COMPSCI 230: Discrete Mathematics for Computer Science	February 25, 2019
Lecture 12	
Lecturer: Debmalya Panigrahi	*Scribe: Erin Taylor*

1 Overview

In this lecture, we introduce the idea of structural induction. We define recursive data structures and use structural induction to prove properties of these data structures.

2 Structural Induction and Recursive Data Structures

So far we have been thinking of induction as a "bottom up" approach for proving a predicate for all natural numbers. First we directly prove a base case, and then we begin to build up. Suppose we wanted to prove $P(n)$ for all natural numbers n. We first proved a base case such as $P(1)$, then we prove that for all n, $P(n) \to P(n+1)$. Thus, we prove $P(1)$ is true and use the implication to show $P(1) \to P(2)$. Now that we've established $P(2)$, we can show $P(3)$ since $P(2)$ implies $P(3)$. In summary, the logic for proving $P(n)$ is true proceeds as follows:

$$P(1) \text{ is true, and } P(1) \to P(2) \to P(3) \to \cdots \to P(n) \text{ is true, so } P(n) \text{ is true.}$$

We can think of this chain of logic in the opposite direction as well (a "top down" approach). If we want to prove $P(n)$ for some arbitrary n, and we know that $P(n-1) \to P(n)$ then we should focus on proving $P(n-1)$. We are using the same logic, but thinking in a more recursive manner. Induction is a more general technique, and can prove predicates for different domains. For instance, if we wanted to prove something about the even positive integers then our base case would be $P(2)$ and we would show $P(n) \to P(n+2)$ for all n to conclude the following:

$$P(2) \text{ is true, and } P(2) \to P(4) \to \cdots \to P(n-2) \to P(n) \text{ is true, so } P(n) \text{ is true.}$$

So far, we have been proving predicates defined on natural numbers, but such a logical chain is even more general. Suppose we had a data structure S. We want to show a property holds for this data structure: $P(S)$. Suppose we have an element $a \in S$, and we know the property holds for $S \setminus \{a\}$. Then we want to show $P(S \setminus \{a\}) \to P(S)$. For example, S could be a stack and a could be the element at the top of the stack. If we want to prove a property $P(S)$, we could do so as follows:

$$P(\{e_1\}) \text{ is true, and } P(\{e_1\}) \to P(\{e_1, e_2\}) \to \cdots \to P(S \setminus \{e_n\}) \to P(S) \text{ is true, so } P(S) \text{ is true.}$$

We see that our base case is directly showing $P(S)$ holds if S has a single element, and then we show implications increasing the number of elements in the stack until we arrive at a stack with n elements. This technique is known as *structural induction*, and is induction defined over the domain of recursively defined structures.

Next, we will study two concrete examples of data structures defined recursively and use structural induction to prove properties of these structures. Definitions of recursive data structures consist of two parts:

1. Base Case: The base case specifies an initial, or minimal element.
2. Constructor Case: The constructor case specifies how to construct new instances of the data structure from base instances or previously constructed instances.

2.1 Binary Strings

A binary string is a sequence of 0s and 1s. Alternatively, we can define binary strings recursively.

Binary Strings:

- **Base Case**: The empty string λ is a binary string.
- **Constructor Case**: If s is a binary string, then $a.s$ is a binary string where $a \in \{0, 1\}$.

We can use this definition to create arbitrary binary strings. To convince ourselves that this is a valid definition of binary strings, we can prove some properties that should be true about binary strings and only use the above definition. Let $|s|$ denote the length of string s. We can define the length of binary strings as follows:

Length:

- **Base Case**: $|\lambda| = 0$.
- **Constructor Case**: $|a.s| = 1 + |s|$ where $a \in \{0, 1\}$.

Next, let's define a recursive operation for string concatenation. This function takes binary strings s and t and outputs $s.t$.

Concatenation:

- **Base Case**: If $s = \lambda$, then $s.t = t$.
- **Constructor Case**: If $s \neq \lambda$, then let $s = a.s'$ where $a \in \{0, 1\}$ by the constructor case of the binary string definition. Then, we define $s.t = (a.s').t = a.(s'.t)$.

Now we want to show that the function we defined actually concatenates strings.

Claim 1. *If s is a binary string, then $s.\lambda = s$.*

This predicate should the correct output of concatenating a string with an empty string. Our predicate $P(s)$ is $s.\lambda = s$. We will prove this property using structural induction and our definition of **concatenation**.

Proof of Claim 1. **Base Case**: Our base case is $s = \lambda$. We want to show $P(\lambda) : \lambda.\lambda = \lambda$. To show this, we will use the base case of the definition of string concatenation. If $s = \lambda$, then $s.t = t$. Here, $t = \lambda$, so $\lambda.\lambda = \lambda$ as desired.

Inductive Case: Let $s = a.s'$ where $a \in \{0, 1\}$. We want to show $P(s') \to P(s)$. For our inductive hypothesis, we assume that $P(s')$ is true. Now,

$$\begin{aligned} s.\lambda &= (a.s').\lambda && \text{by the definition of } S. \\ &= a.(s'.\lambda) && \text{plugging in } t = \lambda \text{ in the constructor case for concatenation.} \\ &= a.s' && \text{by the inductive hypothesis.} \\ &= s && \text{by definition of s.} \end{aligned}$$

Thus, we have proved $P(\lambda)$ and that for any $s = a.s'$, $P(s') \to P(s)$. Therefore, the claim holds for any binary string s. □

Claim 2. *If s and t are binary strings, then* $|s.t| = |s| + |t|$.

Proof. **Base Case:** Our base case will be $s = \lambda$. Then we have $|\lambda.t| = |t|$ by the base case of string concatenation. Thus, $|\lambda.t| = |t| = 0 + |t| = |\lambda| + |t|$. Thus, the claim holds for the base case.
Inductive Case: Let $s = a.s'$ where $a \in \{0, 1\}$. We want to show that $P(s') \to P(s)$. We assume that $P(s')$ is true, that is $|s'.t| = |s'| + |t|$. Now,

$$\begin{aligned} |s.t| &= |(a.s').t| && \text{by definition of } s \\ &= |a.(s'.t)| && \text{by constructor case for concatenation} \\ &= 1 + |s'.t| && \text{by constructor case for length} \\ &= 1 + |s'| + |t| && \text{by inductive hypothesis} \\ &= (1 + |s'|) + |t| && \text{rearranging} \\ &= |a.s'| + |t| && \text{by constructor case for length} \\ &= |s| + |t| && \text{by definition of } s \end{aligned}$$

□

2.2 Rooted Trees

Now let's create a recursive definition of rooted trees.

Rooted Tree

- Base Case: A single node that acts as the root.
- Constructor Case: A node called a root whose children are roots of rooted trees themselves.

See Figure 1 for an example of a rooted tree. The root has 4 children, each who is the root of another rooted tree (in Figure 1 these are T_1, T_2, T_3, T_4).

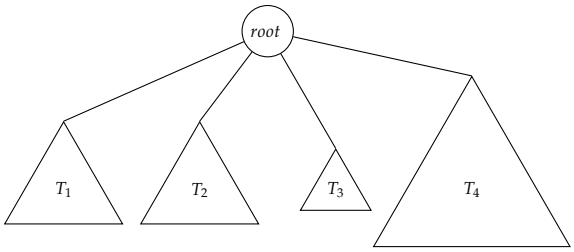

Figure 1: An example of a rooted tree.

Let's prove a property of rooted trees using our inductive definition.

Claim 3. *Any rooted tree with n nodes has $n-1$ edges.*

Proof. **Base Case:** Our base case is when the rooted tree has 1 node. In this case, the tree has no edges. Thus, $n = 1$ and the number of edges is $0 = 1 - 1 = n - 1$. The claim is true for the base case.
Inductive Case: Suppose we have tree T with a root r. Suppose r has k children. Let the trees rooted at the children of r be $T_1, \ldots, T_k$. Let the number of edges of each T_i be m_i and the number of nodes n_i ($i = 1, \ldots k$). For our inductive hypothesis, we assume for every T_i, $m_i = n_i - 1$. Now consider tree T. Let m be the number of edges in T. There are k edges from the root to each of its k children, and then we need to count the edges in each T_i:

$$\begin{aligned}
m &= k + \sum_{i=1}^{k} m_i \\
&= \sum_{i=1}^{k} (1 + m_i) && \text{moving the } k \text{ term into the sum} \\
&= \sum_{i=1}^{k} (1 + (n_i - 1)) && \text{by the inductive hypothesis} \\
&= \sum_{i=1}^{k} n_i && \text{simplifying} \\
&= n - 1 && \text{since this is all nodes except the root}
\end{aligned}$$

□

2.3 Binary Trees

A binary tree is a rooted tree where each node has at most 2 children. Just as rooted trees, we can define binary trees recursively.

Rooted Binary Tree

- Base Case: A single vertex that acts as the root.
- Constructor Case: A vertex which is the root that has one or two children each of which is the root of a binary tree.

Now, let's define the depth (height) of a binary tree.

Depth of a Binary Tree

- Base Case: A tree with only a single vertex has depth 1.
- Constructor Case: Suppose we have tree T where the root has either one child that is the root of a binary tree T_1, or two children that are the roots of binary trees T_1 and T_2. In the first case, $depth(T) = 1 + depth(T_1)$ and in the latter case, $depth(T) = 1 + \max\{depth(T_1),\ depth(T_2)\}$. For brevity, if the root has only a single child, we will say that tree T_2 is empty and assume that its depth is 0.

We have the following theorem relating the depth of a binary tree and the number of nodes in the tree.

Theorem 4. *A binary tree of depth d has at most $2^d - 1$ vertices.*

Proof. Let the number of nodes in a binary tree T be $|T|$.
Base Case: The base case is when we have a binary tree with one vertex. The depth is $1 = d$, and $2^d - 1 = 2 - 1 = 1$. In this case, the theorem holds.
Inductive Case: Suppose T is a binary tree of depth $d + 1$. Our inductive hypothesis is that any binary tree of depth $d' < d + 1$ has at most $2^{d'} - 1$ nodes. Let T_1 and T_2 denote the two subtrees rooted at the two children of the root of T. (Note that it is possible for T_1 or T_2 to be empty, but not both). Now notice that the vertices of T include the root, the vertices in T_1, and the vertices in T_2. So, $|T| = 1 + |T_1| + |T_2|$. Now recall that the depth of T is the following:

$$depth(T) = 1 + \max\{depth(T_1),\ depth(T_2)\}.$$

This implies $depth(T_1) \leq d$ and $depth(T_2) \leq d$. Thus, we can apply our inductive hypothesis on T_1 and T_2.

$$|T| = 1 + |T_1| + |T_2| \leq 1 + 2^d - 1 + 2^d - 1 = 2^{d+1} - 1.$$

Thus, the theorem holds for all binary trees. □

3 Summary

In this lecture, we introduced structural induction and used it to prove properties of recursively defined data structures like binary strings, rooted trees, and binary trees.