CSCI 136
Data Structures &
Advanced Programming

Recursion & Induction
on Trees

Recursion & Induction on Trees

Reasoning About Trees

Recall: A BinaryTree T is either

- Empty, or
- Consists of a root along with two BinaryTrees
  - Left and right subtrees of T

If both the left and right subtrees of T are empty, we call T a leaf

How do we establish properties of trees and algorithms on trees?

- Induction!

An Example

Prove

Number of nodes at depth $d \geq 0$ is at most 2^d.

Idea: Induction on depth d of nodes of tree

Base case: $d = 0$: 1 node. $1 = 2^0$ ✓

Induction Hyp.: For some $d \geq 0$, there are at most 2^d nodes at depth d.

Induction Step: Consider depth $d+1$. There are at most 2 child nodes at depth $d+1$ for every node at depth d

Therefore There are at most $2 * 2^d = 2^{d+1}$ nodes ✓

Strong Induction!

Often, we'll need to use strong induction

Principle of Strong Induction

Let P_0, P_1, P_2, ... be a sequence of statements, each of which could be either true or false. Suppose that, for some $k \geq 0$

- P_0, P_1, ..., P_k are true, and
- For every $n \geq k$, if P_0, P_1, ..., P_n are true, then P_{n+1} is true

Then *all* of the statements are true!

Why?

- Induction is often on size or height of tree
- Sizes/heights of subtrees can be *much smaller than* those of the tree

Example : Correctness of size()

Recall the size() method for BinaryTree

```
// Returns the number of descendants of node
public int size() {
    if (isEmpty()) return 0;
    return left().size() + right().size() + 1;
}
```

Let's try to prove that size() works correctly

Proof: Induction on number of descendants of node

- Note: Node is descendent of itself!

Base Case: n = 0 (Empty tree!)

- method correctly returns 0 ✓

Example : Correctness of size()

```
// Returns the number of descendants of node
public int size() {
    if (isEmpty()) return 0;
    return left().size() + right().size() + 1;
}
```

Induction Hypothesis

For some $n \geq 1$, size() correctly returns the number of descendants of a node for all nodes with at most n-1 descendents.

Induction Step

Now show that if node has n descendants, then size() returns the correct value.

Example : Correctness of size()

Induction Step

Now show that if `node` has n descendants, then `size`() returns the correct value.

Proof

- Since $n \geq 1$, the second return statement is executed

```
return left().size() + right().size() + 1;
```

- Since each of `left` and `right` have *fewer than* n nodes, by the I.H., `size`() returns the correct number of descendants of `left` and `right`
- Adding them, plus 1 for `node` itself, gives the correct number of descendants of `node` ✓

Practice Problems

Prove

- The number of nodes at depth n is at most 2^n. ✓
- The number of nodes in tree of height n is at most $2^{(n+1)}-1$.
- The size() method works correctly ✓
- The height() method works correctly
- The isFull() method works correctly
- The isComplete() method works correctly
- Evaluate correctly evaluates an expression tree

Correctness of height() Method

```
// Returns the height of node
public int height() {
  if (isEmpty()) return -1;
  return 1 + Math.max(left.height(),right.height());
}
```

Proof: Induction on the height of `node`

Base Case: h = -1 (Empty tree!)

- method correctly returns -1 ✓

Induction Hypothesis

For some $h \geq 0$, height() correctly returns the height of `node` for all nodes with height at most h-1.

Correctness of height() Method

Induction Step

Now show that if **`node`** has height h, then height() returns the correct height value for **`node`**.

Proof

- Since h > -1, the second return statement is executed

```
return 1 + Math.max(left.height(),right.height());
```

- Since each of `left` and `right` have *height at most h-1*, by the I.H., `height`() returns the correct heights of `left` and `right`
- The height of **`node`** is then one more than the larger of the heights of `left` and `right`
- This is exactly the value returned by the method ✓

Summary and Observations

Binary trees are naturally recursive structures

- Many tree algorithms are recursive
- Establishing properties of such algorithms is often done via induction
  - Typically using *strong induction*
  - Induction is frequently based on size or height of the tree

Practice with recursion and induction on trees will improve programming/design skills