) = [\,]$. This follows immediately by A.1. If you are confused about why this follows immediately, remember that A.1 refers to the implementation of the method `append` in the class `EmptyL` and that calling this method with any list immediately returns it. In particular, calling the method with $[\,]$ immediately returns $[\,]$.

Inductive case. By IH, we can assume $xs.append([\,]) = xs$ for arbitrary lists xs. We now want to prove $(x : xs).append([\,]) = x : xs$. We will establish that identity by manipulating the left-hand side (LHS) and simplifying it until it becomes identical to the right-hand side (RHS):

$$\begin{array}{rcll} LHS & = & (x : xs).append([\,]) & \\ & = & x : (xs.append([\,])) & \text{(by A.2)} \\ & = & x : xs & \text{(by IH)} \\ & = & RHS & \end{array}$$

Proof Example II.

Let us prove that for arbitrary lists xs, ys, and zs, it is the case that:

$$xs.append(ys.append(zs)) = (xs.append(ys)).append(zs)$$

Now we have three lists which we can analyze to do our proof. Remembering that the proof should typically follow the structure of the code, we reason that the relevant list for the execution of this program is xs, so we structure our proof to be by induction on xs:

- Base case: $xs = [\,]$:

$$\begin{array}{rcll} LHS & = & [\,].append(ys.append(zs)) & \\ & = & append(ys.append(zs)) & \text{(by A.1)} \\ & & & \\ RHS & = & ([\,].append(ys)).append(zs) & \\ & = & append(ys).append(zs) & \text{(by A.1)} \end{array}$$

- Inductive case: $xs = x : xs'$. We can assume:

  $$xs'.append(ys.append(zs)) = (xs'.append(ys)).append(zs)$$

  which is the IH. We want to prove:

  $$(x : xs').append(ys.append(zs)) = ((x : xs').append(ys)).append(zs)$$

$$\begin{array}{rcll} LHS & = & (x : xs').append(ys.append(zs)) & \\ & = & x : (xs'.append(ys.append(zs))) & \text{(by A.2)} \\ & = & x : (xs'.append(ys)).append(zs) & \text{(by IH)} \\ & & & \\ RHS & = & ((x : xs').append(ys)).append(zs) & \\ & = & (x : (xs'.append(ys))).append(zs) & \text{(by A.2)} \\ & = & x : (xs'.append(ys)).append(zs) & \text{(by A.2)} \end{array}$$

4 A Simple Tree Example

Hre are two simple recursive functions on trees that count the numbers of nodes and the number of edges:

```
abstract int nodes ();

// in Leaf
int nodes() { return 1; }

// in NodeT
int nodes() {
    return 1 + t1.nodes() + t2.nodes();
}
```

```
abstract int edges ();

// in Leaf
int edges () { return 0; }

// in NodeT
int edges () {
  return 2 + t1.edges() + t2.edges();
}
```

The number of nodes in a tree is clear: we count the leaf as a node and when we encounter a tree with two subtrees, we add the number of nodes in each subtree and add 1 for the current node. The number of edges counts the number of connections from the current node going towards all its children. A leaf has no children and hence no edges. For a node with two subtrees, we add the number of edges from each subtree and add 2 for the two connections to the children. Since every node (except the root) has an edge to its parent, it should be the case that:

$$t.nodes() = t.edges() + 1$$

Let's proceed by induction on t:

- $t = d$

$$\begin{array}{rcl} LHS & = & d.nodes() = 1 \\ RHS & = & d.edges() + 1 = 0 + 1 = 1 \end{array}$$

- $t = N(d, t_1, t_2)$:

$$\begin{array}{rcll} LHS & = & N(d,t_1,t_2).nodes() & \\ & = & 1 + t_1.nodes() + t_2.nodes() & \\ & = & 1 + (t_1.edges() + 1) + (t_2.edges() + 1) & \text{by two IHs} \\ & = & t_1.edges() + t_2.edges() + 3 & \end{array}$$

$$\begin{array}{rcl} RHS & = & N(d,t_1,t_2).edges() + 1 \\ & = & 2 + t_1.edges() + t_2.edges() + 1 \\ & = & t_1.edges() + t_2.edges() + 3 \end{array}$$

5 More Advanced List Examples

The implementations of the methods below are in the accompanying code. We would like to prove the following properties for arbitrary lists xs, ys, and zs. We use id to refer to the identity function that returns its argument unchanged:

1. $xs.map(id) = xs$
2. $(xs.append(ys)).map(f) = (xs.map(f)).append(ys.map(f))$
3. $(xs.append(ys)).fold(g, a) = xs.fold(g, ys.fold(g, a))$
4. $(xs.append(ys)).length() = xs.length() + ys.length()$
5. $(xs.reverseH(ys)).length() = xs.length() + ys.length()$
6. $xs.length() = (xs.reverse2()).length()$
7. $(xs.append(ys)).reverse() = ys.reverse().append(xs.reverse())$
8. $(xs.reverse()).reverse() = xs$

9. $xs.reverseH(ys) = (xs.reverse()).append(ys)$
10. $(xs.reverseH(ys)).reverseH(zs) = ys.reverseH(xs.append(zs))$
11. $xs.reverseH(ys.append(zs)) = (xs.reverseH(ys)).append(zs)$
12. $(xs.append(ys)).reverseH(zs) = ys.reverseH(xs.reverseH(zs))$
13. $(xs.append(ys)).reverse2() = ys.reverse2().append(xs.reverse2())$
14. $(xs.reverse2()).reverse2() = xs$

6 More Advanced Tree Examples

6.1 Trees and Lists

The implementations of the methods below are in the accompanying code. We would like to prove the following properties for arbitrary trees t and lists xs. We use $f1$ for the function that always returns 1:

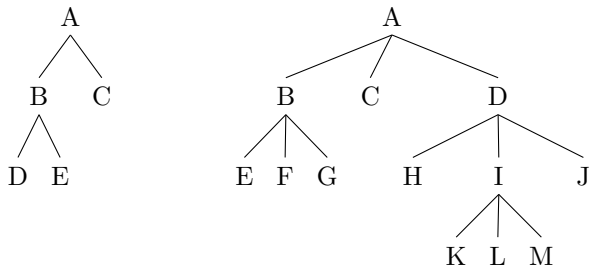
1. $t.flattenH(xs) = t.flatten().append(xs)$
2. $t.flatten2() = t.flatten()$
3. $t.map(f1).sum() = t.nodes()$
4. $t.nodes() = t.longestPath().length() + 1$.
5. For non-empty trees t, it is the case that $t.internalNodes() + 1 = t.leaves()$.

6.2 Chain Letter

A node that is not a leaf is called an *internal node.* A tree is called an m-ary tree if every internal node has no more than m children. The tree is called a *full* m-ary tree if every internal node has exactly m children. The following statements about a full m-ary tree are true:

- A full m-ary with n nodes has $(n-1)/m$ internal nodes and $((m-1)n+1)/m$ leaves.
- A full m-ary with i internal nodes has $mi+1$ nodes and $(m-1)i+1$ leaves.
- A full m-ary with l leaves has $(ml-1)/(m-1)$ nodes and $(l-1)/(m-1)$ internal nodes.

For example, consider the following trees:



The tree on the left is a full 2-ary tree: it has 5 nodes, 3 of them D, E, and C are leaves and two of them A and B are internal nodes. In other words, we have $m = 2$, $l = 3$, $i = 2$, and $n = 5$. The tree on the right is a full 3-ary tree: it has 13 nodes, 9 leaves, and 4 internal nodes. Please convince yourself that the identities above are correct in these two cases.

Now suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to four other people. Some people do this, but others do not send any letters. Assume that everyone who forwarded the letter forwarded it to exactly four people and that all the people are distinct. Furthermore, assume that the chain letter ended after 100 people received it but did not send it out.

1. Prove the three statements above about m-ary trees.
2. How many people have seen the letter, including the first person?
3. How many people sent out the letter?