CMPSCI 187:Programming With Data Structures

Lecture #26: Trees: Definition and Implementation
16 November 2011

Trees: Definition and Implementation

- Tree Vocabulary
- Basic Strategies to Implement Trees
- Ways to Traverse a Tree
- Implementing Trees with Links
- Implementing Trees with Arrays

Tree Vocabulary

- Trees are made up of **nodes**. The key property of a tree is that every node except for one, the **root**, has exactly one **parent**. We use genealogical language throughout, so that children of the same node are **siblings**, a node that can be reached by following parent pointers is an **ancestor**, and a node whose ancestor you are is your **descendent**.

- Nodes with no children are called **leaves**, and nodes with children are **internal nodes**. The root is at **level 0**, and the **level** of any other node is the number of parent pointers you must follow to reach the root. The **height** of a tree is the largest level number of any node in it.

- A tree is **binary** if every internal node has either one or two children, and it is **k-ary** if every node has at most k children. L&C often implicitly assume that every internal node has *exactly* two children, but this isn't a math course...

Kinds of Binary Trees

- A binary tree is called **balanced** in L&C if its leaves are all on one level or on two adjacent levels. A balanced tree of height h has at most 2^h leaves, and the fewest it can have (with no unary nodes) is 2^{h-1} + 1, still O(2^h). The height of a balanced tree with n leaves, or with n nodes, is O(log n).

- L&C call a binary tree **complete** if it is balanced and the nodes are left-justified. Complete binary trees will turn out to have a convenient representation as arrays.

- L&C call a binary tree **full** if it is complete, has all its leaves on the same level, and has two children for each internal node. (This is often called a “complete binary tree” elsewhere.) Such a tree of height h has exactly 2^h leaves, and since a tree with n leaves has n - 1 internal nodes, has exactly 2^{h+1} - 1 total nodes

- These concepts extend naturally to k-ary trees, with k^h leaves for height h.

Basic Strategies to Implement Trees

- The most obvious way to implement trees is for each node to link to its parents and children, just as the nodes in a doubly linked list link to the preceding and following nodes. But especially with very large trees, there are advantages to representing trees as arrays -- they are more likely to remain in the fast memory of the machine. But how do we represent a non-linear tree in a linear structure like an array?

- If we number the root 0 and define the children of any node i to be nodes 2i + 1 and 2i + 2, and the parent of node i to be node (i - 1)/2 (unless i = 0), we get a tree structure on the numbers {0,...,n-1}. This turns out to be what L&C call a complete binary tree. By leaving gaps in the array we can get unbalanced trees as well.

- If the nodes are numbered 0 to n - 1, we can store pointers as `int` values in an array, simulating the links we would have with Java pointers. This can make it easier, for example, to read or write the whole tree to or from a file.

Ways to Traverse a Tree

- There are several ways to traverse the nodes of a binary tree in order -- this will be the subject of next week's Discussion #10 but we'll define/review them here. Recursion is the easiest way to define them.

- Perhaps the most natural order is left-to-right or **inorder**. We visit the nodes of the left subtree, then visit the root, then visit the nodes of the right subtree. Of course visiting the left subtree means visiting its left subtree, root, and right subtree in that order, and so on.

- The **preorder** and **postorder** traversals are similar. In preorder we visit the root, then the left subtree, then the right subtree. In postorder the root comes last. These correspond to prefix, infix, and postfix notation for expressions, as we'll see in the next lecture.

- **Level order** visits the root, then all nodes at level 1, then all at level 2, etc.

Implementing Trees With Links

- This gives the idea -- note L&C miscalls the last method `numChildren`.

```
public class LinkedBinaryTree<T> {
   protected BinaryTreeNode<T> root;
   protected int count;
   public LinkedBinaryTree( ) {count = 0; root = null;}
   public LinkedBinaryTree (T element) {
      count = 1; root = new BinaryTreeNode<T> (element);}}

public class BinaryTreeNode<T> {
   protected T element;
   protected BinaryTreeNode<T> left, right;
   public BinaryTreeNode (T obj) {
      element = obj; left = right = null;}
   public int descendants ( ) {
      int ret = 0;
      if (left != null) ret = 1 + left.descendants( );
      if (right != null) ret += 1 + right.descendants( );}}
```

Finding an Element in a Linked Tree

- We use a helper method and a simple recursive definition -- the target is in the tree if it is at the root, in the left subtree, or in the right subtree.

```
public T find (T target) throws ElementNotFoundException {
   BinaryTreeNode<T> current = findAgain (target, root);
   if (current == null) throw new ElementNotFoundException( );
   return current.element;}

private BinaryTreeNode<T>
      findAgain (T target, BinaryTreeNode<T> next) {
   if (next == null) return null;
   if (next.element.equals (target)) return next;
   BinaryTreeNode<T> temp = findAgain (target, next.left);
   if (temp == null) temp = findAgain (target, next.right);
   return temp;}
```

Implementing Trees With Arrays

- The find operation is a simple linear search, as the array is unordered.

```
public class ArrayBinaryTree {
   protected int count;
   protected T[ ] tree;
   public ArrayBinaryTree( ) {
      count = 0; tree = (T[ ]) new Object [capacity];
   public ArrayBinaryTree (T element) {
      super( ); count = 1; T[0] = element;}

   public T find (T target) throws ElementNotFound Exception {
      T temp = null; boolean found = false;
      for (int ct = 0; ct < count && !found; ct++)
         if (target.equals (tree[ct])) {
            found = true; temp = tree[ct];}
      if (!found) throw new ElementNotFoundException ( );
      return temp;}}
```

Inorder Iterators

- We can make an iterator by copying the elements to a list using inorder, then creating an iterator for the list. The method `inorder` copies the subtree under `node` into `list`, an unordered list.

```
// in LinkedBinaryTree<T>
protected void
     inorder (BinaryTreeNode<T> node, ArrayList<T> list) {
   if (node != null) {
      inorder (node.left, list);
      list.addToRear (node.element);
      inorder (node.right, list);}

// in ArrayBinaryTree<T>
protected void inorder (int node, ArrayList<T> list) {
   if (node < tree.length)
      if (tree[node] != null) {
         inorder (node*2 + 1, list);
         list.addToRear (tree[node]);
         inorder (node*2 + 2. list);}
```